Pyinterpolate

Release 0.5.1

Szymon Moliński

Feb 19, 2024

CONTENTS

CHAPTER

ONE

VERSION 0.5.1 - *MYKOLAIV*

Note: The last documentation update: *2024-02-19*

Pyinterpolate is the Python library for **geostatistics**. The package provides access to spatial statistics tools used in various studies. This package helps you **interpolate spatial data** with the *Kriging* technique.

If you're:

- GIS expert,
- geologist,
- mining engineer,
- ecologist,
- public health specialist,
- data scientist.

Then this package may be useful for you. You could use it for:

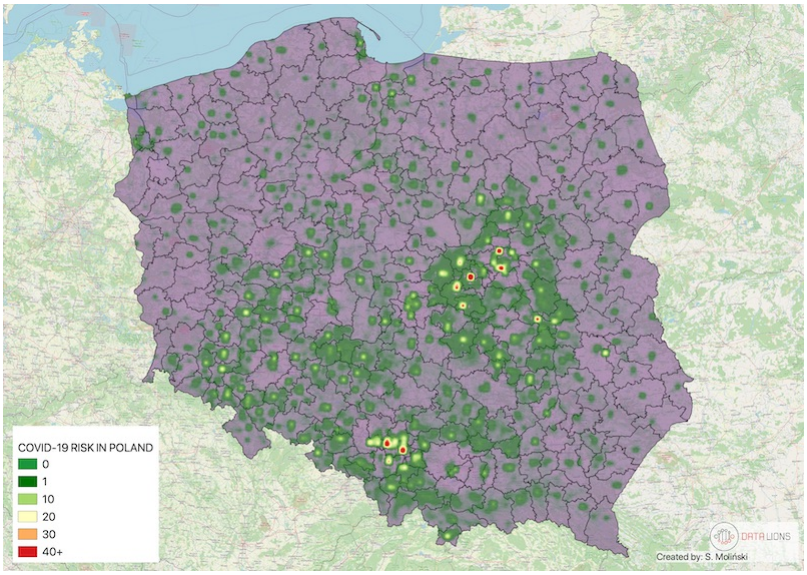
- spatial interpolation and spatial prediction,
- alone or with machine learning libraries,
- for point observations and aggregated data.

Pyinterpolate allows you to perform:

1. *Ordinary Kriging* and *Simple Kriging* (spatial interpolation from points),
2. *Centroid-based Poisson Kriging* of polygons (spatial interpolation from blocks and areas),

3. *Area-to-area* and *Area-to-point Poisson Kriging* of Polygons (spatial interpolation and data deconvolution from areas to points).
4. *Inverse Distance Weighting*.
5. *Semivariogram regularization and deconvolution*.
6. *Semivariogram modeling and analysis*.

With `pyinterpolate` we can retrieve the point support model from blocks. The example is COVID-19 population at risk mapping. Countries worldwide aggregate disease data to protect the privacy of infected people. But this kind of representation introduces bias to the decision-making process. To overcome this bias, you may use Poisson Kriging. Block aggregates of COVID-19 infection rate are transformed into the point support created from population density blocks. We get the population at risk map:



CHAPTER

TWO

CONTENTS

2.1 Setup

2.1.1 Installation guidelines

The package can be installed from *conda* and *pip*. *Conda* installation requires Python >= 3.8, *pip* installation requires Python >= 3.7.

Conda

```
conda install -c conda-forge pyinterpolate
```

pip

```
pip install pyinterpolate
```

2.1.2 Installation - additional topics

Working with Notebooks

Install *pyinterpolate* along *notebook* in your conda environment:

Step 1:

```
conda create -n [NAME OF YOUR ENV]
```

Step 2:

```
conda activate [NAME OF YOUR ENV]
```

Step 3:

```
conda install -c conda-forge notebook pyinterpolate
```

Now you are able to run library from a notebook.

The libspatialindex_c.so dependency error

With Python==3.7 installation *rtree* and *GeoPandas* that are requirements for pyinterpolate may be not installed properly because your operating system does not have *libspatialindex_c.so* file. In this case install it from terminal:

Linux:

```
sudo apt install libspatialindex-dev
```

MacOS:

```
brew install spatialindex
```

2.2 Quickstart

2.2.1 Installation

Install package with *conda*:

```
conda install -c conda-forge pyinterpolate
```

2.2.2 Ordinary Kriging

The package has multiple spatial interpolation functions. The flow of analysis is usually the same for each method. The interpolation of missing value from points is the basic case. We use for it *Ordinary Kriging*.

[1.] Read and prepare data.

```
from pyinterpolate import read_txt


point_data = read_txt('dem.txt')
```

[2.] Analyze data, calculate the experimental variogram.

```
from pyinterpolate import build_experimental_variogram


search_radius = 500
max_range = 40000

experimental_semivariogram = build_experimental_variogram(input_array=point_data,
                                                          step_size=search_radius,
                                                          max_range=max_range)
```

[3.] Data transformation, fit theoretical variogram.

```
from pyinterpolate import build_theoretical_variogram


semivar = build_theoretical_variogram(experimental_variogram=experimental_semivariogram,
```

(continues on next page)

(continued from previous page)

```
                                    model_type='spherical',
                                    sill=400,
                                    rang=20000,
                                    nugget=0)
```

[4.] Interpolation.

```
from pyinterpolate import kriging


unknown_point = (20000, 65000)
prediction = kriging(observations=point_data,
                     theoretical_model=semivar,
                     points=[unknown_point],
                     how='ok',
                     no_neighbors=32)
```

[5.] Error and uncertainty analysis.

```
print(prediction)  # [predicted, variance error, lon, lat]
```

```
>> [211.23, 0.89, 20000, 60000]
```

[6.] Full code.

```
from pyinterpolate import read_txt
from pyinterpolate import build_experimental_variogram
from pyinterpolate import build_theoretical_variogram
from pyinterpolate import kriging


point_data = read_txt('dem.txt')  # x, y, value
search_radius = 500
max_range = 40000

experimental_semivariogram = build_experimental_variogram(input_array=point_data,
                                                          step_size=search_radius,
                                                          max_range=max_range)
semivar = build_theoretical_variogram(experimental_variogram=experimental_semivariogram,
                                    model_type='spherical',
                                    sill=400,
                                    rang=20000,
                                    nugget=0)
unknown_point = (20000, 65000)
prediction = kriging(observations=point_data,
                     theoretical_model=semivar,
                     points=[unknown_point],
                     how='ok',
                     no_neighbors=32)
```

2.3 Tutorials

Step-by-step tutorials, for beginners and advanced users.

2.3.1 Beginner

A.1.1 Semivariogram Estimation

Table of Contents:

1. Read point data
2. Create the experimental variogram
3. Set manually different semivariogram models
4. Set a semivariogram model automatically
5. Export model
6. Import model

Introduction

In this tutorial, we will learn how to read and prepare data for semivariogram modeling, manually set semivariogram type, and do it automatically. We will compare different semivariogram models by visualizing the outcomes of the models.

Semivariogram modeling is an initial step before we can perform spatial interpolation of unknown values with Kriging. When you complete this tutorial, you may learn how to:

- perform point Kriging (ordinary and simple),
- regularize semivariogram of areal data.

We use DEM data which is stored in a file `samples/point_data/txt/pl_dem.txt`.

Import packages and modules

[1]:
```
import numpy as np

import matplotlib.pyplot as plt

# IO - read text
from pyinterpolate import read_txt
# Experimental variogram
from pyinterpolate import build_experimental_variogram
# Theoretical Variogram
from pyinterpolate import TheoreticalVariogram, build_theoretical_variogram
```

1) Read point data

[2]:
```
DATA = 'samples/point_data/txt/pl_dem_epsg2180.txt'
dem = read_txt(DATA)
```

[3]:
```
# Look into a first few lines of data

dem[-5:, :]
```

[3]:
```
array([[2.55032701e+05, 5.47047700e+05, 6.29624100e+01],
       [2.54975796e+05, 5.45920408e+05, 2.03861294e+01],
       [2.54751735e+05, 5.41480271e+05, 4.03093376e+01],
       [2.54682103e+05, 5.40099921e+05, 2.19432678e+01],
       [2.54521994e+05, 5.36925123e+05, 5.15251350e+01]])
```

2) Create the experimental variogram

Checking spatial correlation in our dataset is the first required step. We use variograms - plots of the dissimilarity between point pairs (y-axis) along a rising distance (x-axis). There are two "types" of variograms.

- **Experimental (semi)variogram**: directly showing differences between point pairs at specific distances - lags. It could be very messy. We use it to check if there is a spatial correlation.
- **Theoretical (semi)variogram**: is a particular function applied to the experimental variogram. We fit a function from a pre-defined set to experimental semivariogram points and choose the model with the lowest error possible. We have a few models to try and only three *hyperparameters* for controlling their behavior.

We start our analysis by setting up the experimental model. We use the `build_experimental_variogram()` function, which creates for us an `ExperimentalVariogram` object. We pass to this function three parameters:

1. `input_array`: `numpy array` with coordinates and observed values, for example: `[[0, 0, 10], [0, 1, 20]]`,
2. `step_size`: we must divide our area of analysis into discrete **lags**. **The lags** are intervals where we check if the point has a neighbor. For example, if we look into the lag `500`, then we are going to compare one point with other points in a distance `(0, 1000]` from this point,
3. `max_range`: this parameter represents the possible **maximum range of spatial dependency**. This parameter should be at most half of the area extent.

[4]:
```
# Create experimental semivariogram

step_radius = 500  # meters
max_range = 10000  # meters

experimental_variogram = build_experimental_variogram(input_array=dem, step_size=step_
↪radius, max_range=max_range)
```

[5]:
```
# What is a type of experimental variogram?

type(experimental_variogram)
```

[5]:
```
pyinterpolate.variogram.empirical.experimental_variogram.ExperimentalVariogram
```

The `ExperimentalVariogram` object represents all information that we could estimate from our measurements:

- **semivariance**: dissimilarity over a distance,
- **covariance**: similarity over a distance,
- **variance**: non-spatial variance of all points.

The class has two useful methods:

1. We can print object statistics with the `print()` function,
2. or we can plot *semivariance*, *covariance*, and *variance* and check how they behave with the `.plot()` method.

Let's check both!

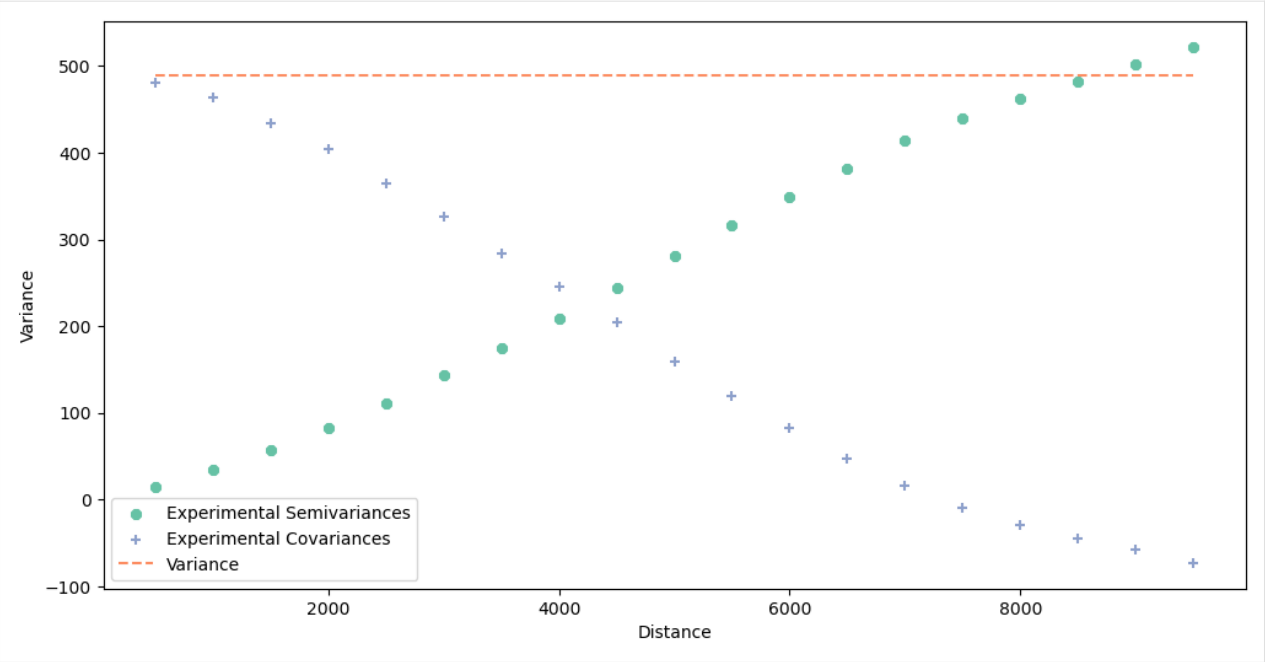
[6]:
```
print(experimental_variogram)
```

```
+--------+--------------------+---------------------+--------------------+
|  lag   |    semivariance    |      covariance     |    var_cov_diff    |
+--------+--------------------+---------------------+--------------------+
| 500.0  | 14.716058620972868 |  480.99904783812127 | 8.821278469608444  |
| 1000.0 | 34.85400868491261  |  464.2196256667607  | 25.600700640969023 |
| 1500.0 | 57.712760007769795 |  433.73651582698574 | 56.08381048074398  |
| 2000.0 | 82.49483539006093  |  403.5799288984297  | 86.24039740929999  |
| 2500.0 | 111.4449493651956  |  364.9336781724029  | 124.88664813532682 |
| 3000.0 | 143.4007980025356  |  325.6459001433955  | 164.1744261643342  |
| 3500.0 | 175.10704725708965 |  284.4093681493841  | 205.41095815834564 |
| 4000.0 | 209.3522343651705  |  245.59671700665646 | 244.22360930107325 |
| 4500.0 | 244.46785072173265 |  204.4305639103993  | 285.3897623973304  |
| 5000.0 | 281.0897575749926  |  159.61259251127132 | 330.2077337964584  |
| 5500.0 | 315.67549229122216 |  118.99201963546533 | 370.8283066722644  |
| 6000.0 | 349.5718260502048  |  81.90206675127321  | 407.9182595564565  |
| 6500.0 | 381.7274369750636  |  47.01829083567338  | 442.80203547205633 |
| 7000.0 | 413.81104972218697 |  15.768848642349267 | 474.05147766538045 |
| 7500.0 |  439.311684307325  |  -9.119116076399862 | 498.93944238412956 |
| 8000.0 | 461.6911951584344  | -29.173423730506535 | 518.9937500382363  |
| 8500.0 | 482.51366131979273 |  -44.75615663700621 |  534.576482944736  |
| 9000.0 | 501.63014408854025 | -58.013038178981994 | 547.8333644867117  |
| 9500.0 | 522.2111025684128  |  -72.92100663867586 | 562.7413329464056  |
+--------+--------------------+---------------------+--------------------+
```

- **Lag** is a column with the lag center,
- **semivariance** is a column with a dissimilarity metric,
- **covariance** is a column with a similarity metric,
- **var_cov_diff** is a difference between the variance and the covariance; ideally, it should equal the semivariance. But it is possible only if a spatial process is second-order stationary. It is a rare situation in real-world applications. Usually, this value is very close to the semivariance.

We can plot semivariance and covariance to understand their relation.

[7]:
```
experimental_variogram.plot(plot_semivariance=True, plot_covariance=True, plot_
↪variance=True)
```



Our plot shows three objects:

1. **Circles** represent *semivariance*,
2. **Plus signs** represent *covariance*,
3. **The dashed line** is a *variance*.

We see here that semivariance and covariance are mirrored. What does it mean? It is normal behavior, and we should expect it - semivariance and covariance have symmetrical trends (they differ in a sign). We can read it as:

- with **semivariance**, the dissimilarity between point pairs over a distance increases,
- **covariance** shows that the similarity between point pairs over a distance decreases.

In the best-case scenario (data is stationary, e.g., mean and variance don't change with a distance), variance and covariance should be equal to semivariance. That's why our `ExperimentalVariogram` object prints the difference between those two. We can quickly check if the process we measure is stationary.

With the experimental variogram, we can create a theoretical model of semivariance.

3. Set manually different semivariogram models

With an experimental variogram, we can start modeling theoretical function that optimally describes observed data. Our role is to choose the modeling function and to set three hyperparameters: nugget, sill, and range. You can read more about semivariogram models here: Geostatistics: Theoretical Variogram Models.

Semivariogram has three basic properties:

- **nugget**: the initial value at a zero distance. In most cases, it is zero, but sometimes it represents a bias in observations.
- **sill**: a distance where the semivariogram flattens and reaches approximately 95% of dissimilarity. Sometimes we cannot find a sill; for example, if differences grow exponentially, but in `pyinterpolate` it is usually set close to the variance of data,
- **range**: is a distance where a variogram reaches its sill. Larger distances are negligible for interpolation.

We are not forced to know all of them at the beginning. The package may easily derive them all with the class `TheoriticalVariogram` `autofit()` method.

We can create a theoretical model in two ways: 1. Manually, with the `build_theoretical_variogram()` function, 2. Semi-automatically, but here's the catch: we must create a `TheoreticalVariogram` object first, and in the second step, we may use the `.autofit()` method. Why is that? The reason is simple - if we want to fit a semivariogram automatically, the algorithm (or creator) assumes we know what we are doing. And with knowledge, we can control multiple parameters of the class.

We start from the `build_theoretical_variogram()` function and look into different models fitted to our data.

Models

We can choose from a set of predefined models:

- circular,
- cubic,
- exponential,
- gaussian,
- linear,
- power,
- spherical.

We will set the following:

- **sill** to the experimental variogram variance,
- **nugget** to zero,
- **range** to 8000 (we see in the experimental variogram plot that the semivariance *flattens* around this range, and it becomes close to the variance).

[8]:
```
sill = experimental_variogram.variance
nugget = 0
var_range = 8000
```

[9]:
```
# circular

circular_model = build_theoretical_variogram(experimental_variogram=experimental_
↪variogram,
                                             model_name='circular',
                                             sill=sill,
                                             rang=var_range,
                                             nugget=nugget)
```

[10]:
```
print(circular_model)
```

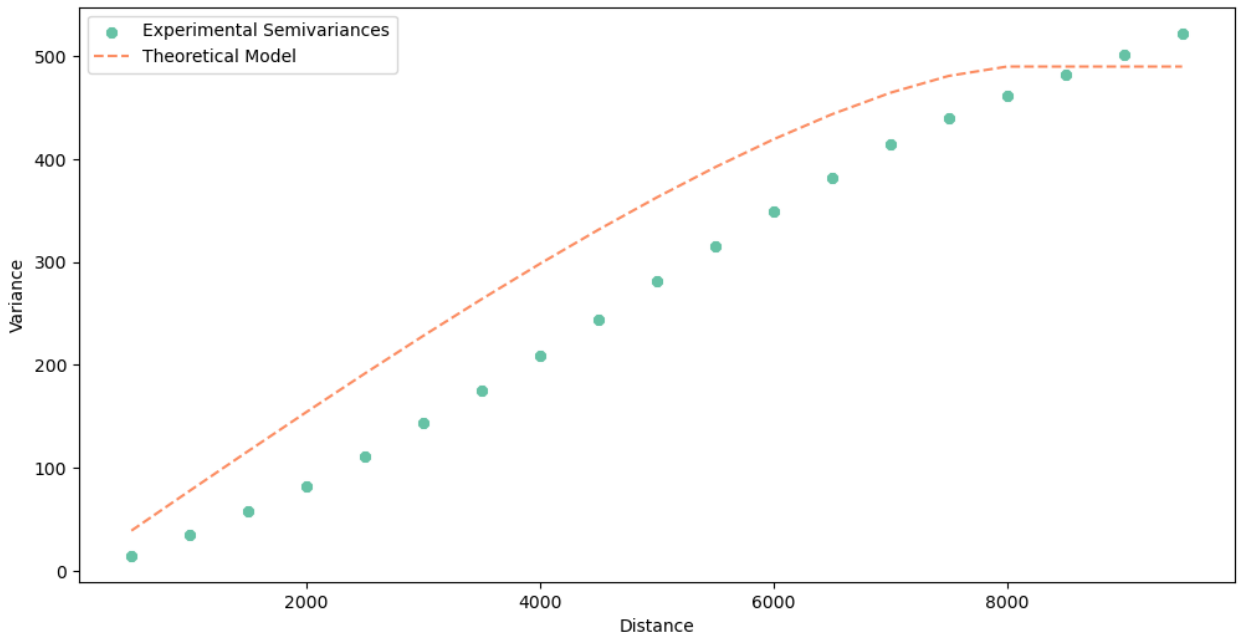
```
* Selected model: Circular model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: -52.60629816538764
```

(continues on next page)

(continued from previous page)

```
* Mean RMSE: 63.00280857591535
* Error-lag weighting method: None


+--------+--------------------+--------------------+----------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')      |
+--------+--------------------+--------------------+----------------------+
| 500.0  | 38.953271455641215 | 14.716058620972868 | -24.237212834668348 |
| 1000.0 | 77.75383379923626  | 34.85400868491261  | -42.899825114323654 |
| 1500.0 | 116.24715805742842 | 57.712760007769795 |  -58.53439804965863 |
| 2000.0 | 154.2749647378938  | 82.49483539006093  |  -71.78012934783288 |
| 2500.0 | 191.6730553536057  | 111.4449493651956  |  -80.22810598841009 |
| 3000.0 | 228.26874220429437 | 143.4007980025356  |  -84.86794420175877 |
| 3500.0 | 263.87764398483665 | 175.10704725708965 |   -88.770596727747  |
| 4000.0 | 298.29949183176774 | 209.3522343651705  |  -88.94725746659725 |
| 4500.0 | 331.3123650670836  | 244.46785072173265 |  -86.84451434535094 |
| 5000.0 | 362.66434202230323 | 281.0897575749926  |  -81.57458444731066 |
| 5500.0 | 392.0606537618678  | 315.67549229122216 |  -76.38516147064564 |
| 6000.0 | 419.14238179486523 | 349.5718260502048  |  -69.57055574466045 |
| 6500.0 | 443.44740428898785 | 381.7274369750636  |  -61.71996731392426 |
| 7000.0 | 464.3273586391969  | 413.81104972218697 |  -50.51630891700995 |
| 7500.0 | 480.71958538405306 |  439.311684307325  |  -41.40790107672808 |
| 8000.0 | 489.8203263077297  | 461.6911951584344  |  -28.12913114929529 |
| 8500.0 | 489.8203263077297  | 482.51366131979273 |  -7.306664987936983 |
| 9000.0 | 489.8203263077297  | 501.63014408854025 |  11.809817780810533 |
| 9500.0 | 489.8203263077297  | 522.2111025684128  |  32.39077626068308  |
+--------+--------------------+--------------------+----------------------+
```

[11]:
```
circular_model.plot(experimental=True)
```

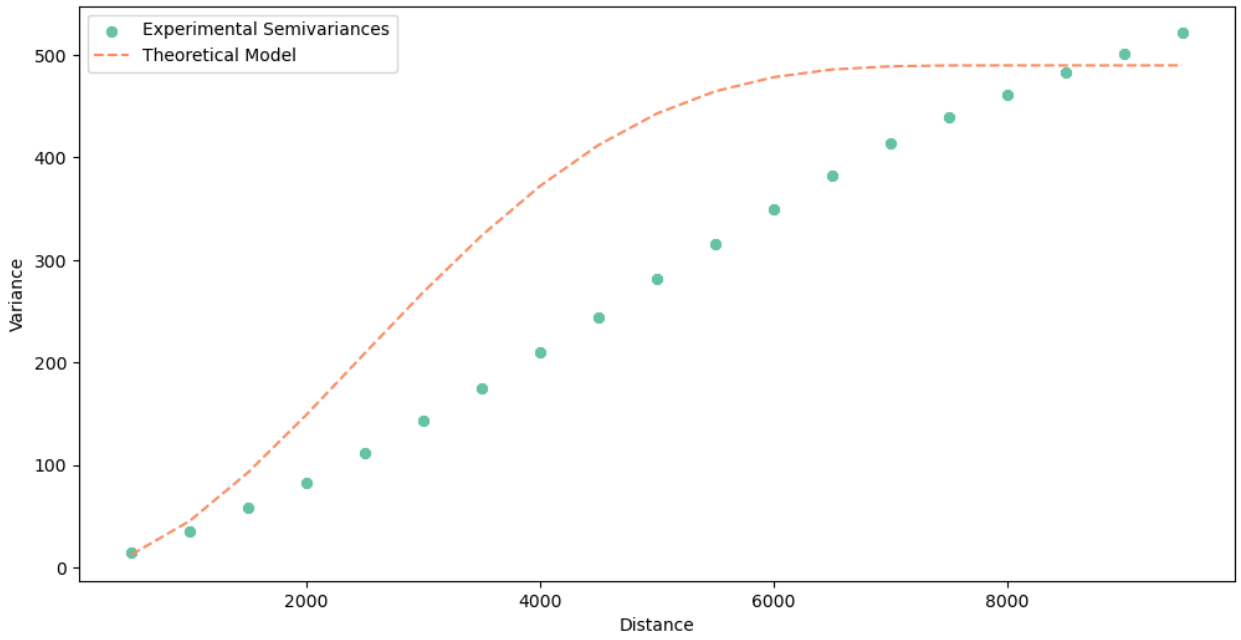


[12]:
```
# cubic

cubic_model = build_theoretical_variogram(experimental_variogram=experimental_variogram,
                                          model_name='cubic',
                                          sill=sill,
                                          rang=var_range,
                                          nugget=nugget)
```

[13]:
```
print(cubic_model)
```

```
* Selected model: Cubic model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: -77.44307814715333
* Mean RMSE: 100.93081804872125
* Error-lag weighting method: None


+--------+--------------------+--------------------+---------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')     |
+--------+--------------------+--------------------+---------------------+
| 500.0  | 12.34878902539015  | 14.716058620972868 |  2.3672695955827177 |
| 1000.0 | 45.255288629599086 | 34.85400868491261  | -10.401279944686479 |
| 1500.0 | 92.68405535286398  | 57.712760007769795 | -34.971295345094184 |
| 2000.0 | 148.9805383955685  | 82.49483539006093  |  -66.48570300550756 |
| 2500.0 | 209.04428231682044 | 111.4449493651956  |  -97.59933295162485 |
| 3000.0 | 268.48143737523486 | 143.4007980025356  | -125.08063937269927 |
| 3500.0 | 323.7296800593239  | 175.10704725708965 | -148.62263280223428 |
| 4000.0 | 372.1486463548962  | 209.3522343651705  |  -162.7964119897257 |
| 4500.0 | 412.06898029686596 | 244.46785072173265 |  -167.6011295751333 |
| 5000.0 | 442.79310035287546 | 281.0897575749926  | -161.70334277788288 |
| 5500.0 | 464.5407861861308  | 315.67549229122216 | -148.86529389490863 |
| 6000.0 | 478.33268834485466 | 349.5718260502048  | -128.76086229464988 |
| 6500.0 | 485.8048634257556  | 381.7274369750636  | -104.07742645069203 |
| 7000.0 |  488.947437258918  | 413.81104972218697 |  -75.13638753673104 |
| 7500.0 | 489.7604986615125  |  439.311684307325  | -50.448814354187505 |
| 8000.0 | 489.8203263077297  | 461.6911951584344  |  -28.12913114929529 |
| 8500.0 | 489.8203263077297  | 482.51366131979273 |  -7.306664987936983 |
| 9000.0 | 489.8203263077297  | 501.63014408854025 |  11.809817780810533 |
| 9500.0 | 489.8203263077297  | 522.2111025684128  |  32.39077626068308  |
+--------+--------------------+--------------------+---------------------+
```

[14]:
```
cubic_model.plot()
```



[15]:
```
# Exponential model

exponential_model = build_theoretical_variogram(experimental_variogram=experimental_
↪variogram,
                                                model_name='exponential',
                                                sill=sill,
                                                rang=var_range,
                                                nugget=nugget)
print(exponential_model)
```

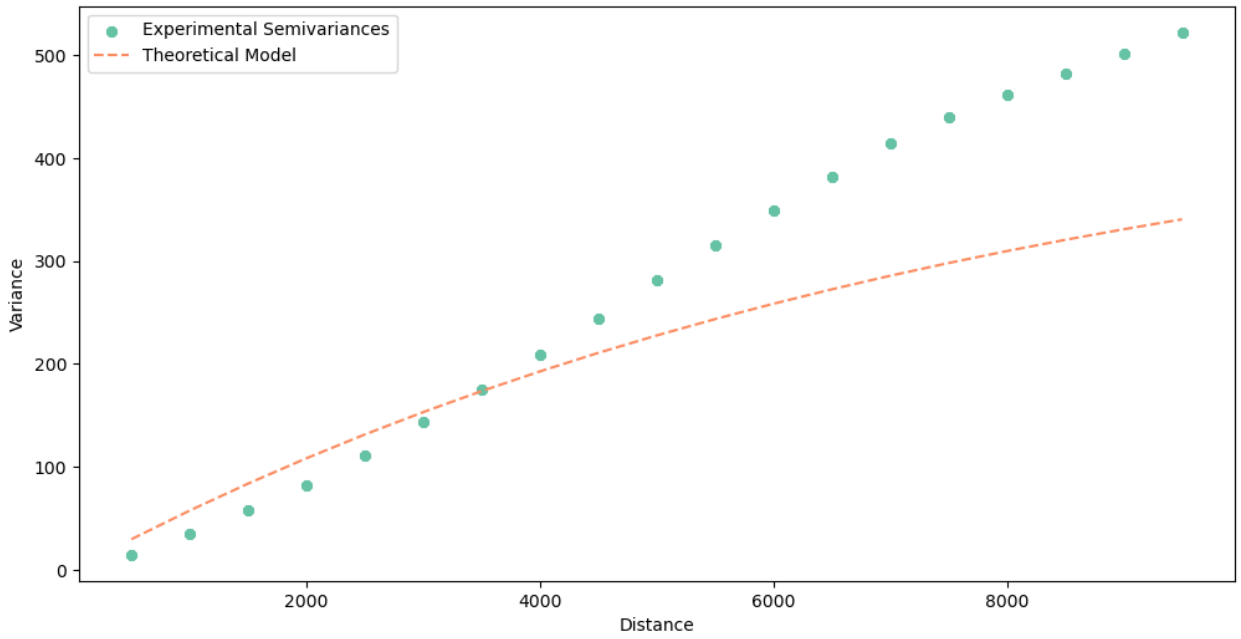
```
* Selected model: Exponential model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: 62.87847144716388
* Mean RMSE: 97.41020702761479
* Error-lag weighting method: None


+--------+--------------------+--------------------+----------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')      |
+--------+--------------------+--------------------+----------------------+
| 500.0  | 29.676713342689204 | 14.716058620972868 | -14.960654721716336 |
| 1000.0 | 57.555405518182404 | 34.85400868491261  | -22.701396833269797 |
| 1500.0 | 83.74501312199658  | 57.712760007769795 | -26.032253114226783 |
| 2000.0 | 108.34787261497875 | 82.49483539006093  | -25.853037224917827 |
| 2500.0 | 131.46012020525072 | 111.4449493651956  |  -20.01517084005512 |
| 3000.0 | 153.17206750253146 | 143.4007980025356  |  -9.771269499995867 |
| 3500.0 | 173.56855441271478 | 175.10704725708965 |  1.538492844374872  |
| 4000.0 | 192.72928065164504 | 209.3522343651705  |  16.622953713525447 |
```

(continues on next page)

(continued from previous page)

```
| 4500.0 | 210.72911717348904 | 244.46785072173265 |   33.7387335482436  |
| 5000.0 | 227.63839873061636 | 281.0897575749926  |  53.451358844376216 |
| 5500.0 | 243.5231987081728  | 315.67549229122216 |  72.15229358304936  |
| 6000.0 | 258.44558730726845 | 349.5718260502048  |  91.12623874293632  |
| 6500.0 | 272.4638740856379  | 381.7274369750636  |  109.26356288942571 |
| 7000.0 | 285.63283580350344 | 413.81104972218697 |  128.17821391868353 |
| 7500.0 |  298.003930464957  |  439.311684307325  |  141.30775384236796 |
| 8000.0 | 309.6254983912286  | 461.6911951584344  |  152.06569676720585 |
| 8500.0 | 320.54295111154215 | 482.51366131979273 |  161.97071020825058 |
| 9000.0 | 330.79894880965327 | 501.63014408854025 |  170.83119527888698 |
| 9500.0 | 340.4335670194438  | 522.2111025684128  |   181.777535548969  |
+--------+--------------------+--------------------+---------------------+
```

[16]:
```
exponential_model.plot()
```



[17]:
```
# Gaussian model

gaussian_model = build_theoretical_variogram(experimental_variogram=experimental_
↪variogram,
                                             model_name='gaussian',
                                             sill=sill,
                                             rang=var_range,
                                             nugget=nugget)
print(gaussian_model)
```

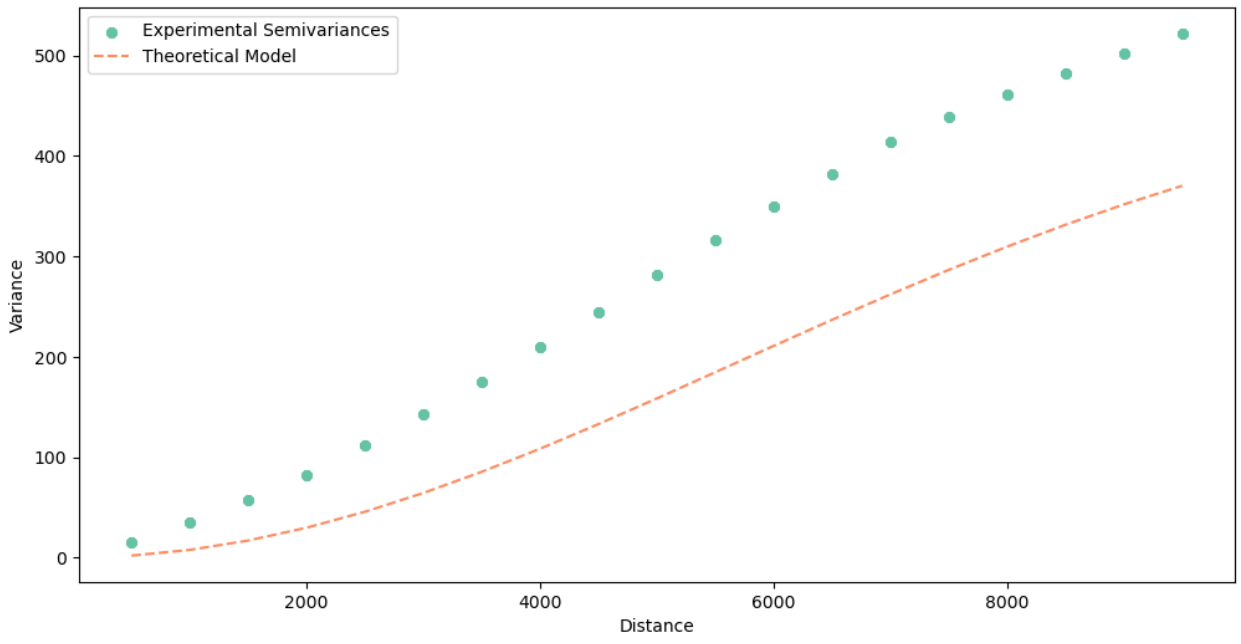
```
* Selected model: Gaussian model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: 106.76795548020137
```

(continues on next page)

(continued from previous page)

```
* Mean RMSE: 116.3487709038113
* Error-lag weighting method: None


+--------+--------------------+--------------------+--------------------+
|  lag   |    theoretical     |    experimental    |    bias (y-y')     |
+--------+--------------------+--------------------+--------------------+
| 500.0  | 1.9096284783003161 | 14.716058620972868 | 12.806430142672552 |
| 1000.0 | 7.593960284943274  | 34.85400868491261  | 27.26004839996933  |
| 1500.0 | 16.92106251490172  | 57.712760007769795 | 40.791697492868074 |
| 2000.0 | 29.676713342689204 | 82.49483539006093  | 52.81812204737172  |
| 2500.0 | 45.57258050587133  | 111.4449493651956  | 65.87236885932427  |
| 3000.0 | 64.25705194799532  | 143.4007980025356  | 79.14374605454027  |
| 3500.0 | 85.32815073541408  | 175.10704725708965 | 89.77889652167556  |
| 4000.0 | 108.34787261497875 | 209.3522343651705  | 101.00436175019173 |
| 4500.0 | 132.85723424545685 | 244.46785072173265 | 111.6106164762758  |
| 5000.0 | 158.39131498993746 | 281.0897575749926  | 122.69844258505512 |
| 5500.0 | 184.49361349105533 | 315.67549229122216 | 131.18187880016683 |
| 6000.0 | 210.72911717348904 | 349.5718260502048  | 138.84270887671573 |
| 6500.0 | 236.69559082440182 | 381.7274369750636  | 145.03184615066178 |
| 7000.0 | 262.0327201487193  | 413.81104972218697 | 151.77832957346766 |
| 7500.0 | 286.42888738780334 |  439.311684307325  | 152.88279691952164 |
| 8000.0 | 309.6254983912286  | 461.6911951584344  | 152.06569676720585 |
| 8500.0 | 331.41891440687124 | 482.51366131979273 | 151.0947469129215  |
| 9000.0 | 351.66015926974364 | 501.63014408854025 | 149.9699848187966  |
| 9500.0 | 370.2526675939889  | 522.2111025684128  | 151.95843497442388 |
+--------+--------------------+--------------------+--------------------+
```

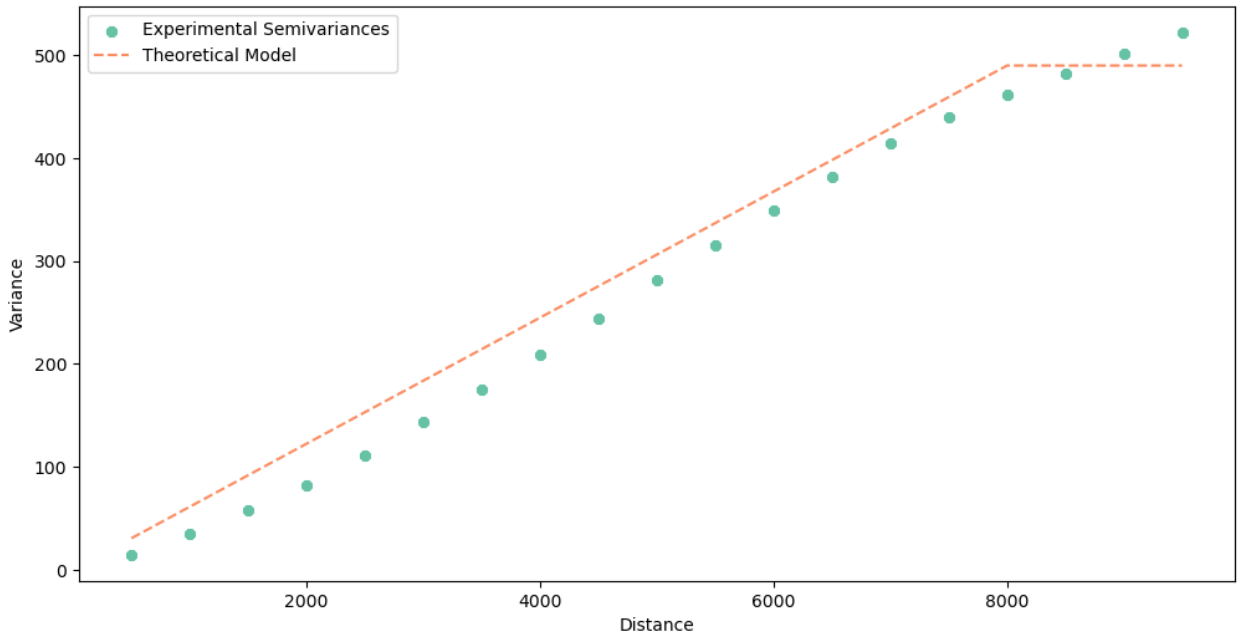
[18]:
```
gaussian_model.plot()
```



[19]:
```
# Linear model

linear_model = build_theoretical_variogram(experimental_variogram=experimental_variogram,
                                            model_name='linear',
                                            sill=sill,
                                            rang=var_range,
                                            nugget=nugget)
print(linear_model)
```

```
* Selected model: Linear model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: -21.58683474038296
* Mean RMSE: 28.218692107535404
* Error-lag weighting method: None


+--------+--------------------+--------------------+---------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')     |
+--------+--------------------+--------------------+---------------------+
| 500.0  | 30.613770394233107 | 14.716058620972868 | -15.897711773260239 |
| 1000.0 | 61.227540788466214 | 34.85400868491261  | -26.373532103553607 |
| 1500.0 | 91.84131118269931  | 57.712760007769795 |  -34.12855117492952 |
| 2000.0 | 122.45508157693243 | 82.49483539006093  |  -39.9602461868715  |
| 2500.0 | 153.06885197116554 | 111.4449493651956  | -41.623902605969946 |
| 3000.0 | 183.68262236539863 | 143.4007980025356  | -40.281824362863034 |
| 3500.0 | 214.29639275963174 | 175.10704725708965 | -39.189345502542096 |
| 4000.0 | 244.91016315386486 | 209.3522343651705  |  -35.55792878869437 |
| 4500.0 | 275.52393354809794 | 244.46785072173265 | -31.056082826365298 |
| 5000.0 | 306.1377039423311  | 281.0897575749926  |  -25.04794636733851 |
| 5500.0 | 336.7514743365642  | 315.67549229122216 | -21.075982045342016 |
| 6000.0 | 367.36524473079726 | 349.5718260502048  |  -17.79341868059248 |
| 6500.0 | 397.9790151250304  | 381.7274369750636  | -16.251578149966804 |
| 7000.0 | 428.5927855192635  | 413.81104972218697 | -14.781735797076522 |
| 7500.0 | 459.20655591349663 |  439.311684307325  | -19.894871606171648 |
| 8000.0 | 489.8203263077297  | 461.6911951584344  |  -28.12913114929529 |
| 8500.0 | 489.8203263077297  | 482.51366131979273 |  -7.306664987936983 |
| 9000.0 | 489.8203263077297  | 501.63014408854025 |  11.809817780810533 |
| 9500.0 | 489.8203263077297  | 522.2111025684128  |  32.39077626068308  |
+--------+--------------------+--------------------+---------------------+
```

[20]:
```
linear_model.plot()
```



[21]:
```
# Power model

power_model = build_theoretical_variogram(experimental_variogram=experimental_variogram,
                                          model_name='power',
                                          sill=sill,
                                          rang=var_range,
                                          nugget=nugget)
print(power_model)
```

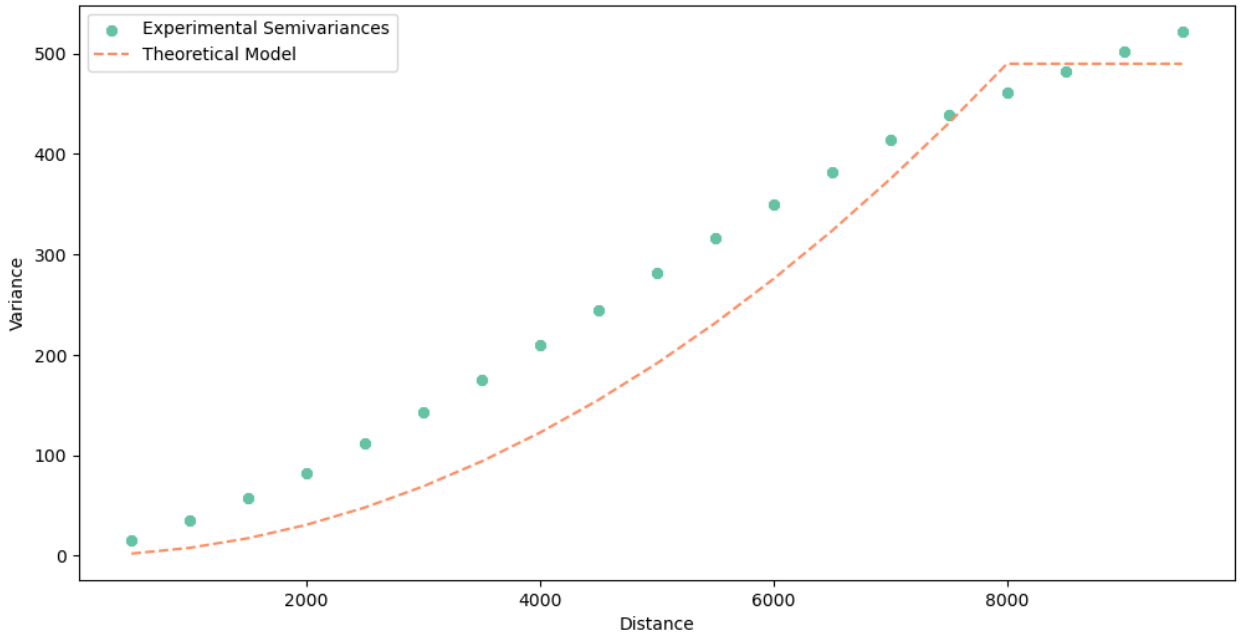
```
* Selected model: Power model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: 46.89133587829636
* Mean RMSE: 58.334699200244835
* Error-lag weighting method: None


+--------+--------------------+--------------------+--------------------+
|  lag   |    theoretical     |    experimental    |    bias (y-y')     |
+--------+--------------------+--------------------+--------------------+
| 500.0  | 1.9133606496395692 | 14.716058620972868 | 12.802697971333298 |
| 1000.0 | 7.653442598558277  | 34.85400868491261  | 27.200566608635433 |
| 1500.0 | 17.22024584675612  | 57.712760007769795 | 40.49251416101367  |
| 2000.0 | 30.613770394233107 | 82.49483539006093  | 51.88106499582782  |
| 2500.0 | 47.83401624098923  | 111.4449493651956  | 63.61093312420637  |
| 3000.0 | 68.88098338702449  | 143.4007980025356  | 74.51981461551111  |
| 3500.0 | 93.75467183233889  | 175.10704725708965 | 81.35237542475076  |
| 4000.0 | 122.45508157693243 | 209.3522343651705  | 86.89715278823806  |
| 4500.0 | 154.98221262080511 | 244.46785072173265 | 89.48563810092753  |
```

(continues on next page)

(continued from previous page)

```
| 5000.0 | 191.33606496395691 | 281.0897575749926  | 89.75369261103566  |
| 5500.0 | 231.5166386063879  | 315.67549229122216 | 84.15885368483427  |
| 6000.0 | 275.52393354809794 | 349.5718260502048  | 74.04789250210683  |
| 6500.0 | 323.3579497890872  | 381.7274369750636  | 58.369487185976425 |
| 7000.0 | 375.01868732935554 | 413.81104972218697 | 38.79236239283142  |
| 7500.0 | 430.50614616890306 |  439.311684307325  | 8.805538138421923  |
| 8000.0 | 489.8203263077297  | 461.6911951584344  | -28.12913114929529 |
| 8500.0 | 489.8203263077297  | 482.51366131979273 | -7.306664987936983 |
| 9000.0 | 489.8203263077297  | 501.63014408854025 | 11.809817780810533 |
| 9500.0 | 489.8203263077297  | 522.2111025684128  | 32.39077626068308  |
+--------+--------------------+--------------------+--------------------+
```

[22]:
```
power_model.plot()
```



[23]:
```
# Spherical model

spherical_model = build_theoretical_variogram(experimental_variogram=experimental_
↪variogram,
                                              model_name='spherical',
                                              sill=sill,
                                              rang=var_range,
                                              nugget=nugget)

print(spherical_model)
```

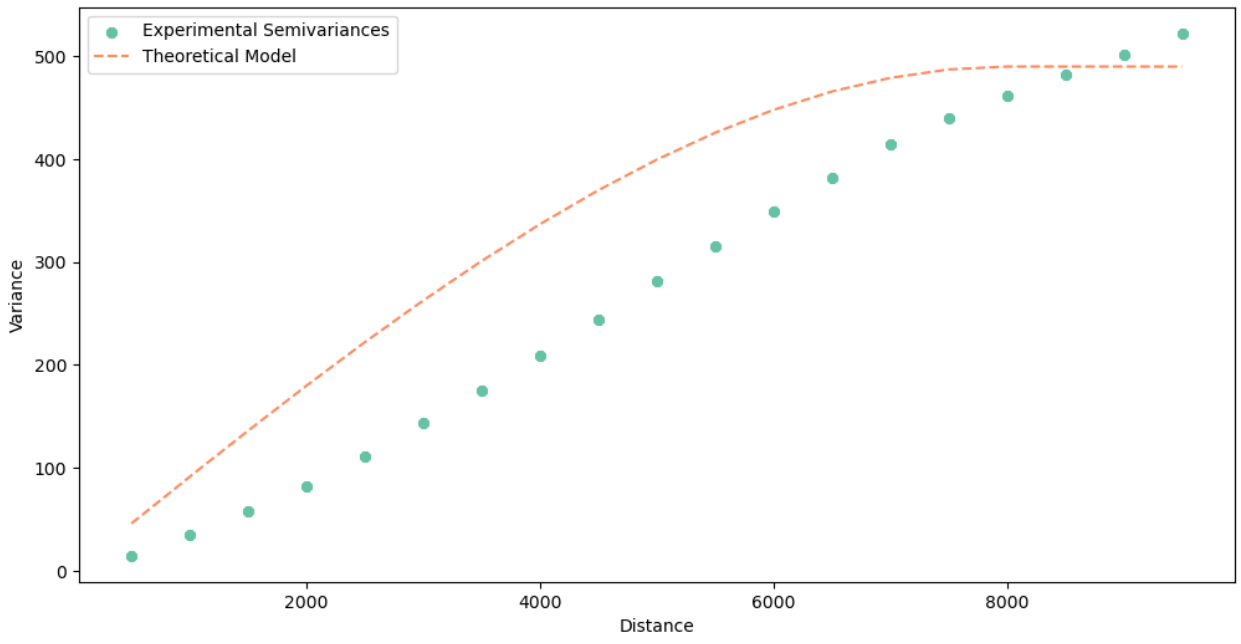
```
* Selected model: Spherical model
* Nugget: 0
* Sill: 489.8203263077297
* Range: 8000
* Spatial Dependency Strength is Unknown
* Mean Bias: -72.94546270439245
```

(continues on next page)

(continued from previous page)

```
* Mean RMSE: 87.44243944688779
* Error-lag weighting method: None


+--------+--------------------+--------------------+----------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')      |
+--------+--------------------+--------------------+----------------------+
| 500.0  | 45.86086307104843  | 14.716058620972868 |  -31.144804445007556 |
| 1000.0 | 91.36297102028944  | 34.85400868491261  |  -56.50896233537683  |
| 1500.0 | 136.1475687259156  | 57.712760007769795 |  -78.4348087181458   |
| 2000.0 | 179.85590106611951 | 82.49483539006093  |  -97.36106567605859  |
| 2500.0 | 222.12921291909373 | 111.4449493651956  | -110.68426355389813  |
| 3000.0 | 262.6087491630309  | 143.4007980025356  |  -119.2079511604953  |
| 3500.0 | 300.9357546761235  | 175.10704725708965 | -125.82870741903386  |
| 4000.0 | 336.7514743365642  | 209.3522343651705  | -127.39923997139368  |
| 4500.0 | 369.69715302254554 | 244.46785072173265 | -125.22930230081289  |
| 5000.0 | 399.4140356122601  | 281.0897575749926  |  -118.3242780372675  |
| 5500.0 | 425.54336698390046 | 315.67549229122216 |  -109.8678746926783  |
| 6000.0 | 447.7263920156592  | 349.5718260502048  |  -98.15456596545442  |
| 6500.0 | 465.6043555857289  | 381.7274369750636  |  -83.87691861066531  |
| 7000.0 | 478.8185025723022  | 413.81104972218697 |  -65.00745285011521  |
| 7500.0 | 487.0100778535716  |  439.311684307325  |  -47.69839354624662  |
| 8000.0 | 489.8203263077297  | 461.6911951584344  |  -28.12913114929529  |
| 8500.0 | 489.8203263077297  | 482.51366131979273 |  -7.306664987936983  |
| 9000.0 | 489.8203263077297  | 501.63014408854025 |  11.809817780810533  |
| 9500.0 | 489.8203263077297  | 522.2111025684128  |  32.39077626068308   |
+--------+--------------------+--------------------+----------------------+
```

[24]:
```
spherical_model.plot()
```



A quick look into plots, and we can bet that the best model is **linear**. We can read each table printed with the `print()`

method, but instead, we will read *Root Mean Squared Error* of each model and select the model with the lowest value of RMSE.

[25]:
```
models = [
    circular_model, cubic_model, exponential_model, gaussian_model, linear_model, power_
↪model, spherical_model
]

lowest_rmse = np.inf
chosen_model = ''

for _model in models:

    # Get attrs
    model_name = _model.name
    model_rmse = _model.rmse

    # Check error
    if model_rmse < lowest_rmse:
        lowest_rmse = model_rmse
        chosen_model = model_name

    # Print status
    msg = f'Model: {model_name}, RMSE: {model_rmse}'
    print(msg)

msg = f'The best model is {chosen_model} with RMSE {lowest_rmse}'
print(msg)
```

```
Model: circular, RMSE: 63.00280857591535
Model: cubic, RMSE: 100.93081804872125
Model: exponential, RMSE: 97.41020702761479
Model: gaussian, RMSE: 116.3487709038113
Model: linear, RMSE: 28.218692107535404
Model: power, RMSE: 58.334699200244835
Model: spherical, RMSE: 87.44243944688779
The best model is linear with RMSE 28.218692107535404
```

4) Set automatically semivariogram model

We can find the optimal semivariogram model automatically, but we can't do it with the `build_theoretical_variogram()` function. Instead, we use the `TheoreticalVariogram` class. Manual initialization allows us to find the best set of hyperparameters automatically.

This process has two steps:

1. The `TheoreticalVariogram` class object initialization.
2. Fitting the best model and finding the optimal set of parameters with the `.autofit()` method.

The `autofit()` method takes multiple parameters. The most important for us are:

- `experimental_variogram` - the experimental variogram object (`ExperimentalVariogram` type),
- `model_name` or `model_types` - the first argument is a name of the model. The second is a list with names to test - if you pass it, then algorithm will check every model from this list. Basic `model_name` parameters are:

  - circular,
  - cubic,
  - exponential,
  - gaussian,
  - linear,
  - power,
  - spherical,
  - (s) all: checks all available model types,
  - (s) safe: check linear, spherical, and power models.

If you want to test different set of models instead then use `model_types` with list of models, as for example: `['circular', 'power']`. You may choose any model excluding `all` and `safe` keywords which refer to multiple models.

- `nugget` - nugget (bias) of a variogram. If given, then the *nugget* is fixed to this value.
- `min_nugget` - **default = 0.0** - the minimal fraction of the variance at a distance 0 to search for the optimal nugget,
- `max_nugget` - **default = 0.5** - the maximum fraction of the variance at a distance 0 to search for the optimal nugget,
- `number_of_nuggets` - **default = 16** - how many equally spaced steps between `min_nugget` and `max_nugget` to check.
- `rang` - if given, then the `range` is fixed to this value.
- `min_range` - **default = 0.1** - the minimal fraction of a study (maximum) extent to find the optimal range.
- `max_range` - **default = 0.5** - the maximum fraction of a study (maximum) extent to find the optimal range.
- `number_of_ranges` - **default = 16** - how many equally spaced ranges are tested between `min_range` and `max_range`.
- `sill` - if given, it is fixed to this value.
- `min_sill` - **default = 0** - the minimal fraction of the dataset variance to find the *sill*.
- `max_sill` - **default = 1** - the maximum fraction of the dataset variance to find the *sill*. It should be lower or equal to 1, but it is possible to set it above. A warning is printed if `max_sill` is greater than 1.
- `number_of_sills` - **default = 16** - how many equally spaced sill values are tested between `min_sill` and `max_sill`.
- `error_estimator` - **default = 'rmse'** - Error estimation to choose the best model. Available options are:
  - rmse: Root Mean Squared Error,
  - mae: Mean Absolute Error,
  - bias: Forecast Bias,
  - smape: Symmetric Mean Absolute Percentage Error.

In the first run, we will set nugget, sill, and range as fixed values, and we will see which model algorithm chooses:

[26]:
```
semivariogram_model = TheoreticalVariogram()
```

[27]:
```
fitted = semivariogram_model.autofit(
    experimental_variogram=experimental_variogram,
    model_name='all',
    nugget=0,
    rang=var_range,
    sill=sill)
```

[28]:
```
fitted
```

[28]:
```
{'model_name': 'linear',
 'nugget': 0,
 'sill': 489.8203263077297,
 'range': 8000,
 'fitted_model': array([[ 500.        ,   30.61377039],
        [1000.        ,   61.22754079],
        [1500.        ,   91.84131118],
        [2000.        ,  122.45508158],
        [2500.        ,  153.06885197],
        [3000.        ,  183.68262237],
        [3500.        ,  214.29639276],
        [4000.        ,  244.91016315],
        [4500.        ,  275.52393355],
        [5000.        ,  306.13770394],
        [5500.        ,  336.75147434],
        [6000.        ,  367.36524473],
        [6500.        ,  397.97901513],
        [7000.        ,  428.59278552],
        [7500.        ,  459.20655591],
        [8000.        ,  489.82032631],
        [8500.        ,  489.82032631],
        [9000.        ,  489.82032631],
        [9500.        ,  489.82032631]]),
 'rmse': 28.218692107535404,
 'bias': nan,
 'mae': nan,
 'smape': nan}
```

The chosen model is **linear**, automatically set as the class parameter. The algorithm performs the same steps as we did before. It has selected a model based on the RMSE. All other parameters were untouched.

Now, we let the algorithm find the optimal sill and range, and we check if our RMSE will be better with a different pair of sill and range:

[29]:
```
fitted = semivariogram_model.autofit(experimental_variogram=experimental_variogram,␣
↪nugget=0)
```

[30]:
```
print(fitted)
```

```
{'model_name': 'linear', 'nugget': 0, 'sill': 489.8203263077297, 'range': 8990.
↪27235812866, 'fitted_model': array([[ 500.        ,   27.2416845 ],
       [1000.        ,   54.483369  ],
       [1500.        ,   81.7250535 ],
       [2000.        ,  108.966738  ],
       [2500.        ,  136.2084225 ],
```

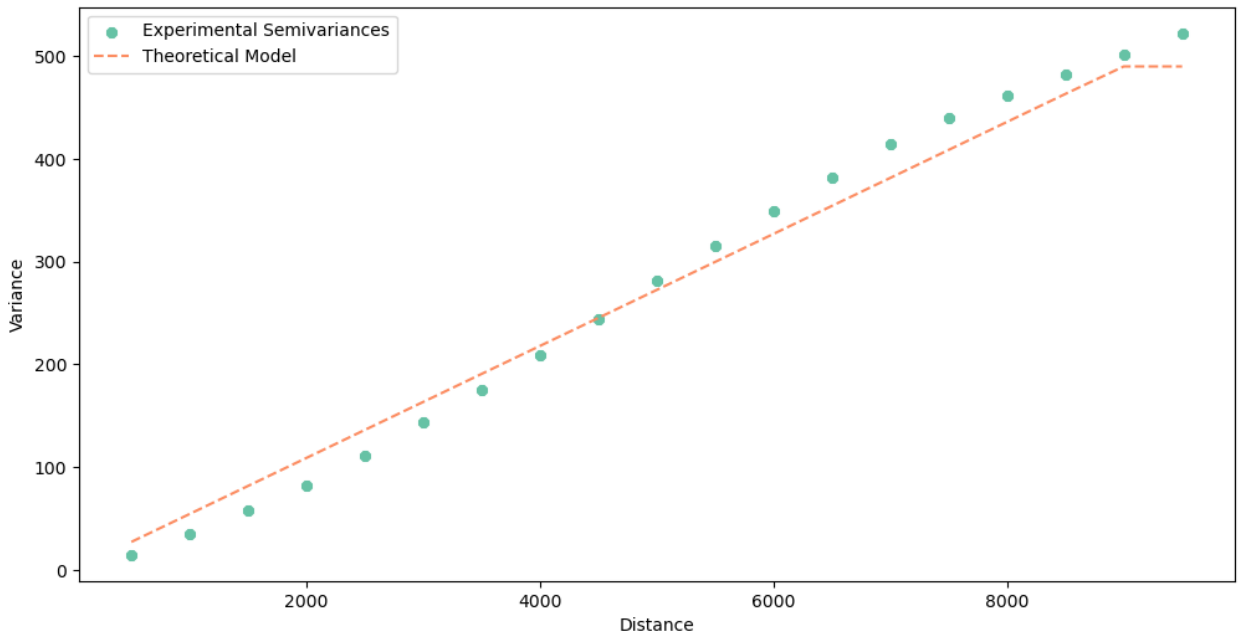
(continues on next page)

(continued from previous page)

```
       [3000.        ,  163.450107  ],
       [3500.        ,  190.6917915 ],
       [4000.        ,  217.933476  ],
       [4500.        ,  245.1751605 ],
       [5000.        ,  272.41684501],
       [5500.        ,  299.65852951],
       [6000.        ,  326.90021401],
       [6500.        ,  354.14189851],
       [7000.        ,  381.38358301],
       [7500.        ,  408.62526751],
       [8000.        ,  435.86695201],
       [8500.        ,  463.10863651],
       [9000.        ,  489.82032631],
       [9500.        ,  489.82032631]]), 'rmse': 21.744673287932994, 'bias': nan, 'mae':␣
↪nan, 'smape': nan}
```

After `.autofit()` our algorithm has chosen a different set of parameters and a model type! Let's plot it against experimental variogram:

[31]:
```
semivariogram_model.plot()
```



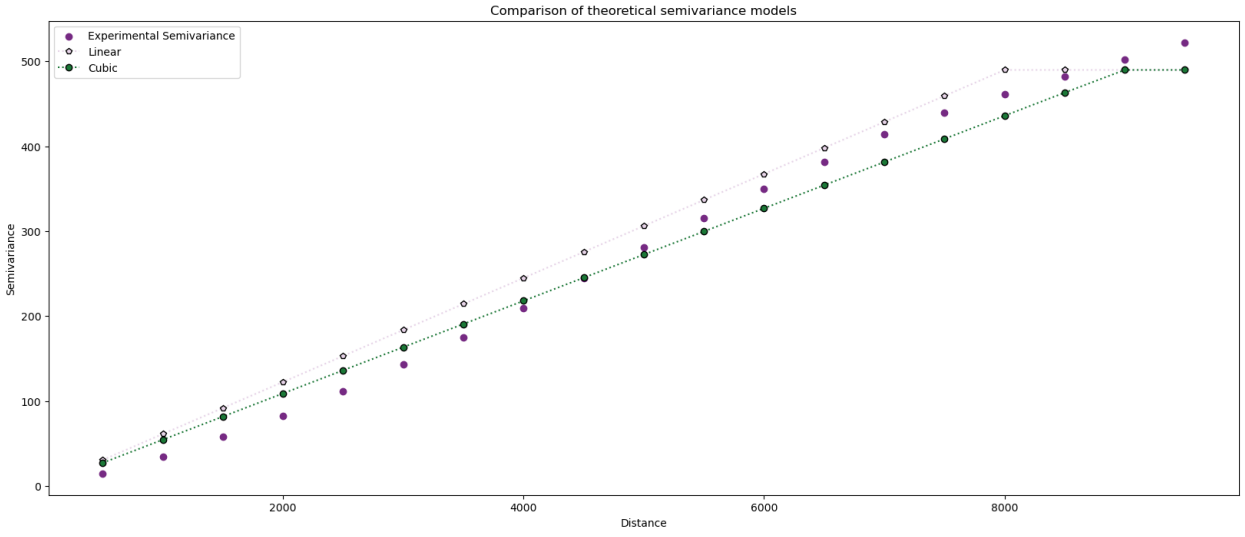
Compare this model to the linear model fitted before:

[32]:
```
_lags = semivariogram_model.lags

plt.figure(figsize=(20, 8))
plt.scatter(_lags, experimental_variogram.experimental_semivariances, color='#762a83')
↪# Experimental
plt.plot(_lags, linear_model.fitted_model[:, 1], ':p', color='#e7d4e8', mec='black')
plt.plot(_lags, semivariogram_model.fitted_model[:, 1], ':o', color='#1b7837', mec='black
↪')
```

(continues on next page)

(continued from previous page)

```
plt.title('Comparison of theoretical semivariance models')
plt.legend(['Experimental Semivariance',
            'Linear',
            'Cubic'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



5) Export model

Models could be exported and used for other purposes. It is vital for the **semivariogram regularization**. Those calculations are computationally intensive, and in production, it is not a good idea to build a complete pipeline; we divide it into two steps.

Model can be exported to `dict` with `.to_dict()` method or to `json` with `.to_json()` method:

[33]:
```
# Set spherical model

dict_model = semivariogram_model.to_dict()

# Save to json
semivariogram_model.to_json('output/semivariogram_calculation_model.json')
```

6) Import model

We can import a semivariogram model into a new **TheoreticalSemivariogram** class instance without passing the experimental semivariogram or actual data points. It is helpful for some applications focused on kriging, where we are sure that our semivariogram model fits data well.

We can import a model with two methods `.from_dict()` if we have our model parameters in the Python dictionary or `.from_json()` if model parameters are stored in a flat file:

[34]:
```
other_model_from_dict = TheoreticalVariogram()

print(other_model_from_dict)
```

```
Theoretical model is not calculated yet. Use fit() or autofit() methods to build or find␣
↪a model or import model with from_dict() or from_json() methods.
```

[35]:
```
other_model_from_dict.from_dict(dict_model)
print(other_model_from_dict)
```

```
* Selected model: Linear model
* Nugget: 0.0
* Sill: 489.8203263077297
* Range: 8990.27235812866
* Spatial Dependency Strength is Unknown
* Mean Bias: 0.0
* Mean RMSE: 0.0
* Error-lag weighting method: None
```

[36]:
```
other_model_from_json = TheoreticalVariogram()
print(other_model_from_json)
```

```
Theoretical model is not calculated yet. Use fit() or autofit() methods to build or find␣
↪a model or import model with from_dict() or from_json() methods.
```

[37]:
```
other_model_from_json.from_json('output/semivariogram_calculation_model.json')
```

[38]:
```
print(other_model_from_json)
```

```
* Selected model: Linear model
* Nugget: 0.0
* Sill: 489.8203263077297
* Range: 8990.27235812866
* Spatial Dependency Strength is Unknown
* Mean Bias: 0.0
* Mean RMSE: 0.0
* Error-lag weighting method: None
```

Where to go from here?

- A.1.2 Theoretical Models
- A.1.3 Spatial Dependence Index
- A.2.1 Directional Semivariogram
- A.2.2 Variogram Point Cloud
- A.2.3 Experimental Variogram and Variogram Point Cloud Classes
- B.1.1 Ordinary and Simple Kriging

Changelog

Date	Change description	Author
2023-08-18	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@Simon-Molinsky
2023-03-11	Spatial Dependence Index	@Simon-Molinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@Simon-Molinsky
2022-08-09	Tutorial updated to work with version 0.3.0	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within the `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-10-13	Refactored `TheoreticalSemivariogram` (name change of class attribute)	@ethmtrgt
2021-05-28	Updated paths for input/output data	@Simon-Molinsky
2021-05-11	Refactored `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting	@Simon-Molinsky

[ ]:

A.1.2 Semivariogram models

Table of Contents:

Introduction

We can find multiple semivariogram models included in the `Pyinterpolate` package. All of them are derived from literature. In this tutorial, we will compare those models on an artificial surface. We will learn the following:

- how to create an artificial surface with Python,
- what variogram models are available for us,
- the difference between variogram models.

Import packages

[1]:
```
from scipy.signal import convolve2d
from scipy.sparse import coo_matrix

from pyinterpolate import build_experimental_variogram
from pyinterpolate.variogram.theoretical.models import *

import matplotlib.pyplot as plt
```

1) Create a random surface

In the first step, we will create the artificial surface. We choose an artificial object instead of real-world observations because we have more control over variogram parameters. We want to compare different models. We should limit the number of unknowns in our dataset to a bare minimum. But don't be disappointed! Our surface won't be *boring* because we will generate it from an exciting function named logistic_map. It is the polynomial mapping that, for some parameters, presents chaotic behavior.

Logistic map is a recurrent relation of the form:

$$x_{n+1} = rx_n(1 - x_n),$$

Where: - x: is a mapped value at a step n or $n + 1$. Its state depends only on the previous step and initial set of parameters, $x \in [0, 1]$. - r: is a special parameter, and its value leads to the different behavior of a system. For us, the most important is a set of possible r values within limits $[3.5, 4]$ where the system has chaotic behavior more.

We will use the `generate_logistic_map` function that takes three parameters:

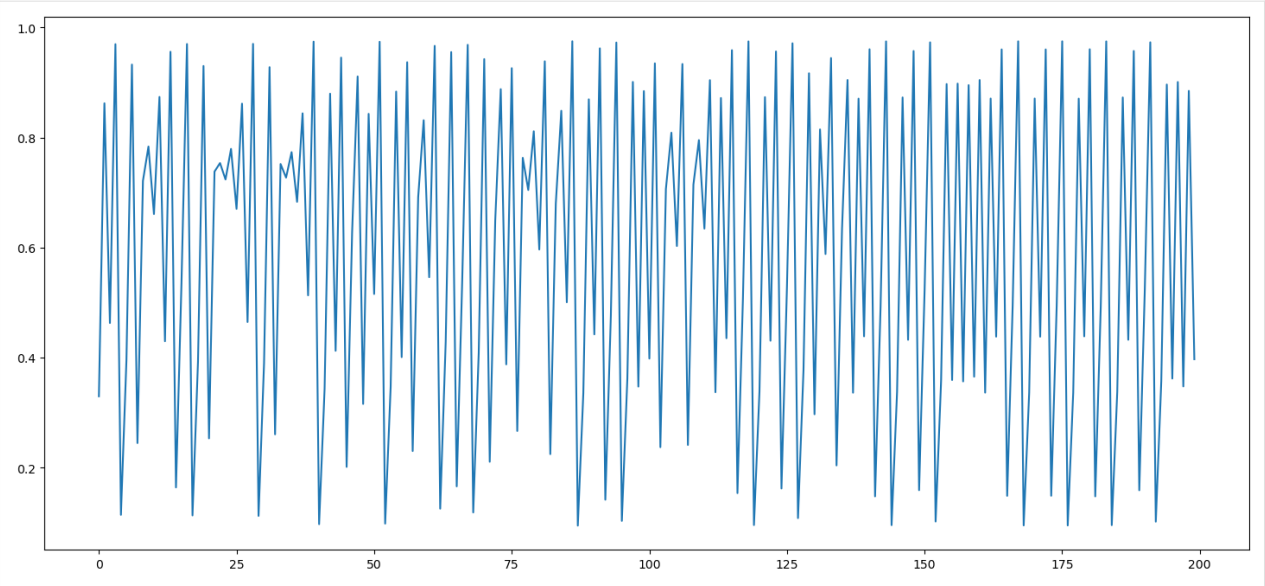
- `r`: the same as r from the **logistic map** system, we set it to **3.9**.
- `size`: the length of a generated sequence. It should be reasonably large to create a surface. We set it to **10k**.
- `initial_ratio`: initial `x` value between 0 and 1. We set it to **0.33**.

[2]:
```
# Create logistic map

def generate_logistic_map(r: float, size: int, initial_ratio: float) -> np.array:
    # rxn(1-xn)
    vals = [initial_ratio]
    for _ in range(size-1):
        new_val = r * vals[-1] * (1 - vals[-1])
        vals.append(new_val)
    return np.array(vals)
```

[3]:
```
values = generate_logistic_map(3.9, 10000, 0.33)
```

At this point, our data is a **1-D** array and we can represent it as a signal:

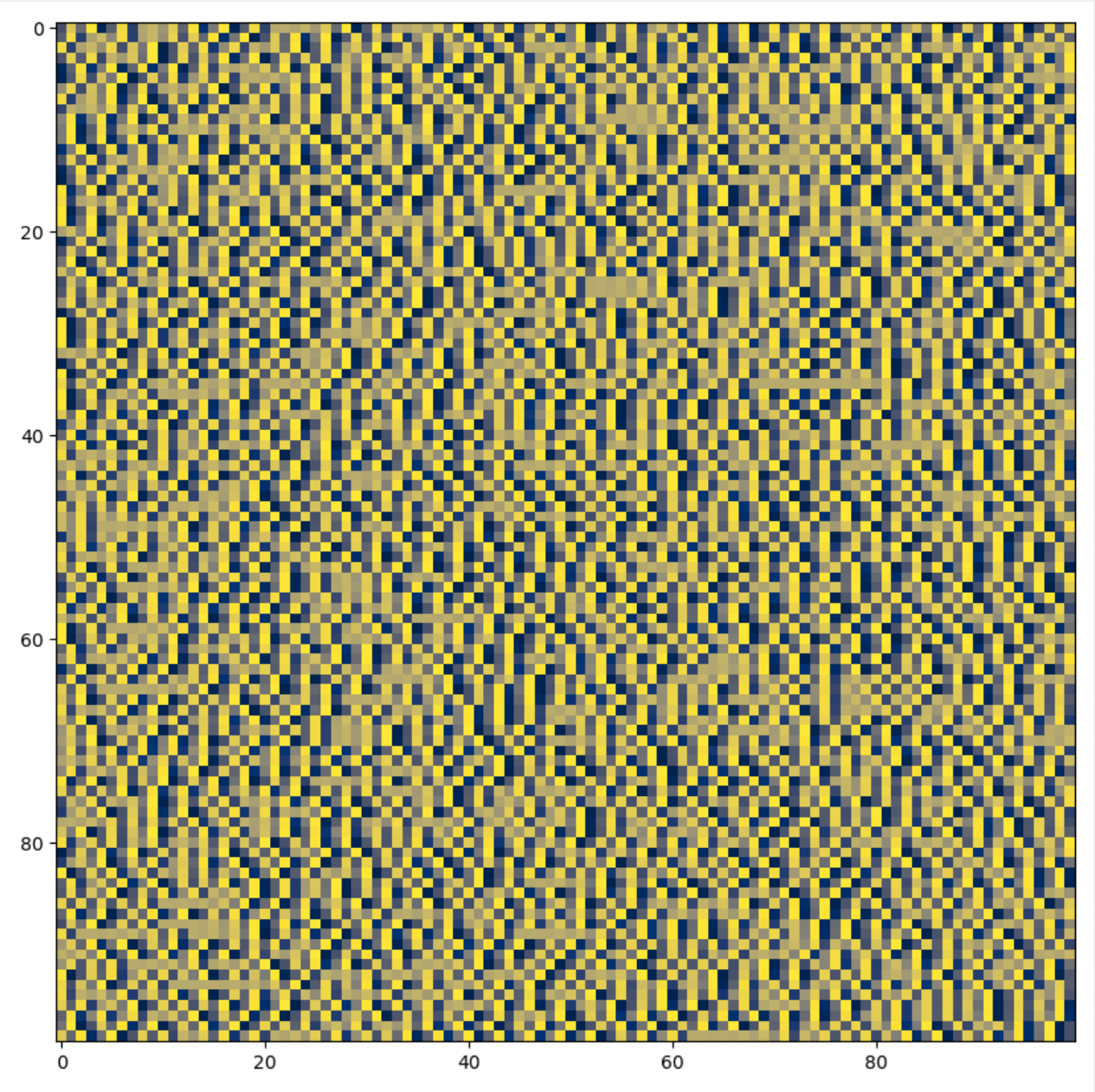
[4]:
```
plt.figure(figsize=(18, 8))
plt.plot(values[0:200])
plt.show()
```



Let's reshape this signal into a **2-D** matrix:

[5]:
```
# Create surface from array

surface = np.reshape(values, (100, 100))
```

[6]:
```
plt.figure(figsize=(10, 10))
plt.imshow(surface, cmap='cividis')
plt.show()
```



The spatial correlation of this structure is very weak. We can change it with a simple blur filter. Its size will be our **range** parameter!

[7]:
```
# Slightly blur image with simple mean filter

mean_filter = np.ones(shape=(7, 7))

surf_blurred = convolve2d(surface, mean_filter, boundary='wrap')
```

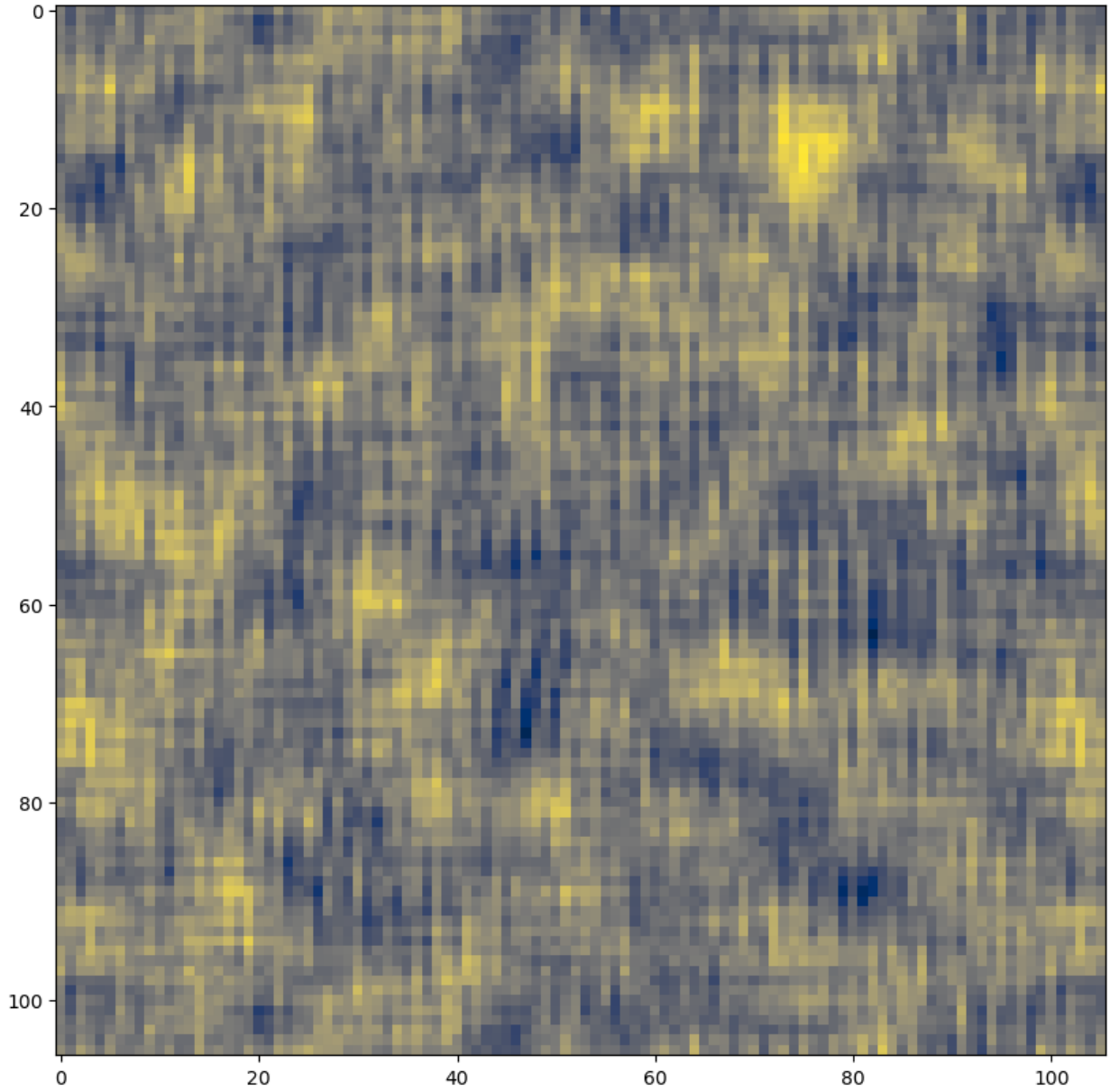
And here we are! We have our artificial surface for testing purposes!

[8]:
```
plt.figure(figsize=(10, 10))
plt.imshow(surf_blurred, cmap='cividis')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



2) Create the experimental semivariogram

We must create the experimental variogram before we start comparing theoretical models. Our 2-D surface must be transformed into an array:

```
[[coordinate x1, coordinate y1, value1],
[coordinate x2, coordinate y2, value2],
[...],]
```

The path to transform a 2D surface into pixel coordinates has two steps. In the first step, we use `scipy` to make a sparse

representation of the surface, and in the second step, we use `numpy` to transform the sparse representation into an array:

```
[9]: # Transform data into x, y, val array
```

```
[10]: sparse_data = coo_matrix(surf_blurred)
```

```
[11]: # data, col, row == value, x, y
```

```
[12]: xyval = np.asarray([sparse_data.col, sparse_data.row, sparse_data.data]).T
```

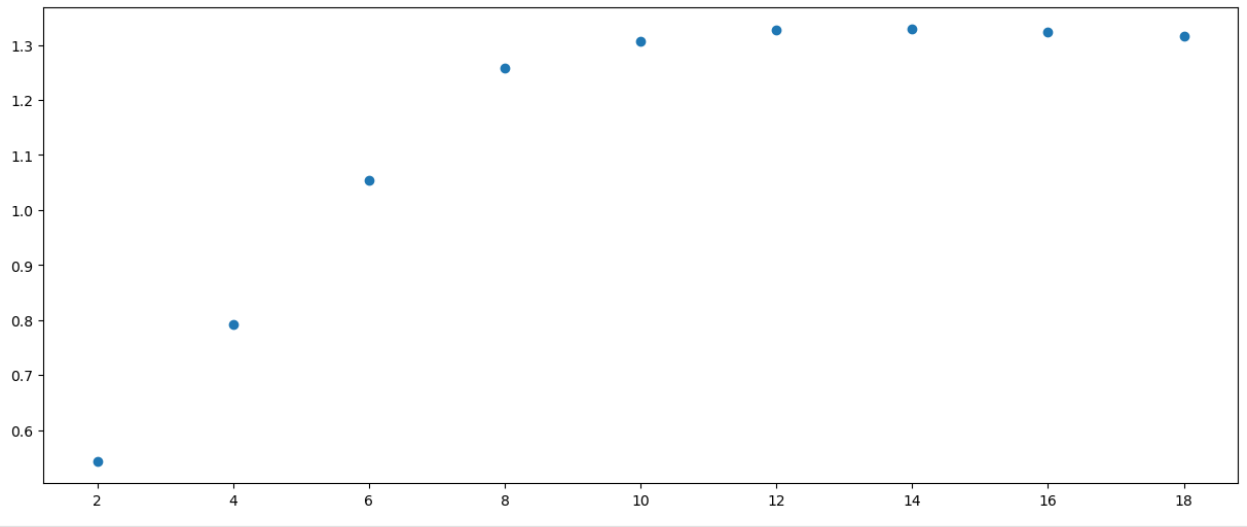
We know our data - the effective range of correlation is seven steps, and we have set the mean filter to this size. With this knowledge, we will set `step_size` into 2 units and `max_range` into 20 (way above the actual maximum range, which is 7, but we show in this way how variograms behave at a large distance).

```
[13]: # Get experimental variogram
```

```
[14]: experimental = build_experimental_variogram(xyval, 2, 20)
```

```
[15]: semivars = experimental.experimental_semivariance_array.copy()
```

```
[16]: plt.figure(figsize=(15, 6))
plt.scatter(semivars[:, 0], semivars[:, 1])
plt.show()
```



We've read the plot and decided to set:

- **nugget** to `0`,
- **sill** to `1.3`,
- **range** to `7`,
- **lags** to list of distances from 2 to 18.

We can create the theoretical variogram from scratch without calling the `build_theoretical_variogram()` function or the `TheoreticalVariogram` class. Those three parameters and the lags are all that we need for modeling.

[17]:
```
# Set nugget, sill and range
```

[18]:
```
_nugget = 0
_sill = 1.3
_range = 7
_lags = semivars[:, 0].copy()
```

3) Set all variogram models

Variogram models are imported as external functions and are not part of any object. We can use this fact, import all models, and calculate their outputs. We will calculate and show seven different theoretical models compared to the experimental variogram.

Models to calculate:

- circular,
- cubic,
- exponential,
- gaussian,
- linear,
- power,
- spherical.

[19]:
```
# Crete different models: circular, cubic, exponential, gaussian, linear, power,␣
↪spherical
```

[20]:
```
circular = circular_model(lags=_lags,
                          nugget=_nugget,
                          sill=_sill,
                          rang=_range)

cubic = cubic_model(lags=_lags,
                    nugget=_nugget,
                    sill=_sill,
                    rang=_range)

exponential = exponential_model(lags=_lags,
                                nugget=_nugget,
                                sill=_sill,
                                rang=_range)

gaussian = gaussian_model(lags=_lags,
                          nugget=_nugget,
                          sill=_sill,
                          rang=_range)

linear = linear_model(lags=_lags,
                      nugget=_nugget,
                      sill=_sill,
```

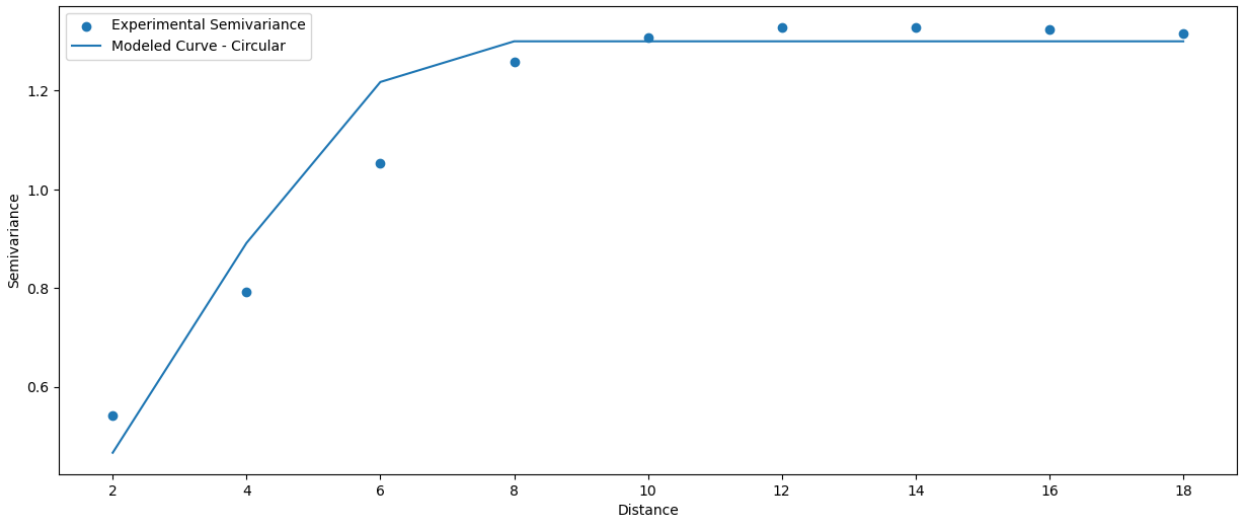
(continues on next page)

(continued from previous page)

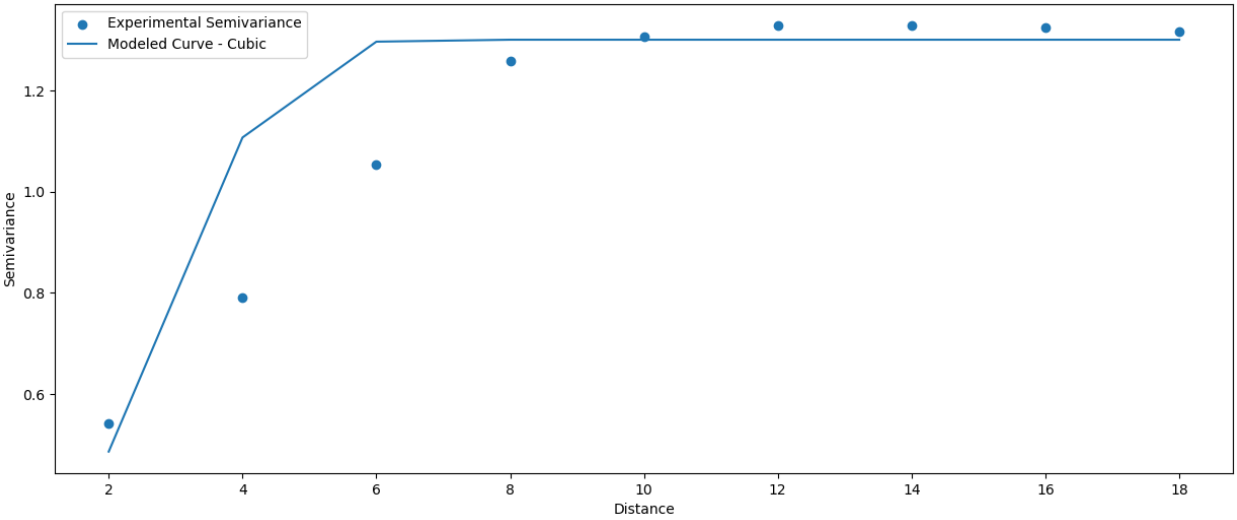
```
                      rang=_range)

power = power_model(lags=_lags,
                    nugget=_nugget,
                    sill=_sill,
                    rang=_range)

spherical = spherical_model(lags=_lags,
                            nugget=_nugget,
                            sill=_sill,
                            rang=_range)
```

With all models calculated, let's take a look at their output. Before we jump into the last step, which model works optimally with the experimental variogram? (It is a tricky question!)

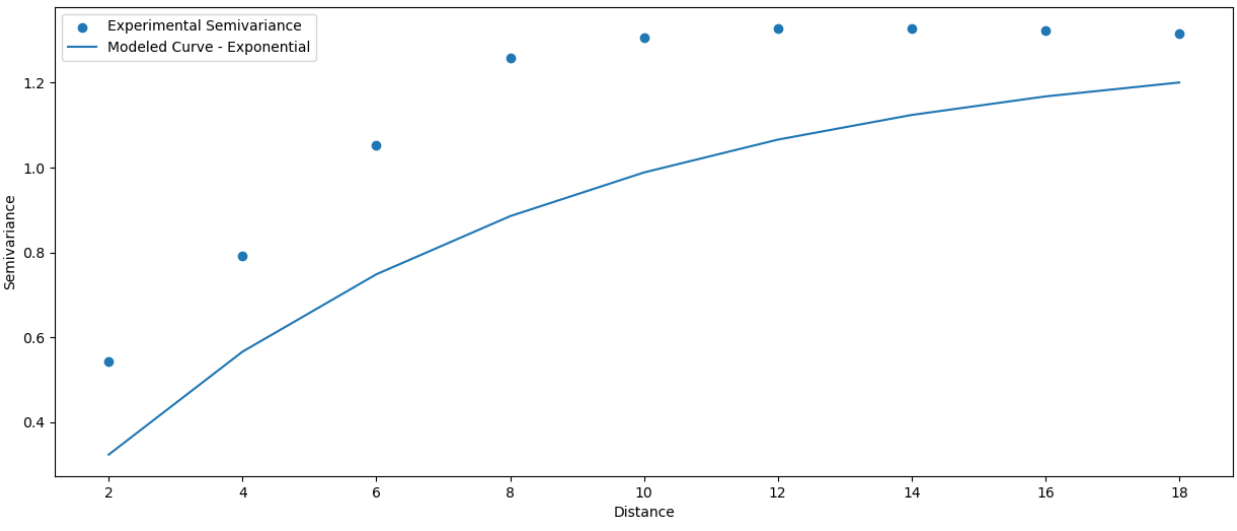
[21]:
```
# Plot circular

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, circular)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Circular'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



[22]:
```
# Plot cubic

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, cubic)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Cubic'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



[23]:
```
# Plot exponential

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, exponential)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Exponential'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```

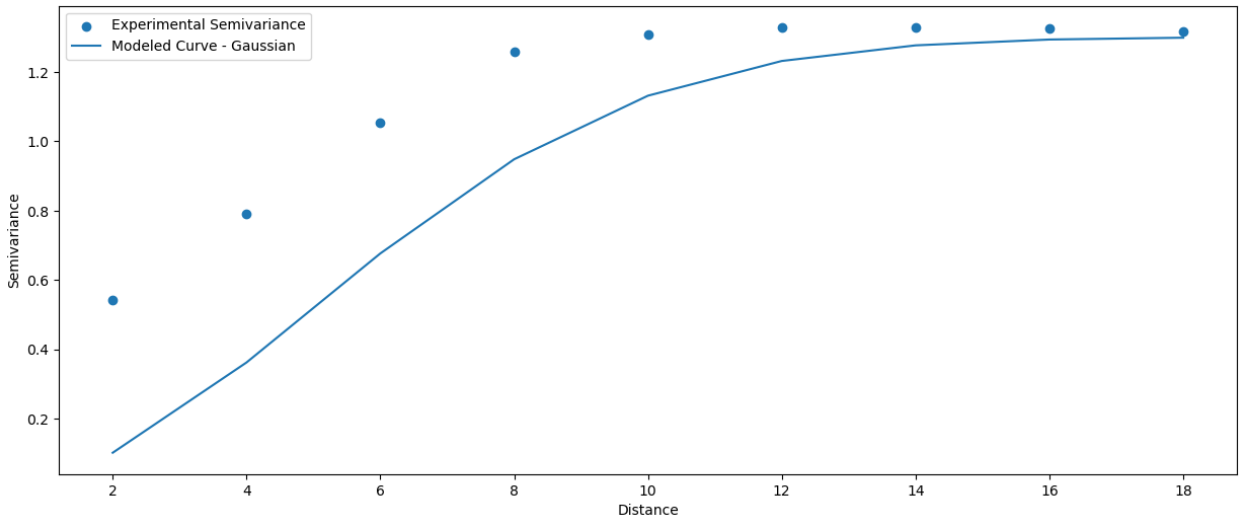


[24]:
```
# Plot gaussian

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, gaussian)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Gaussian'])
plt.xlabel('Distance')
```

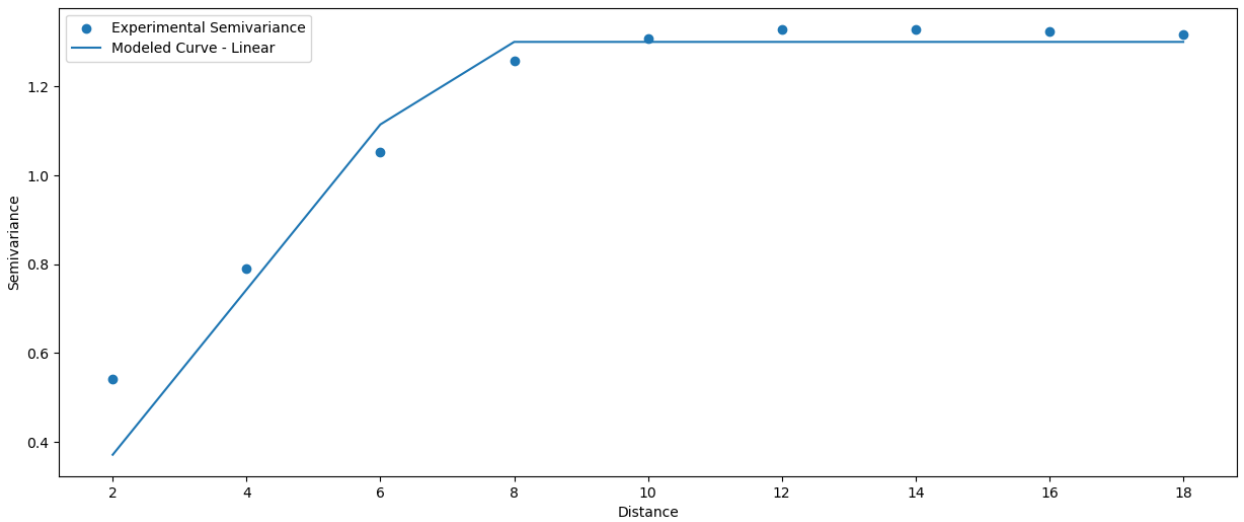
(continues on next page)

(continued from previous page)

```
plt.ylabel('Semivariance')
plt.show()
```



[25]:
```
# Plot linear

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, linear)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Linear'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```

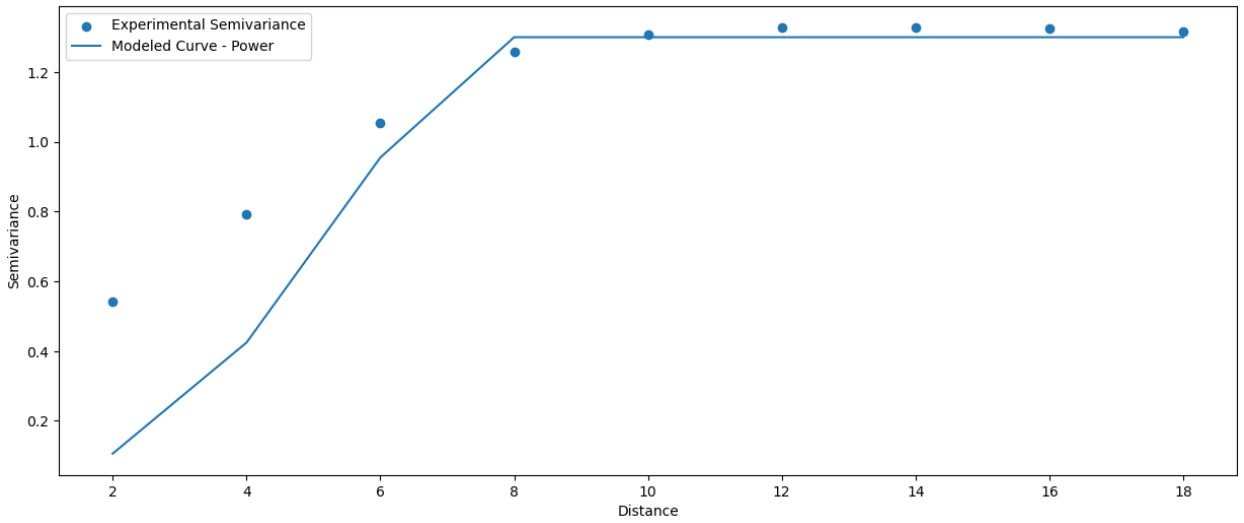


[26]:
```
# Plot power

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
```

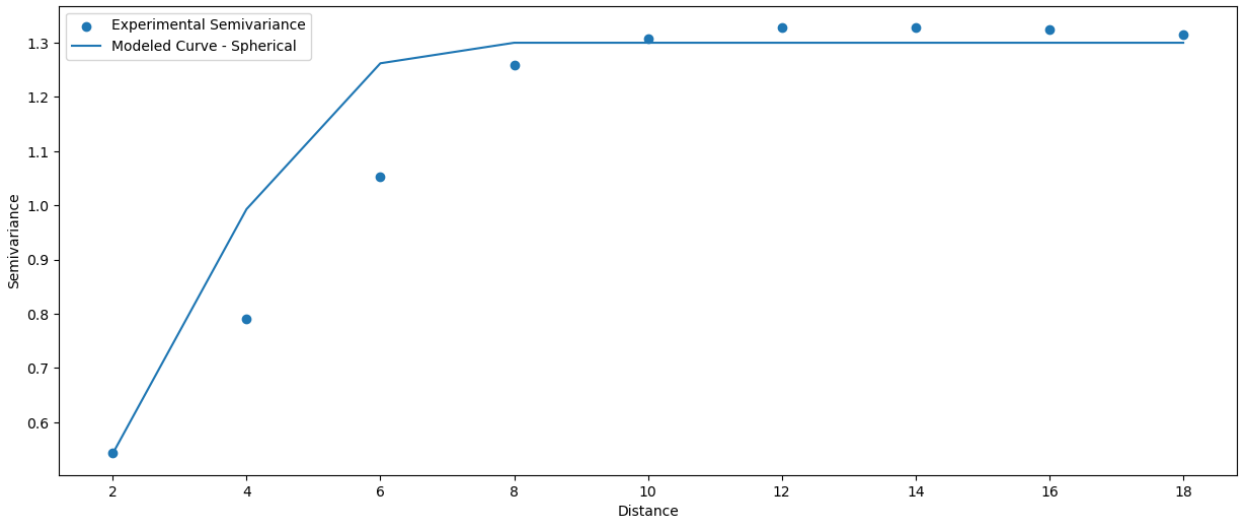
(continues on next page)

(continued from previous page)

```
plt.plot(_lags, power)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Power'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



[27]:
```
# Plot spherical

plt.figure(figsize=(15, 6))
plt.scatter(_lags, semivars[:, 1])
plt.plot(_lags, spherical)
plt.legend(['Experimental Semivariance', 'Modeled Curve - Spherical'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



What was your guess?

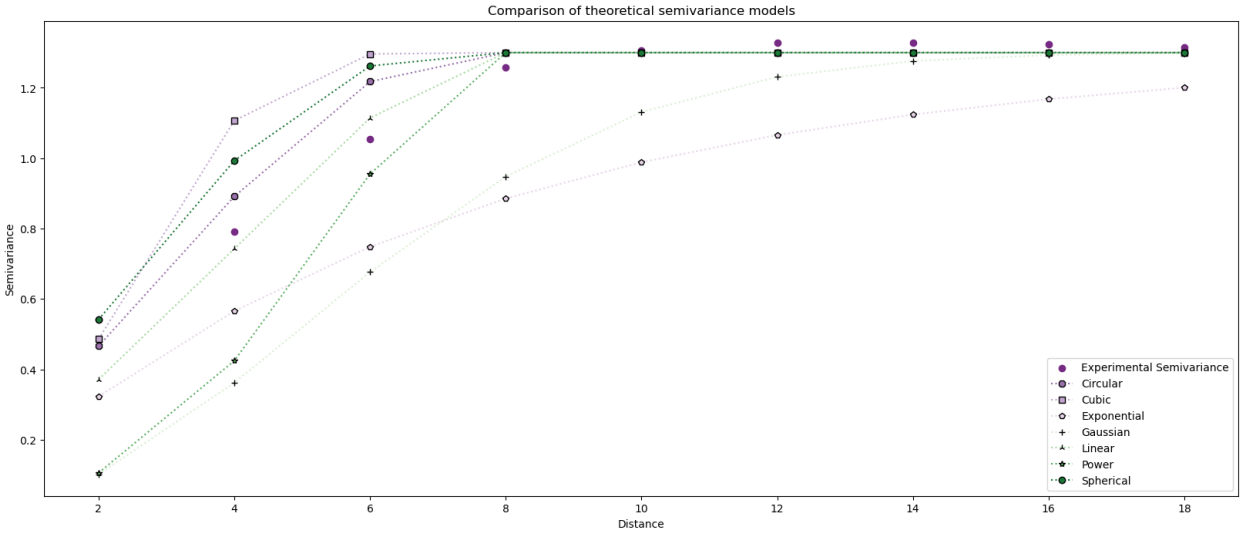
...

I hope we have chosen `circular` or `linear` models! But we should be careful with the final opinion and even more careful if we let the algorithm decide which model is the best based on the **root mean squared error metric** (RMSE). For us, the best model is always the model that has the best fit for the closest distances. A model with low RMSE may be "great" with distances from 10 to 18... which oscillate around the sill of our data and are useless for weighting (weight is constant from some distance).

4) Compare semivariogram models

The last part is a comparison of variogram models. We will still use our "own eyes" to decide which model is the best, but this time we will plot all of them in a single plot.

[28]:
```
# Plot ALL models

plt.figure(figsize=(20, 8))
plt.scatter(_lags, semivars[:, 1], color='#762a83')
plt.plot(_lags, circular, ':8', color='#9970ab', mec='black')
plt.plot(_lags, cubic, ':s', color='#c2a5cf', mec='black')
plt.plot(_lags, exponential, ':p', color='#e7d4e8', mec='black')
plt.plot(_lags, gaussian, ':+', color='#d9f0d3', mec='black')
plt.plot(_lags, linear, ':2', color='#a6dba0', mec='black')
plt.plot(_lags, power, ':*', color='#5aae61', mec='black')
plt.plot(_lags, spherical, ':o', color='#1b7837', mec='black')
plt.title('Comparison of theoretical semivariance models')
plt.legend(['Experimental Semivariance',
            'Circular',
            'Cubic',
            'Exponential',
            'Gaussian',
            'Linear',
            'Power',
            'Spherical'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



We observe that the lowest distance from a modeled value to the experimental curve at a small distance has…

Where to go from here?

- A.1.3 Spatial Dependence Index
- A.2.1 Directional Semivariogram
- A.2.2 Variogram Point Cloud
- A.2.3 Experimental Variogram and Variogram Point Cloud Classes
- B.1.1 Ordinary and Simple Kriging

Changelog

Date	Change description	Author
2023-08-18	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2022-11-05	The tutorial updated for the 0.3.5 version of the package	@SimonMolinsky
2022-08-16	The first release of tutorial	@SimonMolinsky

[ ]:

A.1.3 Spatial Dependence Index

Is dataset worth modeling?

Table of Contents:

1. What is the spatial dependcy index?
2. Why do we use spatial dependency index?
3. Example: The comparison of different Spatial Dependence Index over the same extent but different elements.
4. API links.

Introduction

In this tutorial, we will learn how to estimate **spatial dependency index**. Algorithm is based on the work:

```
[1] CAMBARDELLA, C.A.; MOORMAN, T.B.; PARKIN, T.B.; KARLEN, D.L.; NOVAK, J.M.; TURCO, R.
F.; KONOPKA, A.E. Field-scale variability of soil properties in central Iowa soils. Soil
Science Society of America Journal, v. 58, n. 5, p. 1501-1511, 1994.
```

1. What is the spatial dependency index?

The spatial dependency index (SDI) measures the strength of a spatial process we are modeling. SDI is normalized to the interval between 0 and 1. Therefore, we can transform it into percentages and assign an order of spatial dependency from weak to strong.

The SDI is a ratio of the nugget to the total variance (sill) of a model:

$$SDI = \frac{nugget}{sill} * 100$$

Whenever we fit a theoretical variogram with the `pyinterpolate` package, SDI is calculated, and we will take advantage of it in the examples. Two values represent SDI:

- **numeric**, a ratio of nugget and sill in percent,
- **categorical**, a description of a spatial dependency strength.

There are four levels of spatial dependency.

Lower Limit (included)	Upper Limit (excluded)	Strength
0	25	strong
25	75	moderate
75	95	weak
95	*inf*	no spatial dependence

The lower the ratio, the most substantial spatial dependence. If the ratio is greater than 75 percent, we should be cautious with spatial modeling because spatial similarities may not explain the process.

2. Why do we use the spatial dependency index?

In a world where Tobler's Law can be applied to every spatial phenomenon, we might always use kriging without consideration. We know that spatial dependence exists, and close neighbors are always similar.

This world is not our world! Not every process follows Tobler's Law. We can find examples sampled over the same area and scale, but their spatial dependence indexes differ. In [1] (SDI in `pyinterpolate` is based on this publication), multiple chemical compounds are sampled from the same field. Every compound has its spatial distribution and variogram. Some soil parameters are not spatially dependent (for example, *Mg* or *Ca* that are randomly distributed).

The spatial dependence index level marks the next decision on what to do with the data.

- Strong: just krige it!
- Moderate: there might be some other thing that explains process variation.
- Weak: the other non-spatial process has more influence on data than spatial similarities.
- No spatial dependence: the process is random, or spatial dependencies cannot explain variance.

Note: Be careful because the last two points are red flags, BUT sometimes processes with a low variogram variability at one scale may be explained with spatial relations at a changed scale. A practical example is a comparison of rental apartment prices per night: if you look at the scale of hundreds of kilometers, the spatial dependence may be very weak or none. On the other hand, the spatial similarity between prices of apartments close to each other (up to 10 kilometers, 6 miles) tends to show a *classic variogram curve*. The reason is simple: most managers and algorithms use information about the pricing of the closest neighbors, and neighborhood prices are affected by the same external objects or events.

We should look into the spatial dependence index to ensure our path leads us toward meaningful results.

Example: Spatial Dependence over the same study extent but for different elements

We will compare the spatial dependence index of four elements: cadmium, copper, lead, and zinc. We use the *meuse* dataset. The dataset comes from:

Pebesma, Edzer. (2009). The meuse data set: a tutorial for the gstat R package -> [link to the publication](https://cran.r-project.org/web/packages/gstat/vignettes/gstat.pdf)

[1]:
```
import numpy as np
import pandas as pd
import pyinterpolate as ptp
```

[2]:
```
MEUSE_FILE = 'samples/point_data/csv/meuse/meuse.csv'

# Variogram parameters
STEP_SIZE = 100
MAX_RANGE = 1600
ALLOWED_MODELS = 'safe'


# Elements
ELEMENTS = ['cadmium', 'copper', 'zinc', 'lead']
COLS = ['x', 'y']
COLS.extend(ELEMENTS)
```

[3]:
```
df = pd.read_csv(MEUSE_FILE, usecols=COLS)
df.head()
```

[3]:

	x	y	cadmium	copper	lead	zinc
0	181072	333611	11.7	85	299	1022
1	181025	333558	8.6	81	277	1141
2	181165	333537	6.5	68	199	640
3	181298	333484	2.6	81	116	257
4	181307	333330	2.8	48	117	269

Data is skewed for every element in the table. Before we start variogram modeling, we must transform it into a distribution close to normality. We will use a logarithmic transform, pass data into an experimental variogram, and then create theoretical models with the `autofit()` method. We will use *grid search* to find the best possible nugget value.

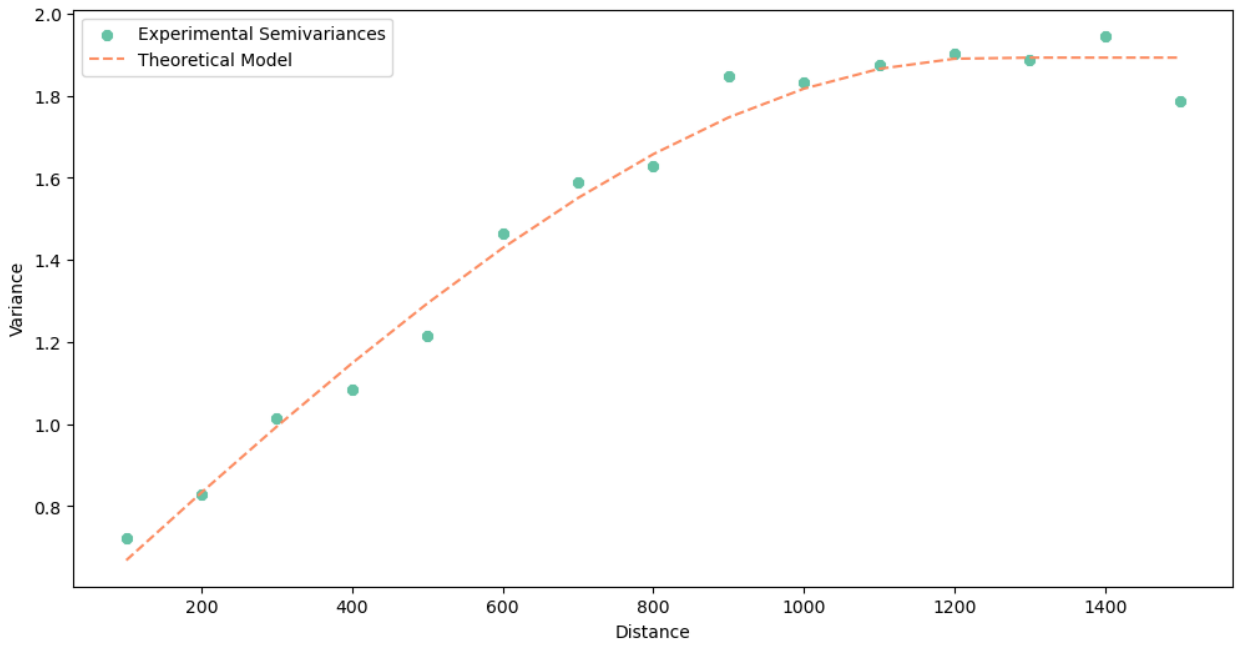
[4]:
```
for element in ELEMENTS:
    # Prepare data
    ds = df[['x', 'y', element]].copy()
    ds[element] = np.log(ds[element])
    arr = ds.values

    # Get experimental variogram
    exp_var = ptp.build_experimental_variogram(arr, step_size=STEP_SIZE, max_range=MAX_
↪RANGE)

    # Find optimal theoretical model
    optimal_model = ptp.TheoreticalVariogram()
    optimal_model.autofit(experimental_variogram=exp_var, model_name=ALLOWED_MODELS, max_
↪nugget=0.8)
    print('Optimal model for element {} has {:.2f}% of nugget to sill ratio. Spatial␣
```

(continues on next page)

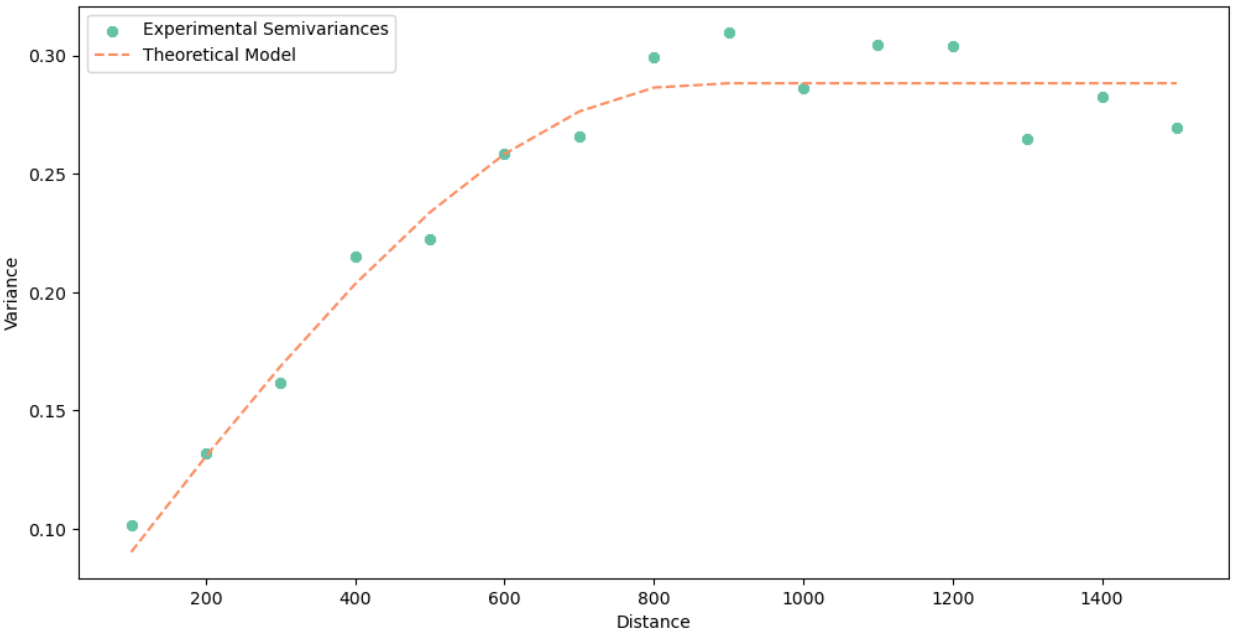
(continued from previous page)

```
↪Dependence is {}.'.format(
        element, optimal_model.spatial_dependency_ratio, optimal_model.spatial_
↪dependency_strength
    ))
    optimal_model.plot()

    print('\n---\n')
```

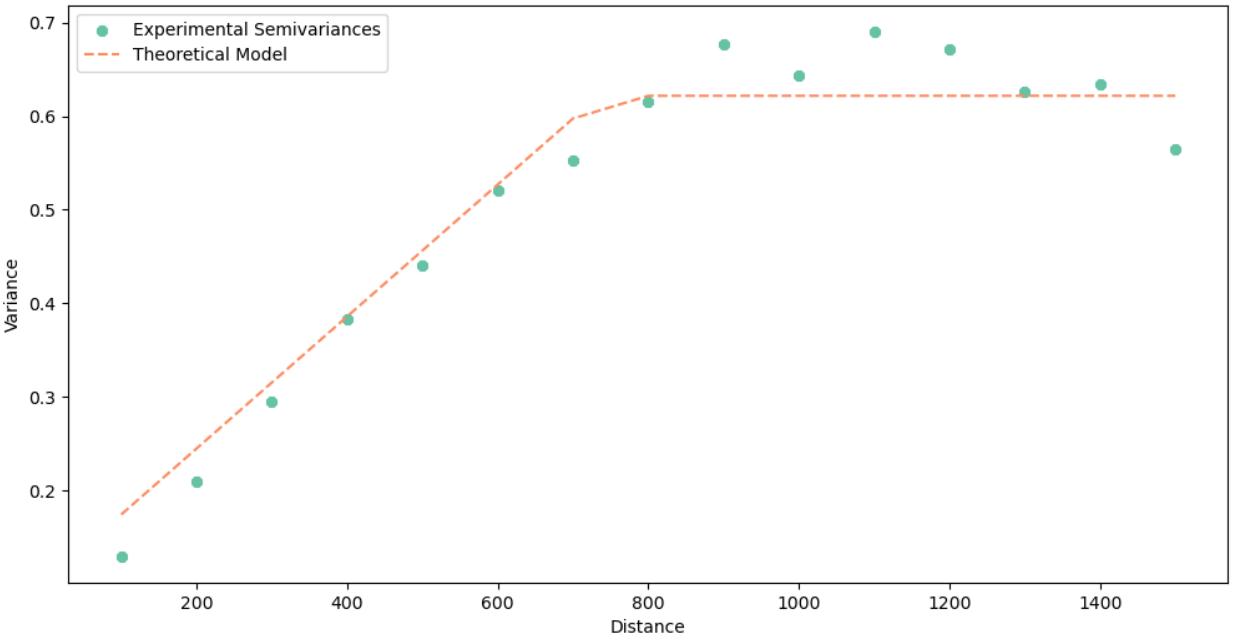
```
Optimal model for element cadmium has 36.02% of nugget to sill ratio. Spatial Dependence␣
↪is moderate.
```



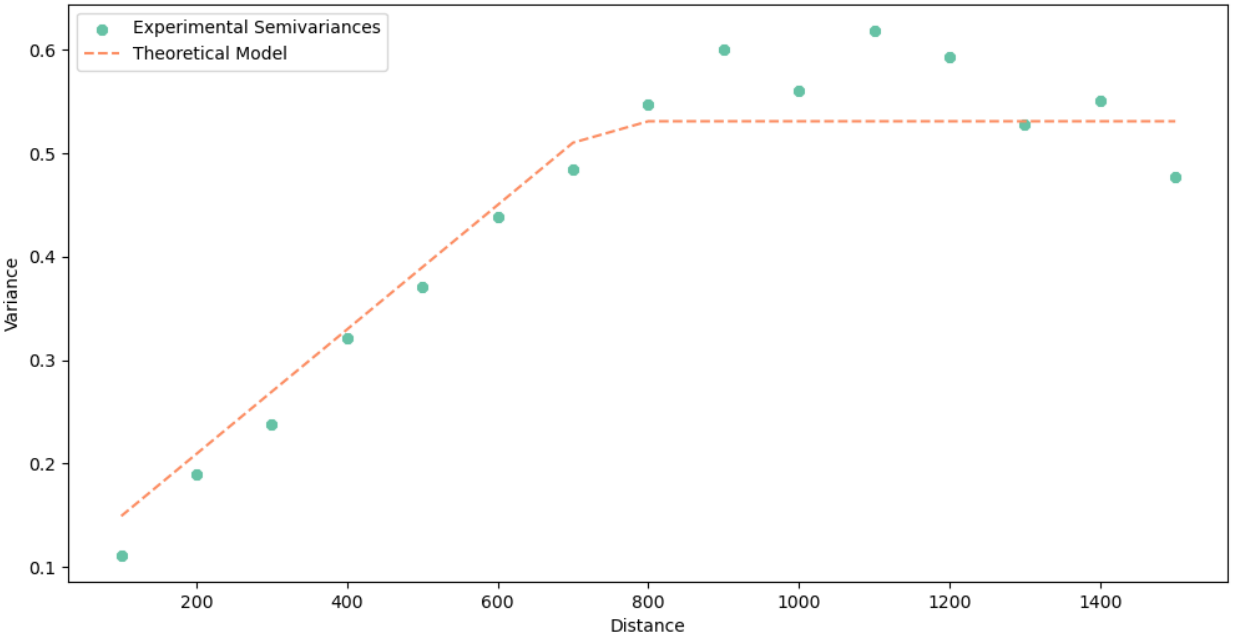

```
---
Optimal model for element copper has 20.31% of nugget to sill ratio. Spatial Dependence␣
↪is strong.
```



Optimal model for element zinc has 20.08% of nugget to sill ratio. Spatial Dependence is strong.



Optimal model for element lead has 20.22% of nugget to sill ratio. Spatial Dependence is strong.



The strength of spatial dependence differs mostly between *Cadmium* and other elements.

API

The spatial dependence index may be calculated directly with the `calculate_spatial_dependence_index()` function that takes two parameters: `nugget` and `sill`. It returns `Tuple` with spatial dependence ratio and spatial dependence strength.

Another way is to calculate `TheoreticalVariogram` with a grid search option for nugget (just leave the `nugget` parameter as `None` and the algorithm will find the optimal nugget value).

Where to go from here?

- A.2.1 Directional Semivariogram
- A.2.2 Variogram Point Cloud
- A.2.3 Experimental Variogram and Variogram Point Cloud Classes
- B.1.1 Ordinary and Simple Kriging

Changelog

Date	Change description	Author
2023-08-19	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@SimonMolinsky
2023-04-08	The first version of the tutorial	@SimonMolinsky

[ ]:

A.2.1 Directional semivariograms

Table of Contents:

1. Read point data,
2. Create directional and isotropic semivariograms,
3. Compare semivariograms,
4. Compare triangular vs elliptical neighbors selection methods.

Introduction

In this tutorial, we will learn about directional semivariograms, how to set the angle of `direction`, and the `tolerance` parameter. We compare two neighbor selection methods: **triangular**, which is fast, and **elliptical**, which is accurate.

A directional process

Not every spatial process may be described by isotropic semivariograms. Sometimes we see a specific trend in one direction (N-S, W-E, or NE-SW, NW-SE). A well-performed analysis includes directions. That's why we will learn to write code in `pyinterpolate` that retrieves the directional semivariogram.

We use the air pollution data.

> **Note**: if you are a machine learning enthusiast, it will be easier to understand **direction** as an additional feature to model semivariance.

[3]:
```
import geopandas as gpd
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt

# Experimental variogram
from pyinterpolate import build_experimental_variogram
```

1) Read and show point data

[4]:
```
DATASET = 'PM2.5'
```

[5]:
```
ds = pd.read_csv('samples/directional_variogram_PM25_20220922.csv', index_col='station_id
↪')
ds.head()
```

[5]:

station_id	y	x	PM2.5
659	50.529892	22.112467	7.12765
736	54.380279	18.620274	4.66432
861	54.167847	19.410942	3.00739
266	51.259431	22.569133	7.10000
355	51.856692	19.421231	2.00000

[6]:
```
def df2gdf(df, lon_col='x', lat_col='y', epsg=4326, crs=None):
    """
    Function transforms DataFrame into GeoDataFrame.

    Parameters
    ----------
    df : pandas DataFrame

    lon_col : str, default = 'x'
        Longitude column name.

    lat_col : str, default = 'y'
        Latitude column name.

    epsg : Union[int, str], default = 4326
        EPSG number of projection.

    crs : str, default = None
        Coordinate Reference System of data.

    Returns
    -------
    gdf : GeoPandas GeoDataFrame
        GeoDataFrame with set geometry column ('geometry'), CRS, and all columns from the␣
↪passed DataFrame.

    """

    geometry = gpd.points_from_xy(x=df[lon_col], y=df[lat_col])
    geometry.name = 'geometry'
    gdf = gpd.GeoDataFrame(df, geometry=geometry)

    if crs is None:
        gdf.set_crs(epsg=epsg, inplace=True)
    else:
        gdf.set_crs(crs=crs, inplace=True)
```

(continues on next page)

(continued from previous page)

```
    return gdf
```

[7]:
```
gds = df2gdf(ds)
gds.head()
```

[7]:
```
                    y          x    PM2.5                    geometry
station_id
659         50.529892  22.112467  7.12765  POINT (22.11247 50.52989)
736         54.380279  18.620274  4.66432  POINT (18.62027 54.38028)
861         54.167847  19.410942  3.00739  POINT (19.41094 54.16785)
266         51.259431  22.569133  7.10000  POINT (22.56913 51.25943)
355         51.856692  19.421231  2.00000  POINT (19.42123 51.85669)
```

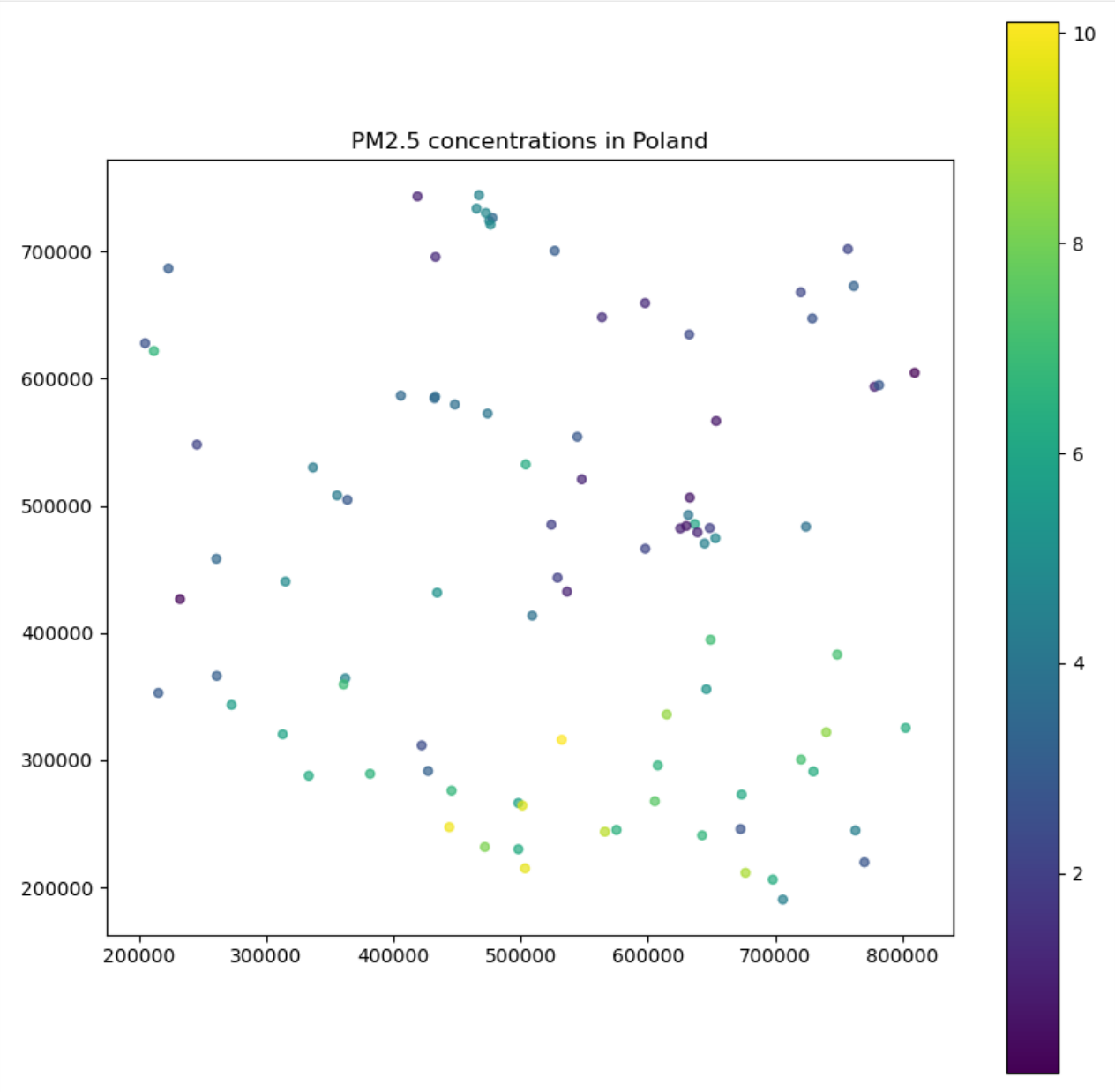
[8]:
```
gds.isna().any()
```

[8]:
```
y           False
x           False
PM2.5        True
geometry    False
dtype: bool
```

[9]:
```
gds = gds.dropna(axis=0)
```

[10]:
```
gds = gds.to_crs(epsg=2180)
gds.plot(figsize=(10, 10), column=DATASET, legend=True, markersize=20, alpha=0.7)
plt.title('PM2.5 concentrations in Poland')
```

[10]:
```
Text(0.5, 1.0, 'PM2.5 concentrations in Poland')
```



2) Create the experimental semivariogram

If we recall the tutorial `A.1.1 Semivariogram Estimation`, there are three main parameters to set for the `build_experimental_variogram()` function:

1. `input_array`: `numpy array` with coordinates and observed values, for example: `[[0, 0, 10], [0, 1, 20]]`,
2. `step_size`: we must divide our analysis area into discrete **lags**. **Lags** are intervals (usually circular) within which we check if the point has a neighbor. For example, if we look into the lag `500`, then we are going to compare one point with other points in a distance `(0, 1000]` from this point,
3. `max_range`: This parameter represents the possible **maximum range of spatial dependency**. This parameter should be at most half of the extent.

But that's not everything! We didn't use three other parameters:

4. `direction`: it is a `float` in the range [0, 360]. We set the direction of the semivariogram:

- 0 or 180: is WE direction,
- 90 or 270 is NS direction,
- 45 or 225 is NE-SW direction,
- 135 or 315 is NW-SE direction.

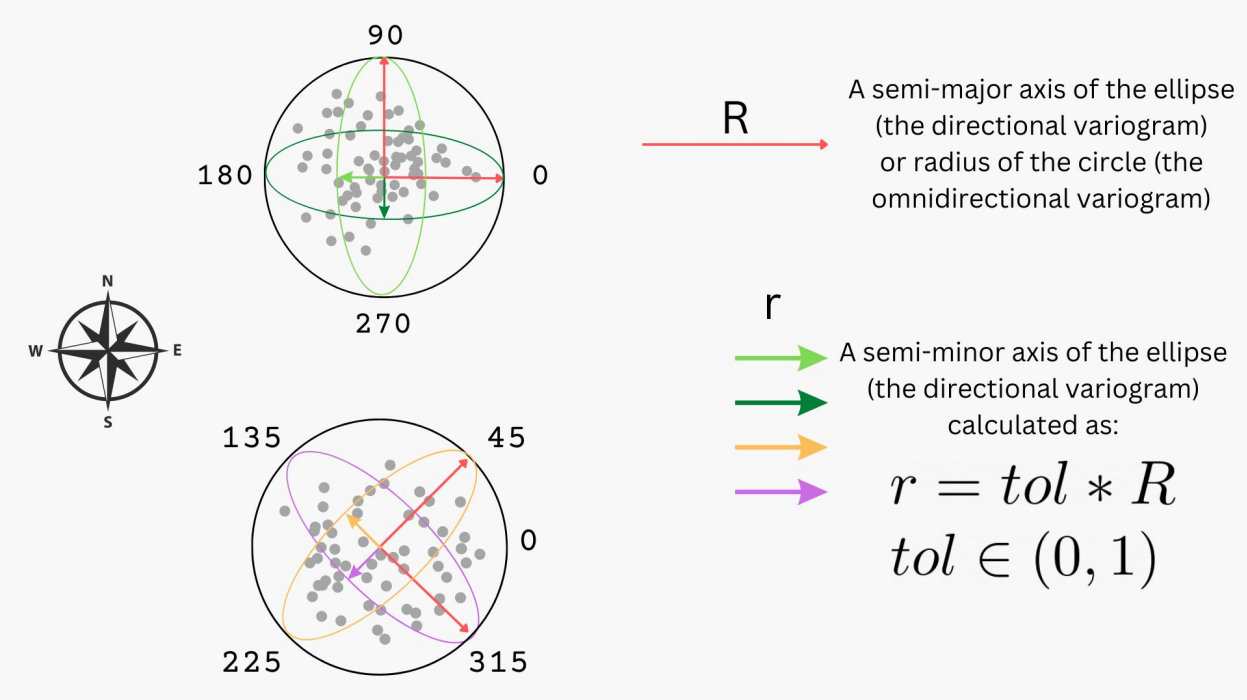
5. `tolerance`: it is a `float` in the range [0, 1]. If we leave `tolerance` with default **1**, we will always get an isotropic semivariogram. Another edge case is if we set `tolerance` to **0**, then points must be placed on a single line with the beginning in the origin of the coordinate system and the angle given by the y-axis and direction parameter. If tolerance is > 0 and < 1, the bin is selected as an elliptical area with a major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size is `(tolerance * step_size)`,
- The minor axis size is `((1 - tolerance) * step_size)`,
- The baseline point is at the center of the ellipse.

6. `method`: it is `str` with possible values:

- `t` or `triangle` for triangular neighbors selection which is fast and recommended for a big dataset,
- `e` or `ellipse` for elliptical neighbors selection which is accurate but slow, recommended for small-size datasets.

The best idea is to visualize it on a cartesian plane:



- The top plane shows a unit circle that is the omnidirectional variogram (black circle). Within it, we see two ellipses: one is bright green, and another is dark green.
- The bottom plane also shows a unit circle and two ellipses: the brighter yellow, and darker purple.
- The long arrows within both circles are radiuses of the omnidirectional variogram or **the semi-major axis** of a directional ellipse. The `step_size` parameter controls its length. The short arrows are present only in the

directional variograms and are **the semi-minor axes**. The `tolerance` parameter controls their length, and it is always:

- a fraction of `step_size`,
- **1**, in this case, a semivariogram is omnidirectional,
- a value very close to 0 (but not 0) - an ellipse falls into the line.
- The good idea is to set `tolerance` to 0.5 and gradually make it smaller or larger (depending on the spatial features).

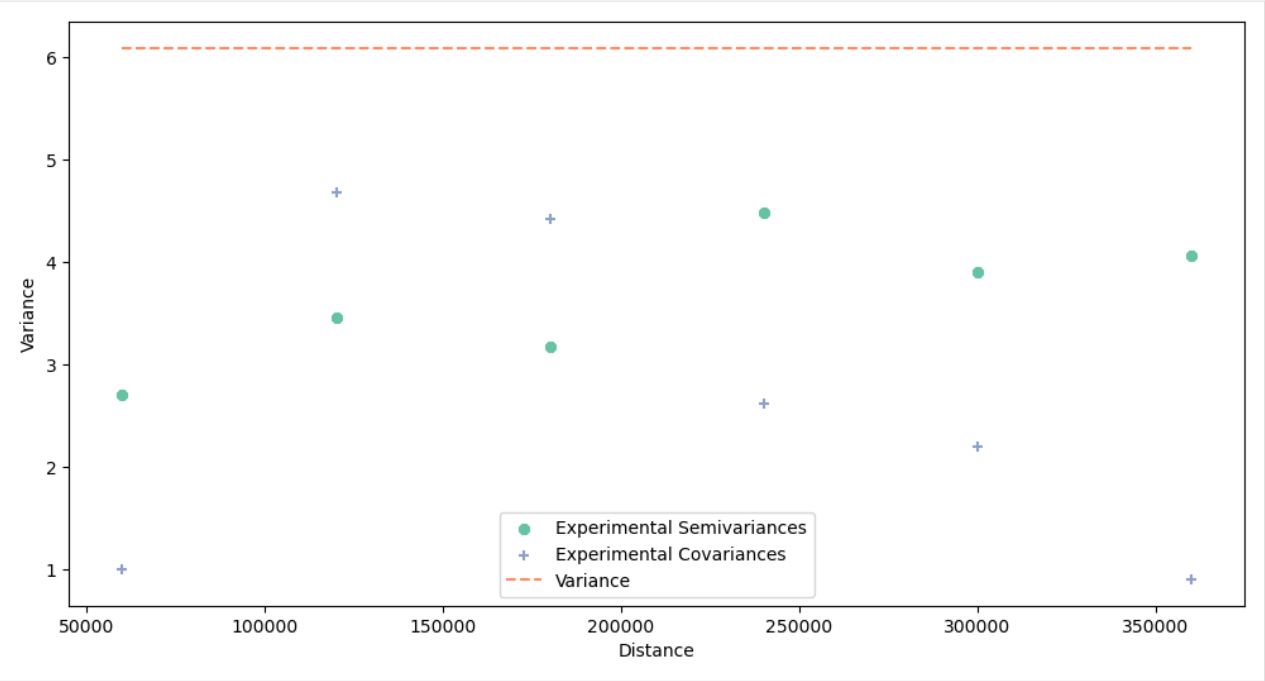
We can start the analysis by understanding how our parameters affect the semivariogram bins range. We will set `tolerance` to 0.2 in each case to better visualize the effects of a leading direction.

Case 1: W-E Direction

[11]:
```
BIN_RADIUS = 60000  # meters
MAX_RANGE = 400000  # meters
TOLERANCE = 0.2
```

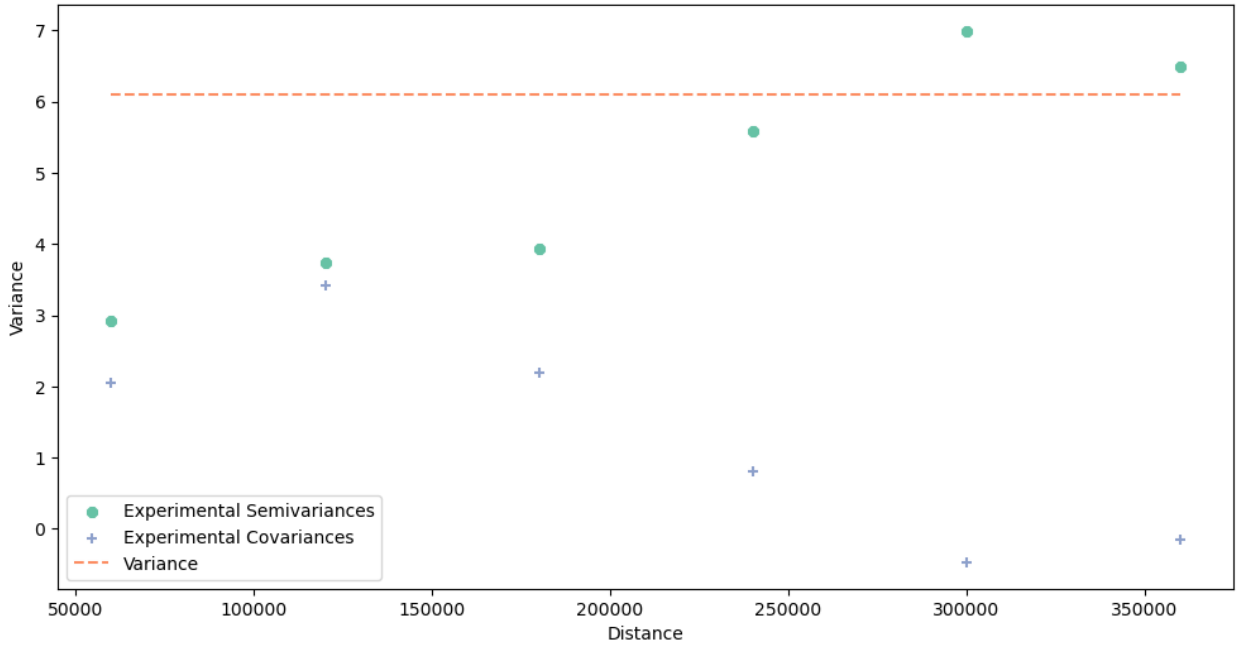
[12]:
```
inp_arr = np.array(list(zip(gds['geometry'].x,
                            gds['geometry'].y,
                            gds[DATASET])))
```

[13]:
```
we_variogram = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=0,
    tolerance=TOLERANCE,
    method='e'
)

we_variogram.plot(plot_semivariance=True)
```



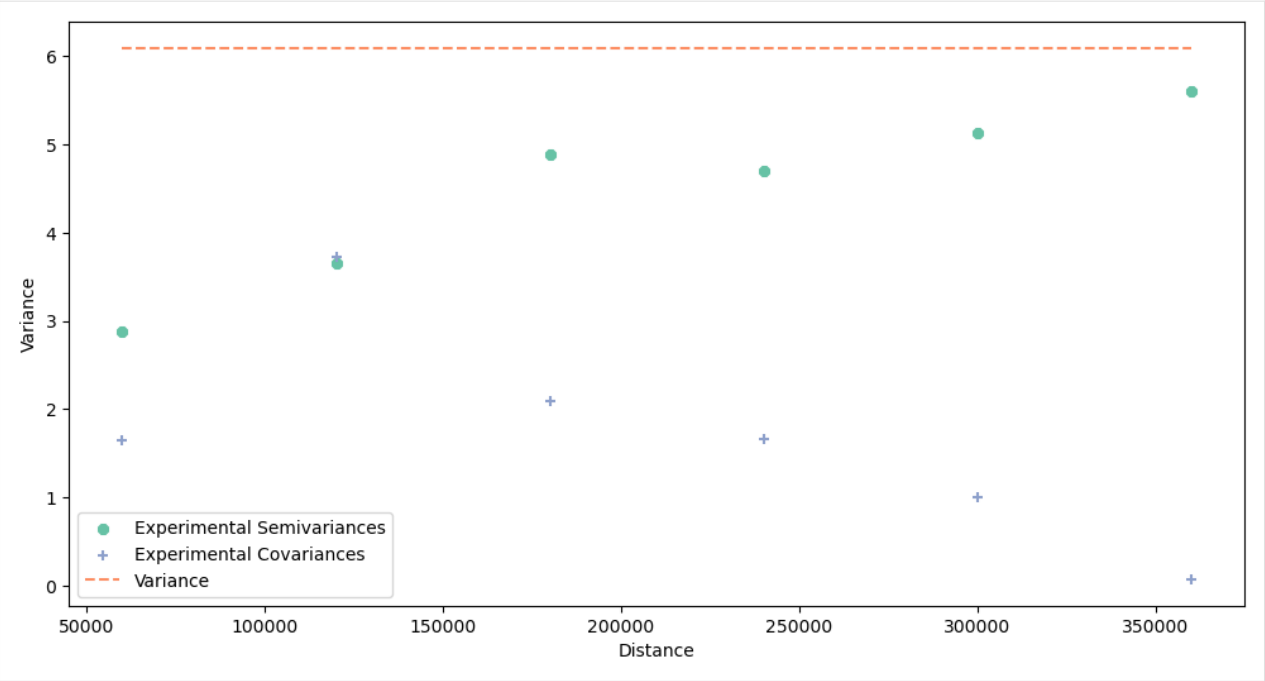
Case 2: N-S Direction

[14]:
```
ns_variogram = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=90,
    tolerance=TOLERANCE,
    method='e'
)

ns_variogram.plot(plot_semivariance=True)
```



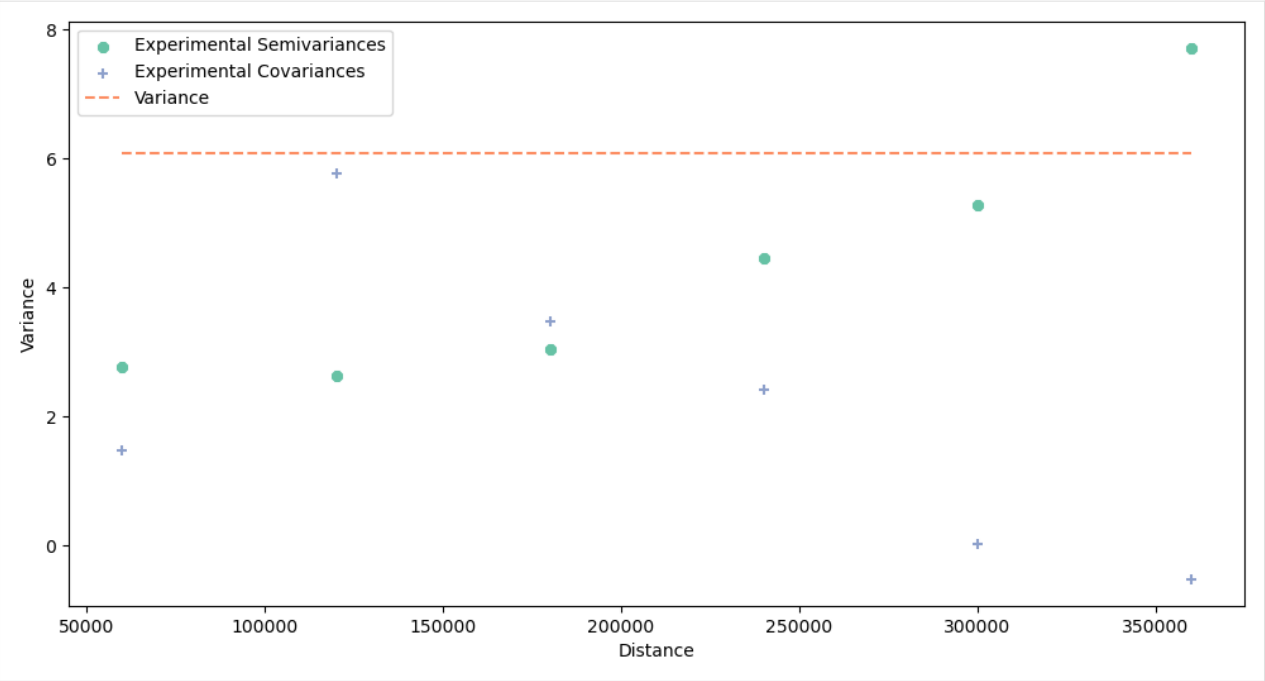
Case 3: NW-SE Direction

[15]:
```
nw_se_variogram = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=135,
    tolerance=TOLERANCE,
    method='e'
)

nw_se_variogram.plot(plot_semivariance=True)
```



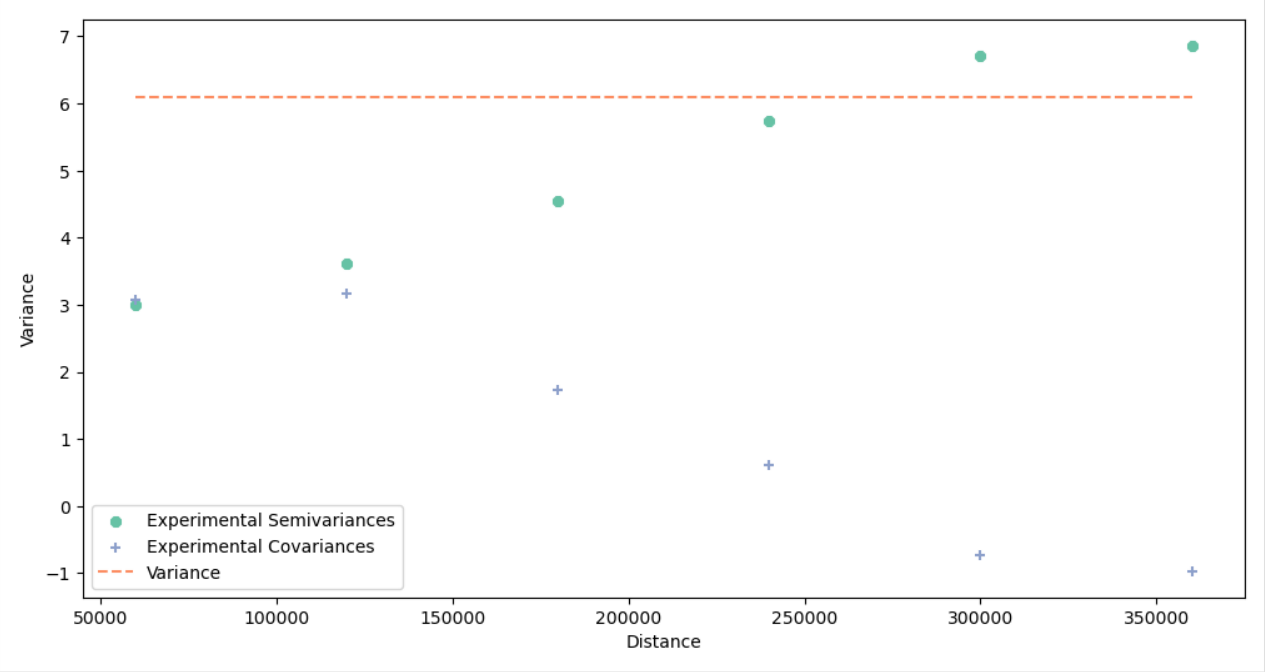
Case 4: NE-SW Direction

[16]:
```
ne_sw_variogram = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=45,
    tolerance=TOLERANCE,
    method='e'
)

ne_sw_variogram.plot(plot_semivariance=True)
```



Case 5: Isotropic Variogram

[17]:
```
iso_variogram = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE
)

iso_variogram.plot()
```



3) Compare semivariograms

We have created a set of variograms. What did we observe?

- The **NE-SW** variogram is very weak at describing a short-range variation (compare it to the map of air pollution from the beginning of the tutorial. Points in this direction are relatively similar).
- The **N-S** variogram works well for a short range.
- The **W-E** variogram catches too much variability, and lags must be longer for this direction.
- The **NW-SE** variogram looks good and shows approximately linear variability change in a distance function. It has the smallest variance from all variograms.

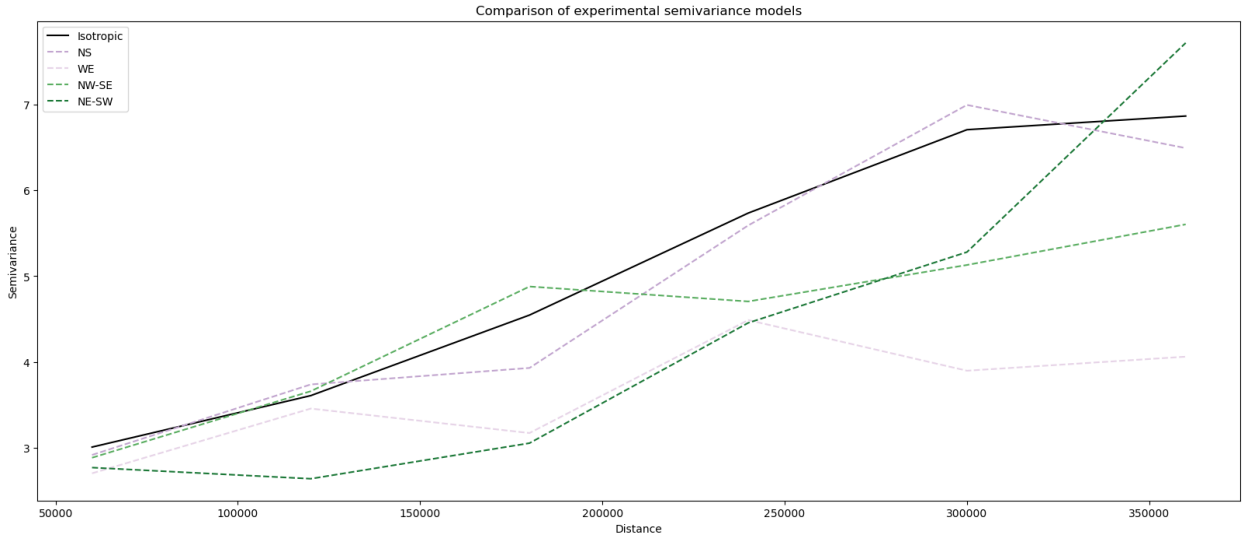
We can visualize and compare all variograms simultaneously, and we are sure that the y-axis is the same for every plot.

[18]:
```
_lags = iso_variogram.lags
_ns = ns_variogram.experimental_semivariances
_we = we_variogram.experimental_semivariances
_nw_se = nw_se_variogram.experimental_semivariances
_ne_sw = ne_sw_variogram.experimental_semivariances
_iso = iso_variogram.experimental_semivariances

plt.figure(figsize=(20, 8))
plt.plot(_lags, _iso, color='black')
plt.plot(_lags, _ns, '--', color='#c2a5cf')
plt.plot(_lags, _we, '--', color='#e7d4e8')
plt.plot(_lags, _nw_se, '--', color='#5aae61')
plt.plot(_lags, _ne_sw, '--', color='#1b7837')
plt.title('Comparison of experimental semivariance models')
plt.legend(['Isotropic',
            'NS',
```

(continues on next page)

(continued from previous page)

```
            'WE',
            'NW-SE',
            'NE-SW'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



What do you think about this comparison? Do you agree that the **NW-SE** variogram best fits the data?

4) Bonus: Compare calculation times and results

Let's compare calculation times and results from two methods of neighbors interpolation: `t` (triangular) and `e` (elliptical). We will follow data in the **NW-SE** direction.

[19]:
```
from datetime import datetime
```

[20]:
```
t0e = datetime.now()

# Here elliptical

nw_se_variogram_elliptical = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=135,
    tolerance=TOLERANCE,
    method='e'
)

tfin_e = (datetime.now() - t0e).total_seconds()
```

[21]:
```
t0t = datetime.now()
```

(continues on next page)

(continued from previous page)

```
# Here triangular

nw_se_variogram_triangular = build_experimental_variogram(
    input_array=inp_arr,
    step_size=BIN_RADIUS,
    max_range=MAX_RANGE,
    direction=135,
    tolerance=TOLERANCE,
    method='t'
)

tfin_t = (datetime.now() - t0t).total_seconds()
```

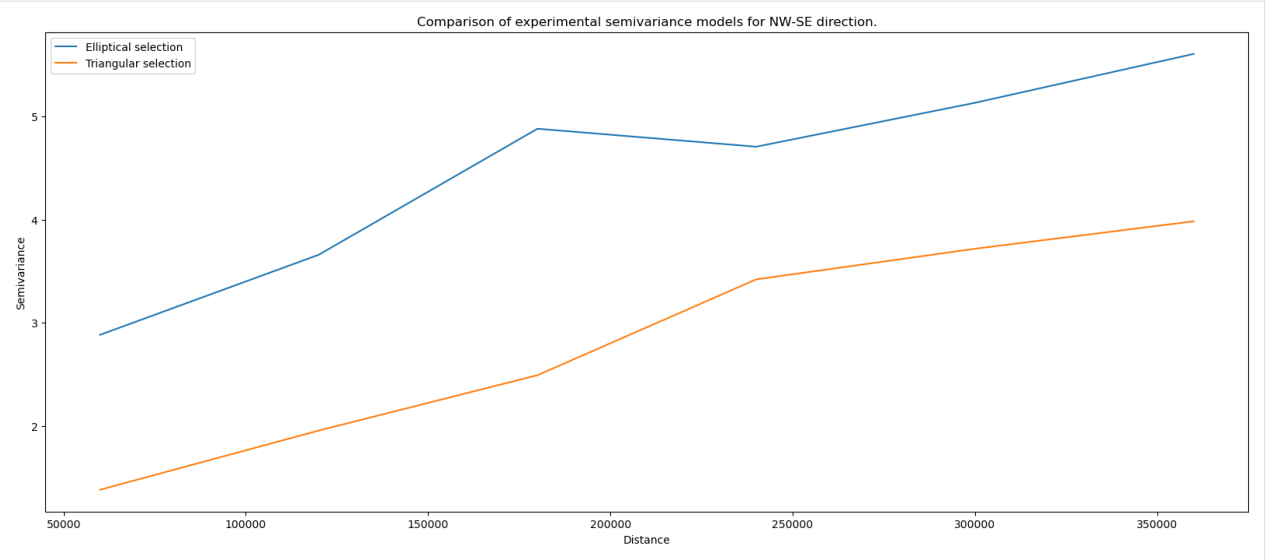
[22]:
```
msg = f'The triangular method is {tfin_e / tfin_t} times faster than the elliptical␣
↪selection.'

print(msg)
```

```
The triangular method is 2.7558084695886036 times faster than the elliptical selection.
```

For this scenario, the triangular method is 1.5x faster than the elliptical selection. However, this difference will be more significant with more points and point pairs, and triangular selection is 5x faster for big datasets.

Are variograms different? Let's compare!

[23]:
```
_lags = iso_variogram.lags
_nw_se_ell = nw_se_variogram_elliptical.experimental_semivariances
_nw_se_tri = nw_se_variogram_triangular.experimental_semivariances

plt.figure(figsize=(20, 8))
plt.plot(_lags, _nw_se_ell)
plt.plot(_lags, _nw_se_tri)
plt.title('Comparison of experimental semivariance models for NW-SE direction.')
plt.legend(['Elliptical selection', 'Triangular selection'])
plt.xlabel('Distance')
plt.ylabel('Semivariance')
plt.show()
```



The general shape is the same, but the variance of a semivariogram from a triangular selection is lower; it is harder to distinguish the covariance between neighboring points and could be dangerously close to the nugget effect. You should consider this (especially) in automatic calculations to avoid strange results or models that are not better than a simple IDW technique.

But semivariogram for the elliptical selection can have different parameters than for the triangular selection. If you choose a different step size and tolerance, the semivariogram's shape may change and could better describe dissimilarities.

Where to go from here?

- A.2.2 Variogram Point Cloud
- A.2.3 Experimental Variogram and Variogram Point Cloud Classes
- B.1.1 Ordinary and Simple Kriging
- B.3.1 Directional Ordinary Kriging

Changelog

Date	Change description	Author
2023-08-21	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-04-15	Update: 0.4.1	@Simon-Molinsky
2023-03-04	Removed import for the download air quality dataset	@Simon-Molinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@Simon-Molinsky
2022-10-31	Bug update: wrong direction of angle from origin, it was clockwise but now it is counter-clockwise	@Simon-Molinsky
2022-10-21	Bug update: wrong name of the N-S variogram	@Simon-Molinsky
2022-10-17	Bug update: directions are calculated in a valid way. New feature: added comparison between elliptical and triangular neighbors selection	@Simon-Molinsky
2022-09-22	The first version of tutorial	@Simon-Molinsky

[ ]:

A.2.2 Variogram Points Cloud

Table of Contents

1. Read point data,
2. Create variogram points cloud,
3. Detect and remove outliers.
4. Calculate the experimental semivariance from the points cloud.

Introduction

We will learn how to read and prepare data for semivariogram modeling, manually find the best step size between lags, and detect outliers in our data.

Variogram Point Cloud analysis is an additional, essential data preparation step that may save you a lot of headaches with more sophisticated analysis. You should learn about Variogram Point Cloud analysis before you move on to the semivariogram estimation and semivariogram fitting operations.

We use:

- for points 1 and 2: DEM data stored in a file `samples/point_data/txt/pl_dem_epsg2180.txt`,
- for points 3 and 4: Breast cancer rates data is stored in the shapefile in folder `samples/regularization/cancer_data.gpkg`.

Import packages

[1]:
```
import numpy as np

from pyinterpolate import Blocks, read_txt, calc_point_to_point_distance, VariogramCloud
from pyinterpolate.variogram.empirical.experimental_variogram import calculate_
↪semivariance
```

1) Read point data

[2]:
```
dem = read_txt('samples/point_data/txt/pl_dem_epsg2180.txt')
```

[4]:
```
sample_size = int(0.05 * len(dem))
indices = np.random.choice(len(dem), sample_size, replace=False)
dem = dem[indices]

# Look into a first few lines of data

dem[:3, :]
```

[4]:
```
array([[2.51837397e+05, 5.45249474e+05, 2.11946869e+01],
       [2.41974281e+05, 5.38958171e+05, 1.64789104e+01],
       [2.45063630e+05, 5.43685759e+05, 1.90657024e+01]])
```

2) Set proper lag size with variogram cloud histogram

We will generate the Variogram Point Cloud. We calculate it for 16 lags and test different cloud variogram visualization methods.

[5]:
```
# Create analysis parameters

# Check max distance between points
distances = calc_point_to_point_distance(dem[:, :-1])
maximum_range = np.max(distances) / 2

number_of_lags = 16
step_size = maximum_range / number_of_lags

vc = VariogramCloud(input_array=dem, step_size=step_size, max_range=maximum_range)
```

Check how many points are grouped for each lag:

[6]:
```
for idx, _lag in enumerate(vc.lags):
    print(f'Lag {_lag} has {vc.points_per_lag[idx]} point pairs.')
```

```
Lag 809.6271354731758 has 2 point pairs.
Lag 1619.2542709463517 has 4 point pairs.
Lag 2428.8814064195276 has 4 point pairs.
Lag 3238.5085418927033 has 12 point pairs.
Lag 4048.135677365879 has 8 point pairs.
```

(continues on next page)

(continued from previous page)

```
Lag 4857.762812839055 has 16 point pairs.
Lag 5667.389948312231 has 14 point pairs.
Lag 6477.017083785407 has 16 point pairs.
Lag 7286.644219258583 has 16 point pairs.
Lag 8096.271354731759 has 14 point pairs.
Lag 8905.898490204934 has 10 point pairs.
Lag 9715.52562567811 has 14 point pairs.
Lag 10525.152761151287 has 16 point pairs.
Lag 11334.779896624463 has 8 point pairs.
Lag 12144.407032097637 has 10 point pairs.
```

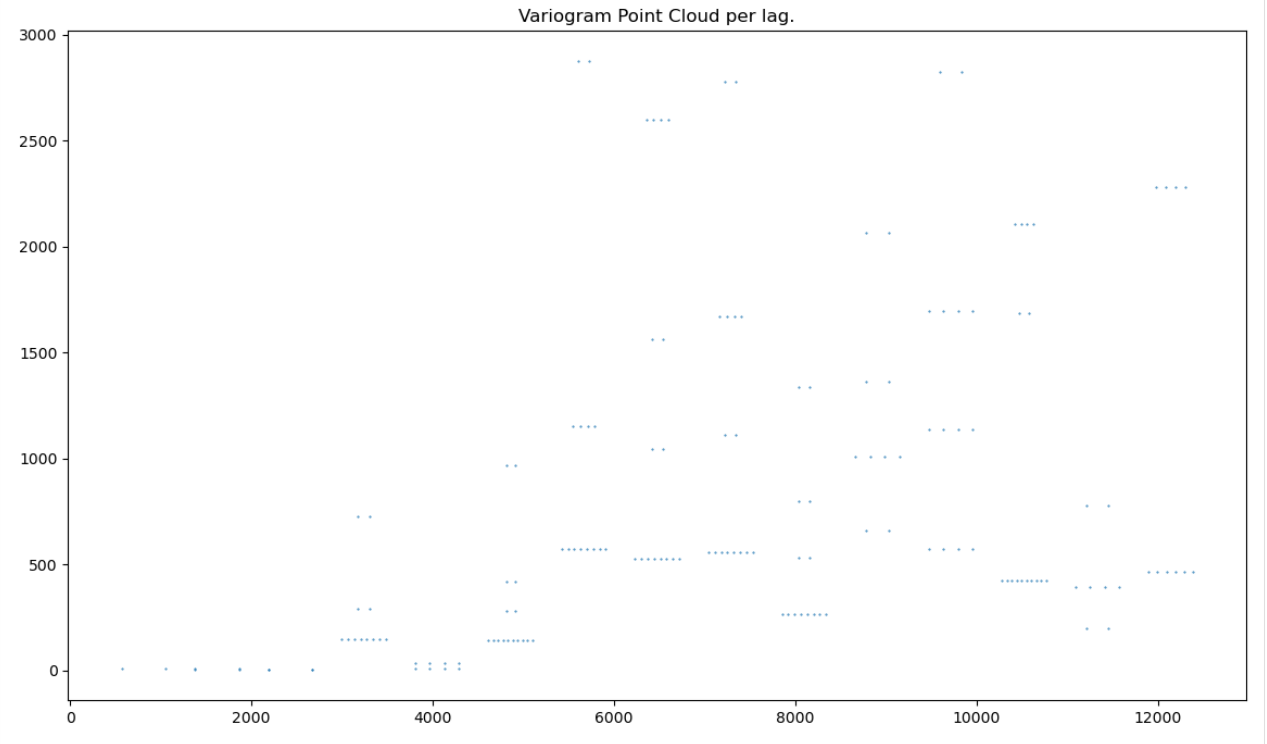
As we see, there are a lot of points per lag! Data is messy, but to be sure, we must check its distribution. What are those **points per lag**? They are a set of semivariances between all point pairs within a specific lag. This information helps us to understand if there are any undersampled ranges. Then, we can change `step_size` accordingly (to avoid a situation where one lag has the order of magnitude less/more point pairs than the other).

We know the density of the points per lag. Now we can analyze semivariances distribution per lag. The `VariogramCloud` class has three plot types we use for analysis:

1. *Scatter plot*. It shows the general dispersion of semivariances.
2. *Box plot* - an excellent tool for outliers detection. It shows dispersion and quartiles of semivariances per lag. We can check distribution differences and their deviation from normality or skewness.
3. *Violin plot*. It is a box plot on steroids. We can read all information from the box plot and see kernel density plots.

Let's plot them all. The first is a scatter plot.

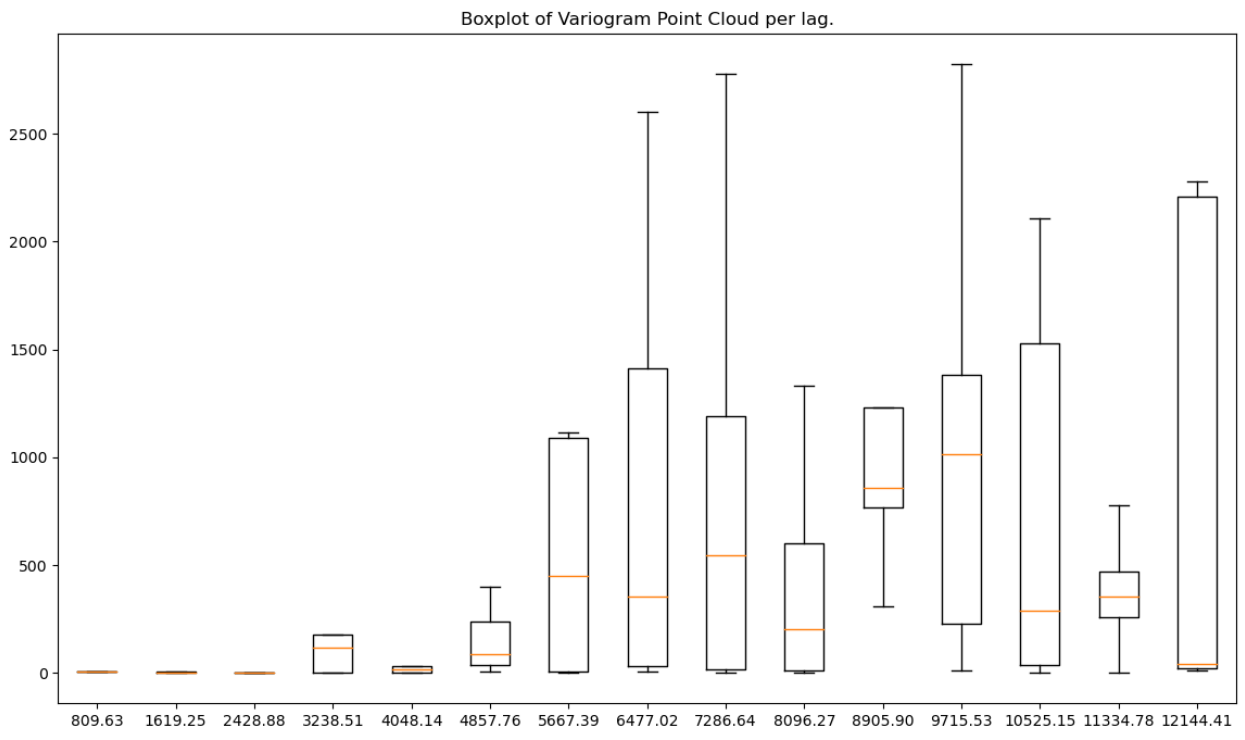
[8]:
```
vc.plot('scatter');
```



The output per lag is dense, and distributions are hard to read. But, we can follow a general trend and see maxima on

the plot. We can make it better if we use a box plot instead:

[9]:
```
vc.plot('box')
```



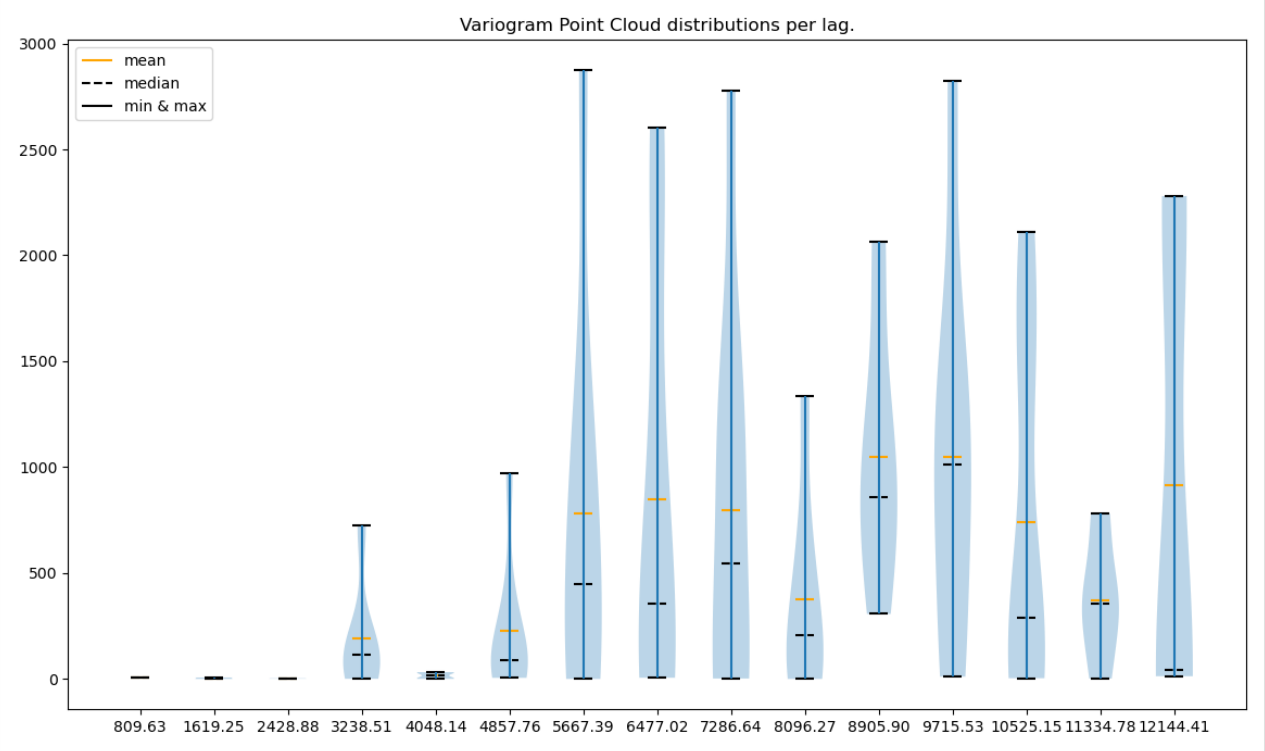
[9]:
```
True
```

The orange line in the middle of each box represents the median (50%) value per lag. The trend is more visible if we look into it. What else can be seen here?

1. Dispersion of values is greater with a distance, but at the same time, median and the **3rd quartile** of semivariance values are rising (**3rd quartile** is plotted as a horizontal line on a top of a box.
2. The maximum value rises with a distance (it is a horizontal line on a top of a whisker).
3. Data is skewed towards lower values of semivariance. Why? Because the median is closer to the bottom of a box than the center or the top.
4. The **1st quartile** (25% of the lowest values) is higher for distant lags.

We could stop our analysis here, but there is a better option to analyze data distribution than a box plot. We can use a violin plot:

[10]:
```
vc.plot('violin')
```



[10]:
```
True
```

The violin plot supports our earlier assumptions, but we can learn more from it. For distant lags, a distribution starts to be multimodal. We see one mode around very low semivariance values and another close to the 1st quartile. Multimodality may affect our outcomes, and it may tell us that there is more than one level of spatial dependency.

There is still the elephant in the room. Our data is pulled up with outliers. Do you see long whiskers on the top of every distribution? Let them be, but the better idea is to clean it before we start Kriging. In the next paragraph, we will detect and remove outliers from a block dataset.

3) Detect and remove outliers

With the idea of how the Variogram Point Cloud works, we can detect and "remove" outliers from our dataset. In this part of the tutorial, we use another dataset. It represents the breast cancer rates in counties of the Northeastern part of the U.S. Each county will be transformed into its centroid. Those centroids are not evenly spaced, and we expect the dataset may be modeled incorrectly for several steps.

[11]:
```
# Read and prepare data

DATASET = 'samples/regularization/cancer_data.gpkg'
POLYGON_LAYER = 'areas'
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'
MAX_RANGE = 400000
STEP_SIZE = 40000

AREAL_INPUT = Blocks()
AREAL_INPUT.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
```

(continues on next page)

(continued from previous page)

```
↪name=POLYGON_LAYER)

AREAL_INPUT.data.head()
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

[11]:

```
    FIPS                                           geometry   rate  \
0  25019  MULTIPOLYGON (((2115688.816 556471.240, 211569...  192.2
1  36121  POLYGON ((1419423.430 564830.379, 1419729.721 ...  166.8
2  33001  MULTIPOLYGON (((1937530.728 779787.978, 193751...  157.4
3  25007  MULTIPOLYGON (((2074073.532 539159.504, 207411...  156.7
4  25001  MULTIPOLYGON (((2095343.207 637424.961, 209528...  155.3

     centroid_x     centroid_y
0  2.132630e+06  557971.155949
1  1.442153e+06  550673.935704
2  1.958207e+06  766008.383446
3  2.082188e+06  556830.822367
4  2.100747e+06  600235.845891
```

[12]:

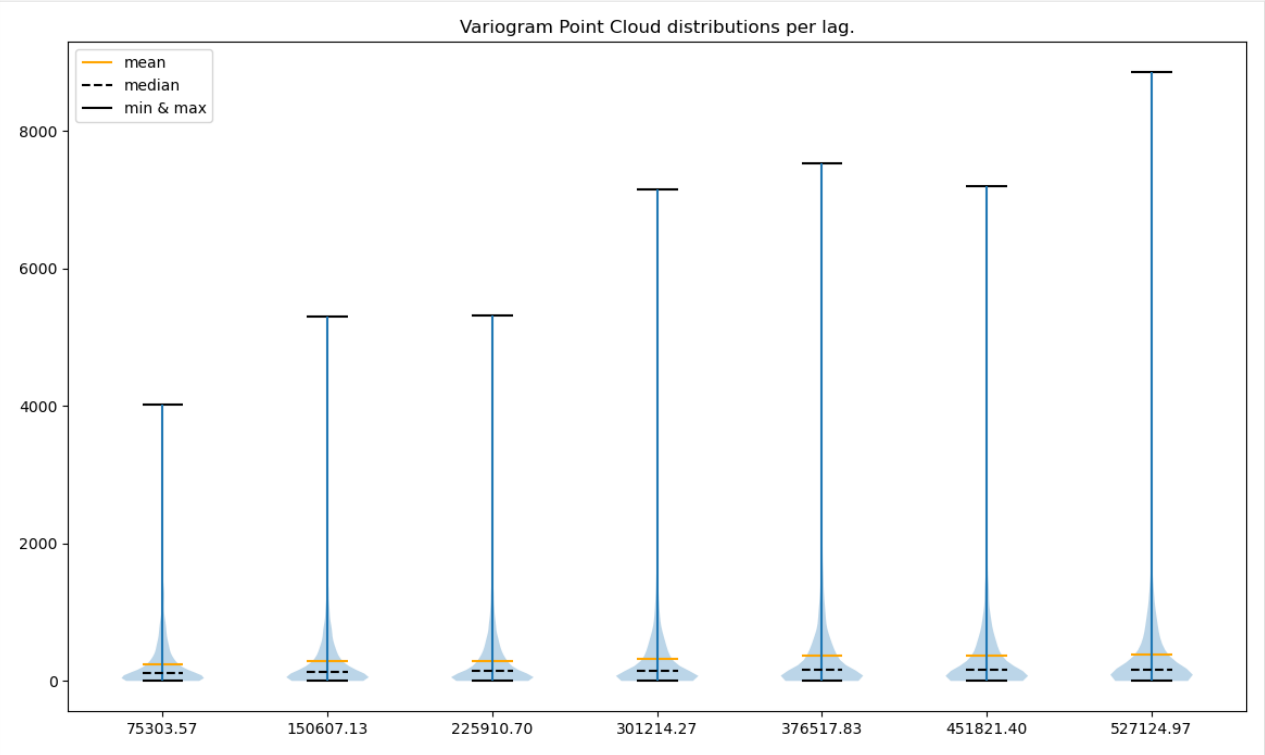
```
areal_centroids = AREAL_INPUT.data[[AREAL_INPUT.cx, AREAL_INPUT.cy, AREAL_INPUT.value_
↪column_name]].values
```

[13]:

```
# Create analysis parameters

# Check max distance between points

distances = calc_point_to_point_distance(areal_centroids[:, :-1])
maximum_range = np.max(distances) / 2

number_of_lags = 8
step_size = maximum_range / number_of_lags

vc = VariogramCloud(input_array=areal_centroids, step_size=step_size, max_range=maximum_
↪range)
```

[14]:

```
vc.plot('violin')
```



[14]:
```
True
```

[15]:
```
for idx, _lag in enumerate(vc.lags):
    print(f'Lag {_lag} has {vc.points_per_lag[idx]} point pairs.')
```

```
Lag 75303.56650292415 has 1936 point pairs.
Lag 150607.1330058483 has 4740 point pairs.
Lag 225910.69950877246 has 6532 point pairs.
Lag 301214.2660116966 has 7254 point pairs.
Lag 376517.83251462074 has 7066 point pairs.
Lag 451821.39901754487 has 5904 point pairs.
Lag 527124.9655204691 has 4494 point pairs.
```

Our data is skewed towards large values. Thus, we will remove outliers from the upper part of the dataset. We will use the interquartile range algorithm. It detects outliers as all points below the first quartile of a data plus `m` (positive or zero `float`) standard deviations and all points above the 3rd quartile plus `n` (positive or zero `float`) standard deviations. The important thing is that the absolute values of `m` and `n` can differ. We should fit those values to the distribution and its skewness.

The class `VariogramCloud` has the internal method `.remove_outliers()`. With the parameter `inplace` set to `True`, we can overwrite the variogram point cloud, but if we set it to `False`, it will return a new `VariogramCloud` object with a cleaned variogram point cloud. Other parameters:

- `method`: is a string that removes outliers. We have two methods, one based on **z-scores** and the **interquartile range** method. Both have upper and lower bounds. The Z-score method is invoked with `method='zscore'`, and arguments for this function are `z_lower_limit` and `z_upper_limit` (number of standard deviations from the mean down and up, or negative and positive).
- `iqr_lower_limit`: how many standard deviations should be subtracted from the first quartile to set a limit of valid values,
- `iqr_upper_limit`: how many standard deviations should be added to the 3rd quartile to set a limit of valid

values.

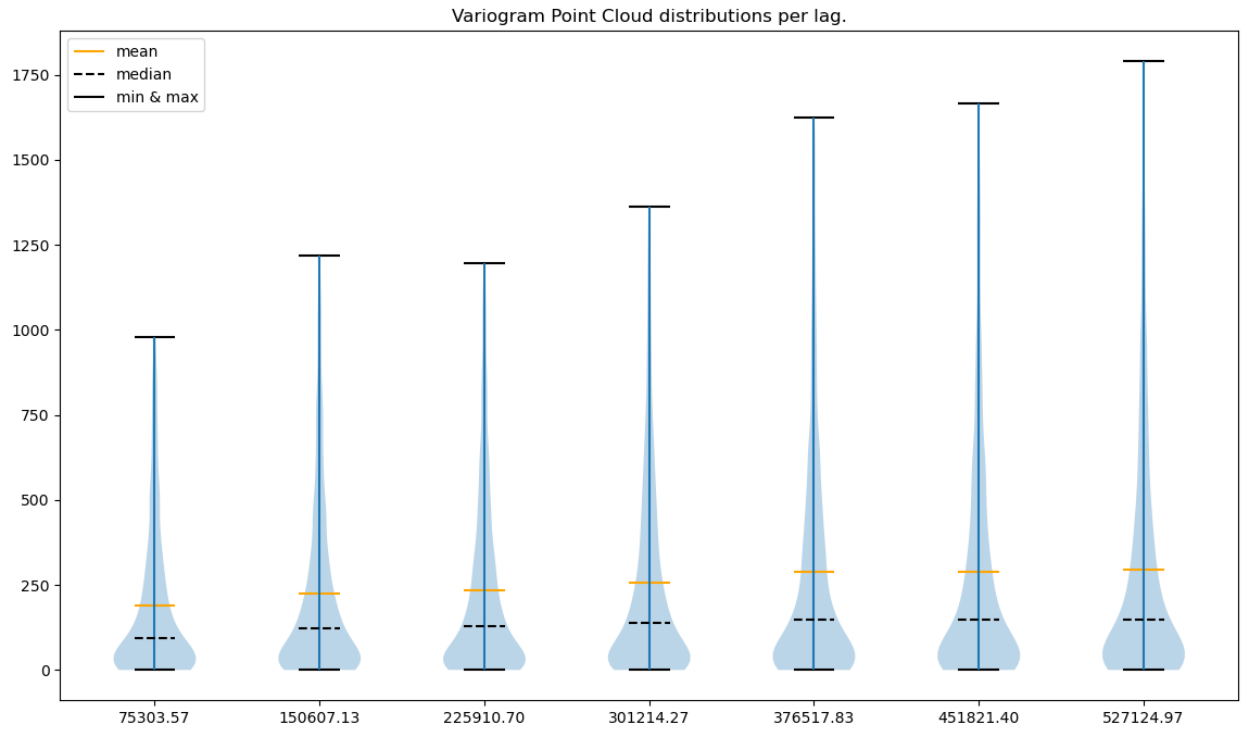
The values we will get will be within limits:

$$q1 - (m * std) < V < q3 + (n * std),$$

Where: * $q1$: 1st quartile, * $q3$: 3rd quartile, * std: standard deviation, * m: real value greater than 0, * n: real value greater than 0, * V: the cleaned array.

[16]:
```
cvc = vc.remove_outliers(method='iqr', iqr_lower_limit=3, iqr_upper_limit=2,␣
↪inplace=False)
```

[17]:
```
cvc.plot('violin')
```



[17]:
```
True
```

When we compare both figures - before and after data cleaning - we see that the y-axis of the second plot is 6-7 times smaller than the y-axis of the first plot! We've cleaned our data from the extreme values.

4) Calculate experimental semivariogram from point cloud

The last but not least property of the experimental variogram point cloud is that we may calculate the semivariogram directly from it. The method for it is `.calculate_experimental_variogram()`. Let's compare outputs from two algorithms.

[18]:
```
exp_variogram_from_points = calculate_semivariance(areal_centroids, step_size=step_size,␣
↪max_range=maximum_range)
exp_variogram_from_p_cloud = vc.calculate_experimental_variogram()
```

[19]:
```
np.array_equal(exp_variogram_from_points, exp_variogram_from_p_cloud)
```

[19]:
```
True
```

Both semivariograms are the same, but methods to obtain these were different.

Where to go from here?

- A.2.3 Experimental Variogram and Variogram Point Cloud Classes
- B.1.1 Ordinary and Simple Kriging
- B.1.3 Outliers and Kriging Model
- B.2.1 Directional Ordinary Kriging

Changelog

Date	Change description	Author
2023-08-21	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2023-02-01	Tutorial updated for the version 0.3.7 of the package	@SimonMolinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@SimonMolinsky
2022-10-21	Updated to the version 0.3.4	@SimonMolinsky
2022-10-17	Corrected data selection, now tutorial will run faster	@SimonMolinsky
2022-08-17	Updated to the version 0.3.0	@SimonMolinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@SimonMolinsky
2021-10-13	Refactored TheoreticalSemivariogram (name change of class attribute) and refactored `calc_semivariance_from_pt_cloud()` function to protect calculations from `NaN's`.	@ethmtrgt & @SimonMolinsky
2021-08-22	Refactored the Outlier Removal algorithm - quantile based algorithm	@SimonMolinsky
2021-08-10	Refactored the Outlier Removal algorithm	@SimonMolinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@SimonMolinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@SimonMolinsky

[ ]:

Experimental Variogram and Variogram Cloud classes

Table of Contents:

1. Create Experimental Variogram with the `ExperimentalVariogram` class.
2. Create Experimental Variogram Point Cloud with the `VariogramCloud` class.

Introduction

The geostatistical analysis starts with a dissimilarity estimation. We must know if a process is spatially dependent and at which distance points tend to be related to each other. Thus, we start with the experimental variogram estimation. In this tutorial, we will look closely into the API and learn what can be done with two basic experimental semivariogram estimation functionalities. One is the `ExperimentalVariogram` class, and another is the `VariogramCloud` class. The first is a foundation of every other complex function, from semivariogram modeling to Poisson Kriging. The latter is used to analyze relations between points and their dispersion.

We will use **DEM data** stored in a file `samples/point_data/txt/pl_dem_epsg2180.txt`

Import packages & read data

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pyinterpolate import read_txt, ExperimentalVariogram, VariogramCloud
```

[2]:
```
dem = read_txt('samples/point_data/txt/pl_dem_epsg2180.txt')

# Take a sample
sample_size = int(0.05 * len(dem))
indices = np.random.choice(len(dem), sample_size, replace=False)
dem = dem[indices]

# Look into a first few lines of data
dem[:10, :]
```

[2]:
```
array([[2.39250338e+05, 5.48631524e+05, 8.84804077e+01],
       [2.41201861e+05, 5.49103756e+05, 7.72773132e+01],
       [2.50945606e+05, 5.48731991e+05, 8.15302811e+01],
       [2.50384028e+05, 5.36743412e+05, 3.87784348e+01],
       [2.53739451e+05, 5.44230034e+05, 2.12480278e+01],
       [2.42744089e+05, 5.29666893e+05, 5.23056946e+01],
       [2.38056136e+05, 5.34391921e+05, 2.95818558e+01],
       [2.45023500e+05, 5.51069770e+05, 3.36740112e+01],
       [2.44771460e+05, 5.32997525e+05, 4.72923431e+01],
       [2.53090205e+05, 5.48829925e+05, 8.16635361e+01]])
```

1. Experimental Variogram class

Let's start from the `ExperimentalVariogram` class.

```
class ExperimentalVariogram:

    def __init__(self,
                 input_array,
                 step_size,
                 max_range,
                 weights=None,
                 direction=None,
                 tolerance=1.0,
                 method='t',
                 is_semivariance=True,
                 is_covariance=True):
        ...
```

The class has three required parameters and seven optional.

- `input_array` is a list of coordinates and values in the form `(x, y, value)`. Those are our observations.
- `step_size` parameter, which describes the distance between lags.
- `max_range`, which describes the maximum range of analysis.

What do we know at the beginning? Our coordinate reference system is metric. It is EPSG:2180. We don't know the limits of the region of interest to set the `max_range` parameter. We must define our area size:

[3]:
```
# Get max distances for a region

minx = np.min(dem[:, 0])
maxx = np.max(dem[:, 0])

miny = np.min(dem[:, 1])
maxy = np.max(dem[:, 1])

x_max_distance = maxx - minx
y_max_distance = maxy - miny

print('Max lon distance:', x_max_distance)
print('Max lat distance:', y_max_distance)
```

```
Max lon distance: 17926.58479234294
Max lat distance: 25125.91187161021
```

We can assume that the maximum distance `max_range` parameter shouldn't be larger than the maximum distance in a lower dimension (**~18000** m). In reality, the `max_range` parameter is relatively close to the halved value of this distance, and we can check it on a variogram.

But we don't know the `step_size` parameter... Let's assume it is 100 meters, and we will see if we've estimated it correctly.

[4]:
```
max_range = 18000
step_size = 100
```

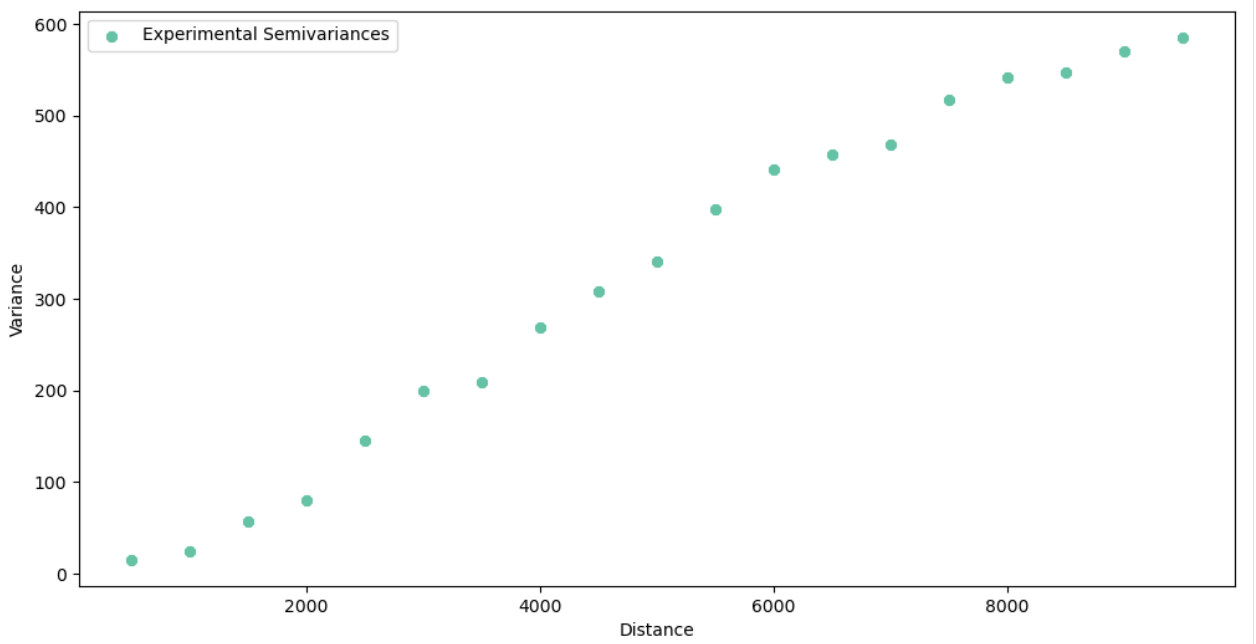
(continues on next page)

(continued from previous page)

```
experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range
)
```

We will visually inspect semivariogram with the `.plot()` method of `ExperimentalVariogram` class:

[10]:
```
experimental_variogram.plot(plot_covariance=False, plot_variance=False)
```

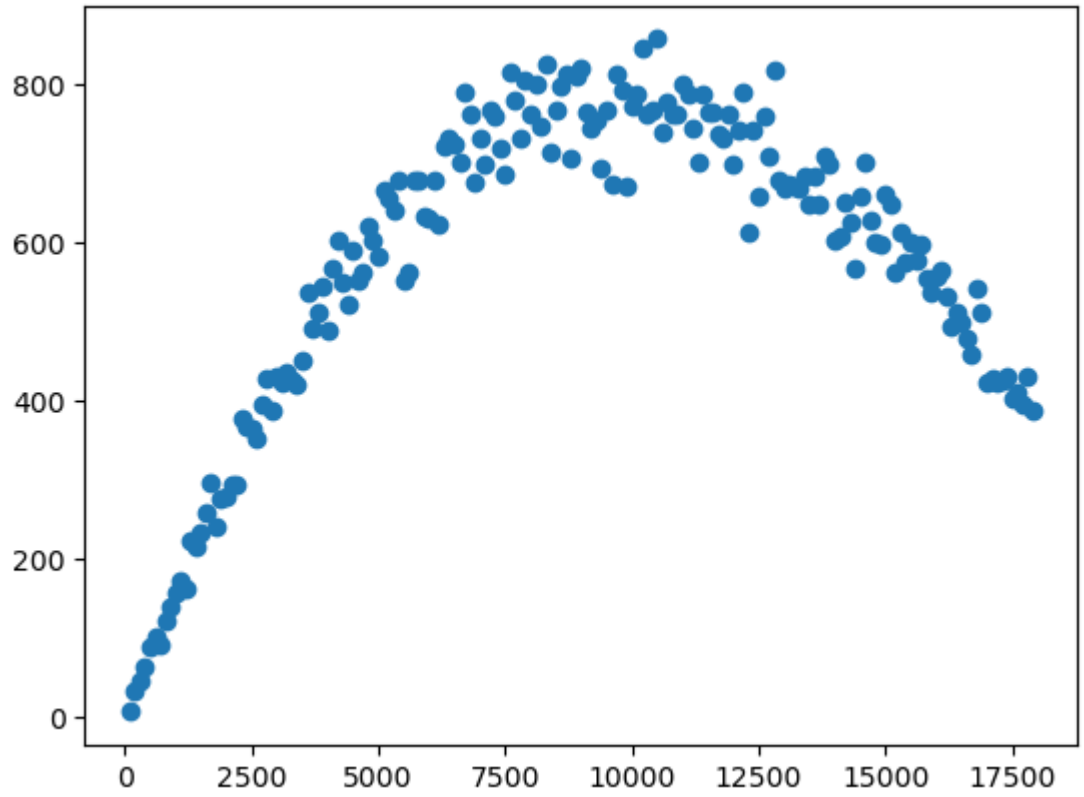


Variogram looks nice. We can see a trend, but points are oscillating around it. This piece of information tells us that `step_size` should be larger.

What with the `max_range`? It is a tricky problem because the semivariogram seems right up to the end of its range. The idea is to set the maximum range around half of the study extent because the number of point pairs will fall from this distance.

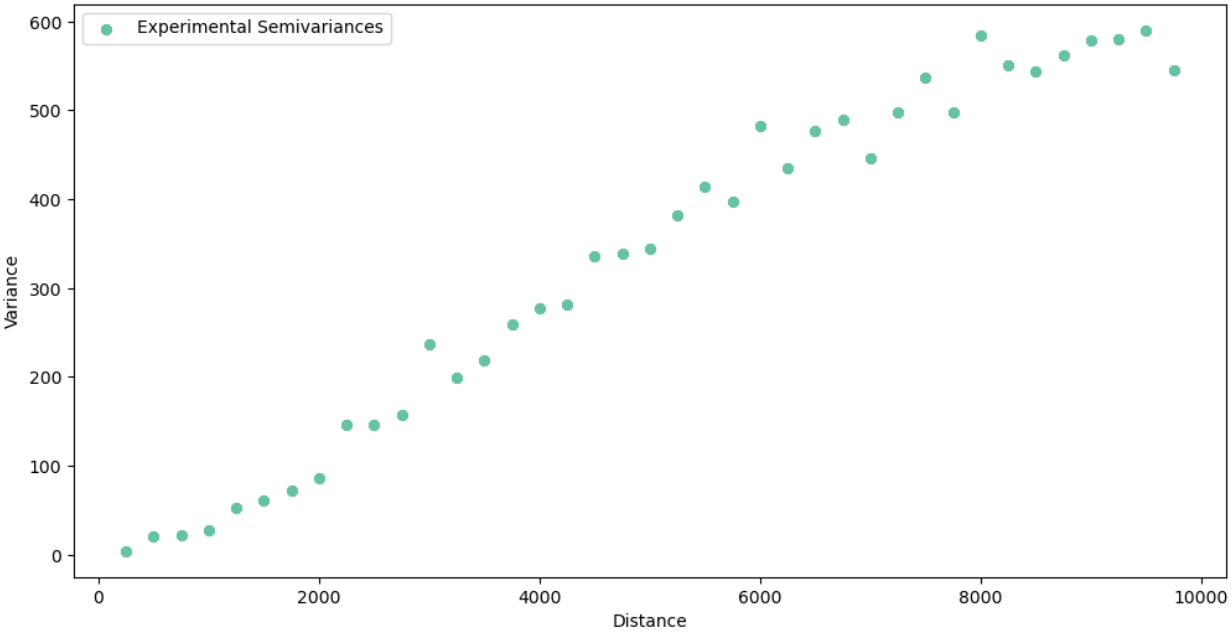
We can check how many point pairs we have for each distance to make a better decision. The `ExperimentalVariogram` class stores information about points within its attribute `.experimental_semivariance_array`. It is a `numpy` array with three columns: `[lag, semivariance, number of point pairs]`, and we will use the last column to decide where to cut `max_range`.

[7]:
```
_ds = experimental_variogram.experimental_semivariance_array
plt.figure()
plt.scatter(x=_ds[:, 0], y=_ds[:, -1])
plt.show()
```



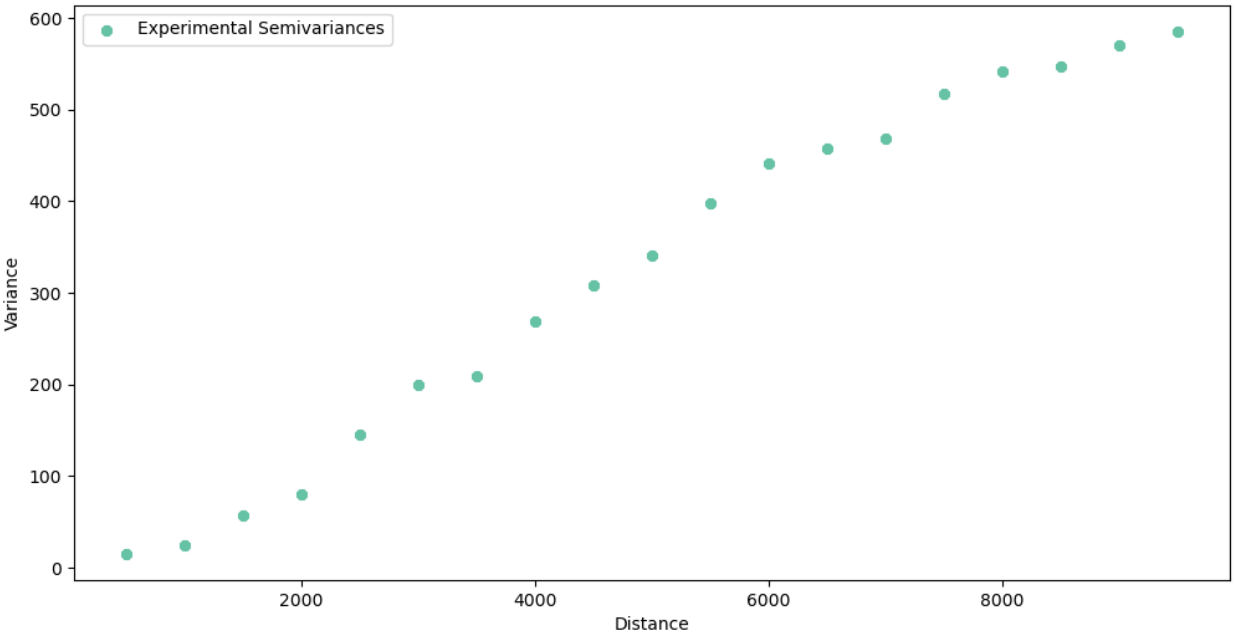
Around the twelve-kilometer number of point pairs rapidly falls, so we can set `max_range` to some value between 7.5 and 12.5 kilometers, we will choose **10000** kilometers as our `max_range`.

[11]:
```
max_range = 10_000
step_size = 250

experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range
)

experimental_variogram.plot(plot_covariance=False, plot_variance=False)
```



This time variogram looks cleaner. However, it still has oscillating readings. Let's set `step_size` to 500 meters:

[12]:
```
step_size = 500

experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range
)

experimental_variogram.plot(plot_covariance=False, plot_variance=False)
```



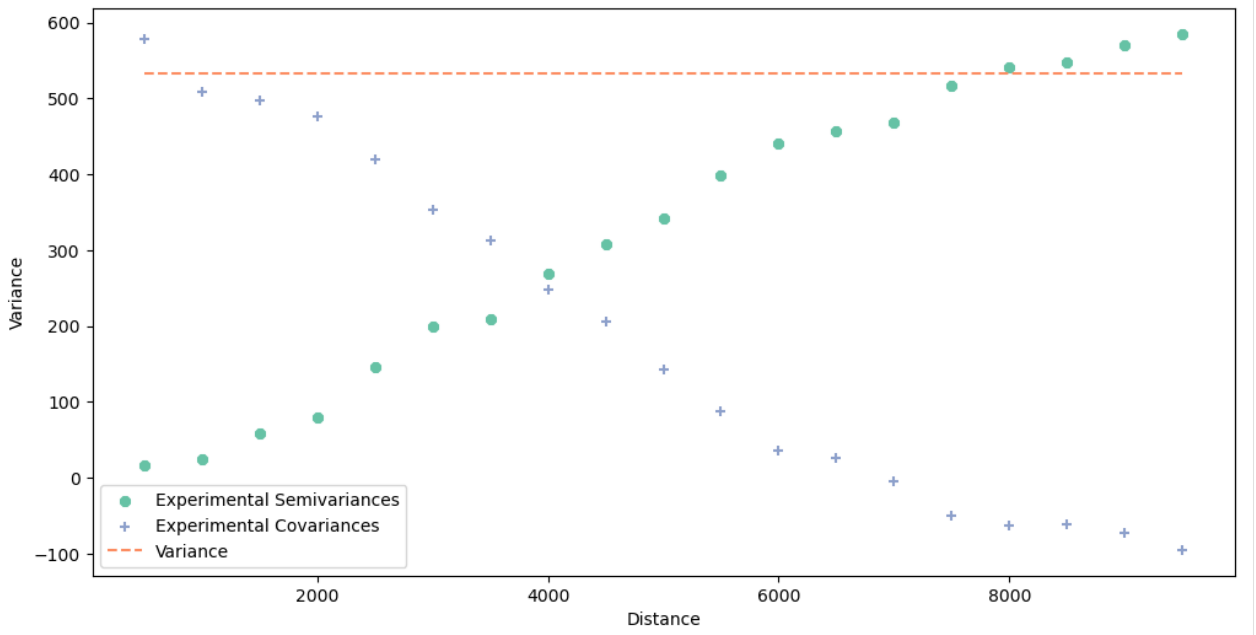
We can use those parameters! Next, we will look into the `is_covariance` parameter of the `ExperimentalVariogram` class. It is set to `True` by default. If we want to know the covariance of a dataset, then we should set the `is_covariance` parameter to `True`. Covariance is a measure of spatial similarity, opposite to semivariance. You will see in the plot that they mirror each other. But we can use it in kriging systems instead of semivariance.

`ExperimentalVariogram` plotting function `.plot()` has additional parameters:

- `plot_semivariance` (boolean, set to `True` by default),
- `plot_covariance` (boolean, set to `True` by default),
- `plot_variance` (boolean, set to `True` by default).

We will plot all of those objects in a single figure.

[13]:
```
experimental_variogram.plot()
```



Two important things that we can read from this plot:

1. Experimental covariance is a mirrored version of experimental semivariance, and we can use it to solve the Kriging system instead of semivariance. It is a measure of similarity between point pairs. The highest similarity can be noticed on the first lags and decreases with a rising distance.
2. Variance calculated from data is used later as the **sill** in kriging models, and it can be used to transform semivariances into covariances.

This plot summarizes the basic steps we can do with the `ExperimentalVariogram` class. We left many parameters untouched, but only for a moment. Now we are going to explore all capabilities of the class.

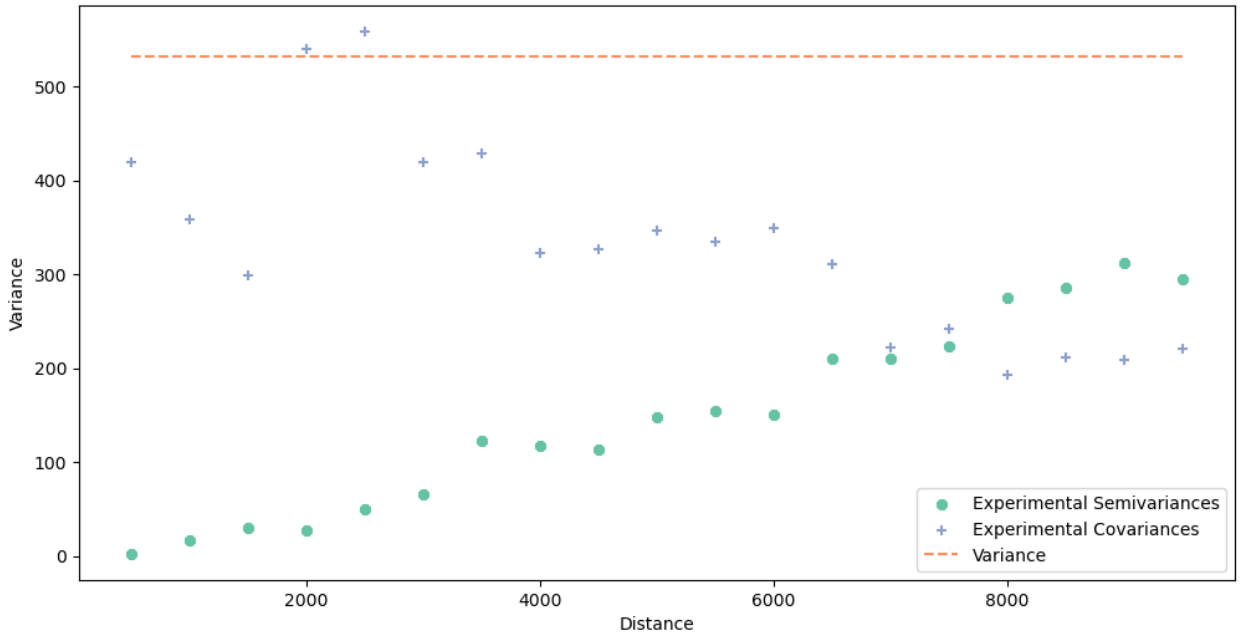
Optional parameter: `weights`

We won't use this parameter in our tutorial, but we should learn where it comes from. It is a part of **Poisson Kriging** operations. **Poisson Kriging** algorithms use `weights` to weight the semivariance between blocks by in-block population.

Optional parameters: `direction, tolerance, method`

Those three parameters are used to define a directional variogram. We will check how the variogram behaves in the Weast-East axis (0 degrees).

[14]:
```
direction = 0
tolerance = 0.1
method = 'e'


experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range,
    direction=direction,
    tolerance=tolerance,
    method=method
)

experimental_variogram.plot()
```



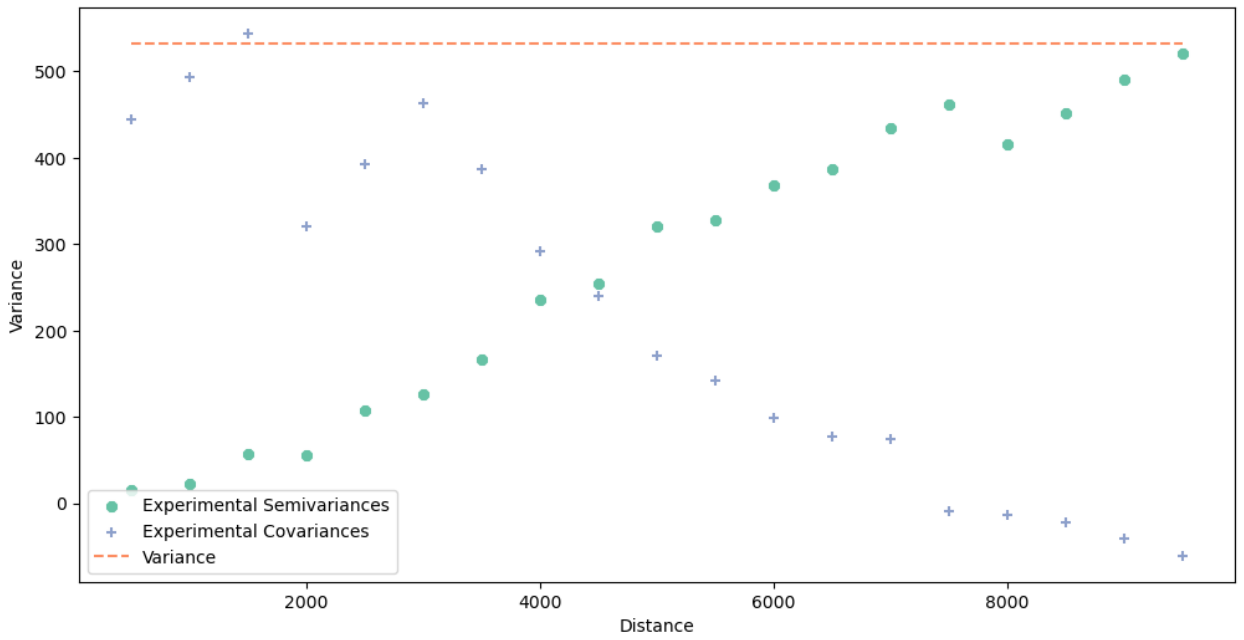
Let's look into the North-South axis too:

[15]:
```
direction = 90
```

(continues on next page)

(continued from previous page)

```
experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range,
    direction=direction,
    tolerance=tolerance,
    method=method
)

experimental_variogram.plot()
```



What can we learn from those plots? Variability along the W-E axis is lower than along the N-S axis. It may indicate continuous spatial structures along the W-E axis (for example, a river plain).

At this point, we are ready for modeling. However, if we want to be precise, we should look into outliers and analyze `VariogramCloud`.

A classic variogram analysis tells us almost everything about spatial dependencies and dissimilarities in data. But there are more spatial investigation tools. We need to learn about the distribution of variances within a single lag, which is valuable information. Semivariogram shows us averaged semivariances. If we want to dig deeper, we should use the `VariogramCloud` class and plot... the variogram cloud.

The `VariogramCloud` class definition API is:

```
class VariogramCloud:

    def __init__(self,
                 input_array,
                 step_size,
                 max_range,
                 direction=0,
                 tolerance=1,
                 calculate_on_creation=True):
```

(continues on next page)

(continued from previous page)

```
    pass
```

We pass the same parameters to the class as for the `ExperimentalVariogram` class, and the only difference is the `calculate_on_creation` boolean value. Variogram point cloud calculation is a slow operation, and that's why we can start it during or after the class initialization. If we set `calculate_on_creation` to `False`, we must run the `.calculate_experimental_variogram()` method later.

`VariogramCloud` has more methods than `ExperimentalVariogram`:

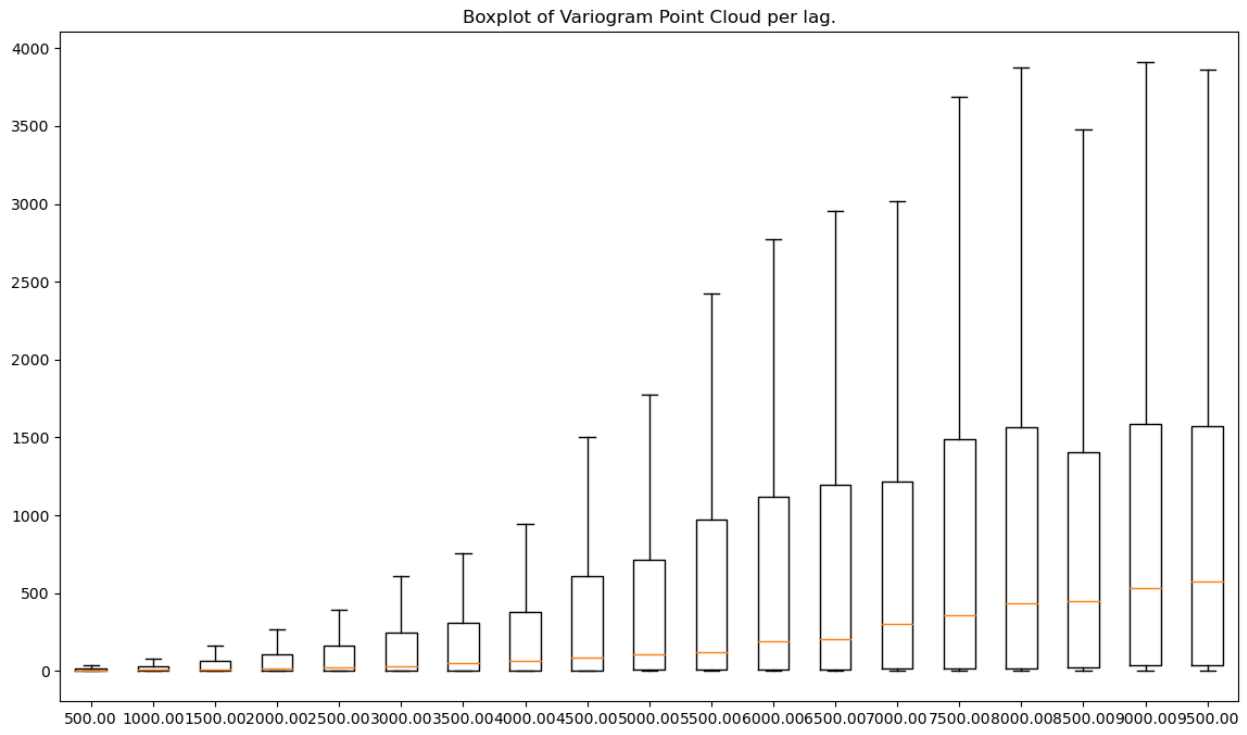
- `calculate_experimental_variogram()` to obtain the variogram (similar to the output from the `ExperimentalVariogram` class,
- `describe()` to get lag statistics,
- `plot()` to plot variogram cloud,
- `remove_outliers()` to clean the variogram.

We will start with calculation and plotting variogram cloud.

[16]:
```
variogram_cloud = VariogramCloud(
    input_array=dem,
    step_size=step_size,
    max_range=max_range
)
```

We can plot three different kinds of distribution plots: `'scatter', 'box', 'violin'`. We will choose `box` because it clearly shows data distribution in contrast to `scatter` and it's easier to interpret than `violin` plot.

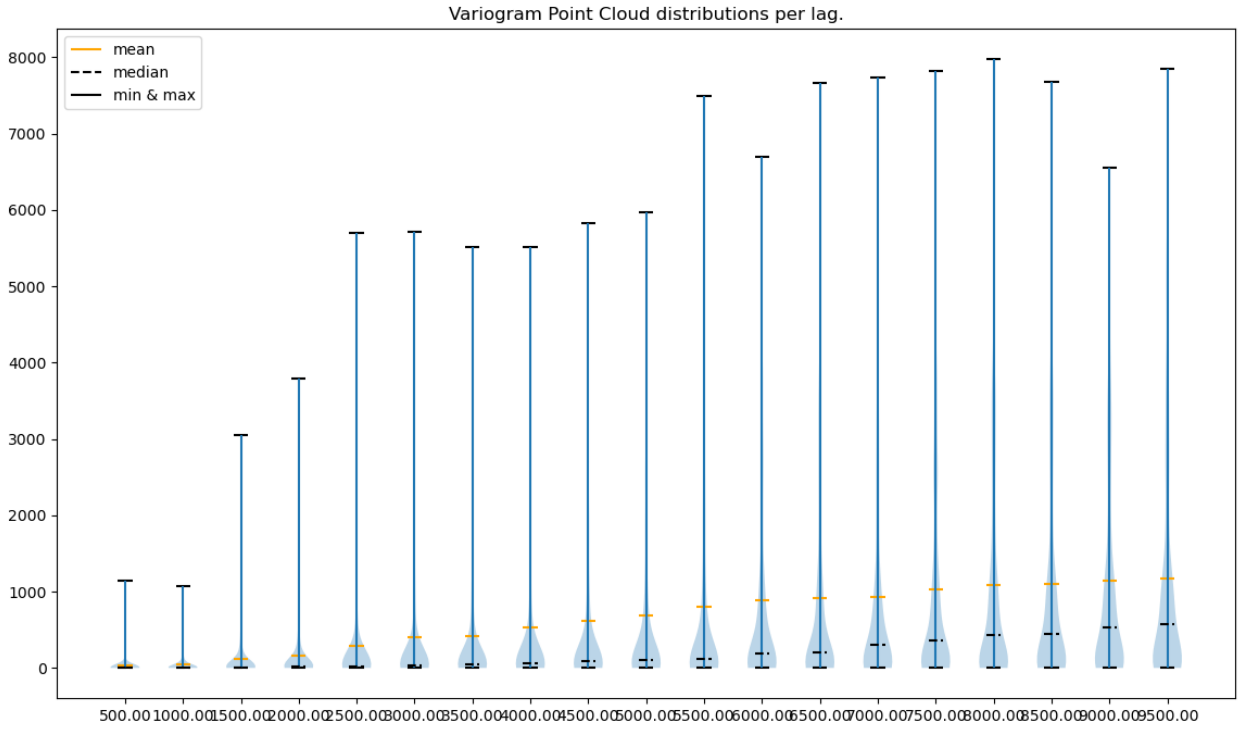
[17]:
```
variogram_cloud.plot(kind='box')
```



[17]:
```
True
```

The general shape is similar to the shape of the experimental variogram. We can see that semivariances between point pairs contain outliers, especially for the lags close to our maximum range. To be sure about outliers' existence, let's look into a more complex violin plot:

[18]:
```
variogram_cloud.plot(kind='violin')
```



[18]:
```
True
```

Extreme outliers are more visible. Another thing is the difference between the mean and median. Distribution is highly skewed toward large values. If graphical representation is not enough, we may describe statistics with the method `.describe()`:

[19]:
```
variogram_cloud.describe()
```

[19]:
```
{500: {'count': 234,
  'avg semivariance': 15.48837895423851,
  'std': 129.30689361933185,
  'min': 0.0,
  'max': 1143.5633926376795,
  '25%': 0.3451266676565865,
  'median': 1.3447192054627521,
  '75%': 13.65846407461504,
  'skewness': 6.865653366086163,
  'kurtosis': 50.79024671616988,
  'lag': 500},
 1000: {'count': 606,
  'avg semivariance': 25.151443803715708,
  'std': 130.72724691457688,
  'min': 1.3315911928657442e-06,
  'max': 1077.959058797118,
  '25%': 0.7740007161801259,
```

(continues on next page)

(continued from previous page)

```
 'median': 5.823250695397292,
 '75%': 31.42736609817075,
 'skewness': 4.567298658957882,
 'kurtosis': 24.214435519746083,
 'lag': 1000},
1500: {'count': 1008,
 'avg semivariance': 57.835574935521514,
 'std': 331.73215147893416,
 'min': 7.597882358822972e-05,
 'max': 3046.333026852517,
 '25%': 0.968566885097971,
 'median': 7.822932168508487,
 '75%': 64.66452046956692,
 'skewness': 5.608396965535479,
 'kurtosis': 37.38797721538239,
 'lag': 1500},
2000: {'count': 1348,
 'avg semivariance': 79.75218552208693,
 'std': 428.8461018481998,
 'min': 3.025872865691781e-05,
 'max': 3796.9629175248883,
 '25%': 1.2690777453244664,
 'median': 18.391886290972252,
 '75%': 109.60415145155275,
 'skewness': 4.869674455730696,
 'kurtosis': 27.620361886790118,
 'lag': 2000},
2500: {'count': 1698,
 'avg semivariance': 145.95094727929936,
 'std': 805.2460216899517,
 'min': 4.243338480591774e-08,
 'max': 5699.690698992359,
 '25%': 1.7351306386444776,
 'median': 22.872865243130946,
 '75%': 162.32509036679767,
 'skewness': 4.209560111010213,
 'kurtosis': 18.750587407226615,
 'lag': 2500},
3000: {'count': 1992,
 'avg semivariance': 199.33218908929288,
 'std': 911.2555224174007,
 'min': 1.0608346201479435e-08,
 'max': 5720.016503487772,
 '25%': 2.4267591861680557,
 'median': 32.15229412448389,
 '75%': 249.17055305647318,
 'skewness': 3.2787661143480946,
 'kurtosis': 11.293592711474577,
 'lag': 3000},
3500: {'count': 2154,
 'avg semivariance': 209.00387222055934,
 'std': 913.5411306035585,
```

(continues on next page)

(continued from previous page)

```
 'min': 6.0737496824003756e-05,
 'max': 5516.191060331708,
 '25%': 2.9810246527231357,
 'median': 48.88620506064035,
 '75%': 306.07525457920565,
 'skewness': 3.132272141037236,
 'kurtosis': 9.972795491912944,
 'lag': 3500},
4000: {'count': 2572,
 'avg semivariance': 268.3259773207153,
 'std': 1056.4261950767657,
 'min': 2.6099187380168587e-06,
 'max': 5514.402312608567,
 '25%': 2.6644885819441697,
 'median': 61.834192301528674,
 '75%': 381.5180078523699,
 'skewness': 2.473310799225225,
 'kurtosis': 5.473415218908773,
 'lag': 4000},
4500: {'count': 2832,
 'avg semivariance': 307.6134533281816,
 'std': 1113.7493026553932,
 'min': 4.211964551359415e-06,
 'max': 5829.092762578512,
 '25%': 5.008626806946268,
 'median': 82.54248778484907,
 '75%': 606.8352795074716,
 'skewness': 2.3016213999312303,
 'kurtosis': 4.838611890454055,
 'lag': 4500},
5000: {'count': 2924,
 'avg semivariance': 341.33197977216213,
 'std': 1184.6820756378802,
 'min': 7.017538882791996e-05,
 'max': 5967.1652707402245,
 '25%': 6.364002417831216,
 'median': 105.43215119686101,
 '75%': 714.8062540315768,
 'skewness': 2.217432052486689,
 'kurtosis': 4.458010177884344,
 'lag': 5000},
5500: {'count': 3192,
 'avg semivariance': 397.7678526465361,
 'std': 1310.190495093556,
 'min': 3.9138831198215485e-07,
 'max': 7494.872101836896,
 '25%': 7.306133981550374,
 'median': 117.5330814906647,
 '75%': 973.5924935907096,
 'skewness': 2.048285104289412,
 'kurtosis': 3.80948479120119,
 'lag': 5500},
```

(continues on next page)

(continued from previous page)


```
6000: {'count': 3188,
 'avg semivariance': 441.4588322976255,
 'std': 1341.9100279445179,
 'min': 4.899957275483757e-07,
 'max': 6701.4388024286745,
 '25%': 9.152769157113653,
 'median': 191.89130915285932,
 '75%': 1115.490027037231,
 'skewness': 1.8576607961887481,
 'kurtosis': 2.8155666464376194,
 'lag': 6000},
6500: {'count': 3480,
 'avg semivariance': 457.2660981146536,
 'std': 1364.1466464135244,
 'min': 6.846037285868078e-05,
 'max': 7663.801433086061,
 '25%': 10.418562886014115,
 'median': 204.0729240650271,
 '75%': 1193.259201243348,
 'skewness': 1.8383466986007484,
 'kurtosis': 3.1069471071828714,
 'lag': 6500},
7000: {'count': 3662,
 'avg semivariance': 467.70408815019107,
 'std': 1383.7114275988758,
 'min': 2.620747545734048e-05,
 'max': 7732.3306628377795,
 '25%': 12.876975894745556,
 'median': 302.43025554287306,
 '75%': 1214.8304466051668,
 'skewness': 1.9378885777226609,
 'kurtosis': 3.5563617512199306,
 'lag': 7000},
7500: {'count': 3634,
 'avg semivariance': 516.9270183174606,
 'std': 1389.344310228298,
 'min': 1.1408701539039612e-06,
 'max': 7813.73612767295,
 '25%': 18.949909821152687,
 'median': 359.12963762053187,
 '75%': 1490.0087687096093,
 'skewness': 1.554949463509357,
 'kurtosis': 1.7478735159246037,
 'lag': 7500},
8000: {'count': 3896,
 'avg semivariance': 541.1530674487567,
 'std': 1428.070471120106,
 'min': 0.00011741125854314305,
 'max': 7973.467257830838,
 '25%': 19.034676041905186,
 'median': 433.6325574710372,
 '75%': 1563.9827408457786,
```


(continues on next page)

(continued from previous page)

```
 'skewness': 1.528514122993604,
 'kurtosis': 1.7506732493933423,
 'lag': 8000},
8500: {'count': 3858,
 'avg semivariance': 547.5056312500515,
 'std': 1468.6396045259646,
 'min': 1.7320417100563645e-06,
 'max': 7673.916123840958,
 '25%': 23.704184239628376,
 'median': 448.1990261501778,
 '75%': 1405.225225292179,
 'skewness': 1.5977571608298324,
 'kurtosis': 1.8190966680505722,
 'lag': 8500},
9000: {'count': 3948,
 'avg semivariance': 570.1379938012599,
 'std': 1447.8532805956781,
 'min': 3.0405793950194493e-05,
 'max': 6551.827118135992,
 '25%': 37.25730361373644,
 'median': 532.4391559513406,
 '75%': 1588.2390081191297,
 'skewness': 1.4628469824631245,
 'kurtosis': 1.232784763520586,
 'lag': 9000},
9500: {'count': 3730,
 'avg semivariance': 585.3034449746383,
 'std': 1511.2436330394091,
 'min': 0.00023756933296681382,
 'max': 7855.885675042067,
 '25%': 39.582661400436336,
 'median': 571.2002264822368,
 '75%': 1571.7003516764962,
 'skewness': 1.5794687444921045,
 'kurtosis': 1.8317167345076042,
 'lag': 9500}}
```

We get Python's `dict` as an output, but it is not especially readable; let's convert it to a `DataFrame`:

[20]:
```
desc = pd.DataFrame(variogram_cloud.describe())
```

[21]:
```
desc
```

[21]:

	500	1000	1500	2000	\
count	234.000000	606.000000	1008.000000	1348.000000	
avg semivariance	15.488379	25.151444	57.835575	79.752186	
std	129.306894	130.727247	331.732151	428.846102	
min	0.000000	0.000001	0.000076	0.000030	
max	1143.563393	1077.959059	3046.333027	3796.962918	
25%	0.345127	0.774001	0.968567	1.269078	
median	1.344719	5.823251	7.822932	18.391886	
75%	13.658464	31.427366	64.664520	109.604151	

(continues on next page)

(continued from previous page)

```
skewness              6.865653     4.567299     5.608397     4.869674
kurtosis             50.790247    24.214436    37.387977    27.620362
lag                 500.000000  1000.000000  1500.000000  2000.000000

                          2500          3000         3500         4000  \
count             1.698000e+03  1.992000e+03  2154.000000  2572.000000
avg semivariance  1.459509e+02  1.993322e+02   209.003872   268.325977
std               8.052460e+02  9.112555e+02   913.541131  1056.426195
min               4.243338e-08  1.060835e-08     0.000061     0.000003
max               5.699691e+03  5.720017e+03  5516.191060  5514.402313
25%               1.735131e+00  2.426759e+00     2.981025     2.664489
median            2.287287e+01  3.215229e+01    48.886205    61.834192
75%               1.623251e+02  2.491706e+02   306.075255   381.518008
skewness          4.209560e+00  3.278766e+00     3.132272     2.473311
kurtosis          1.875059e+01  1.129359e+01     9.972795     5.473415
lag               2.500000e+03  3.000000e+03  3500.000000  4000.000000

                         4500         5000          5500          6000  \
count             2832.000000  2924.000000  3.192000e+03  3.188000e+03
avg semivariance   307.613453   341.331980  3.977679e+02  4.414588e+02
std               1113.749303  1184.682076  1.310190e+03  1.341910e+03
min                  0.000004     0.000070  3.913883e-07  4.899957e-07
max               5829.092763  5967.165271  7.494872e+03  6.701439e+03
25%                  5.008627     6.364002  7.306134e+00  9.152769e+00
median              82.542488   105.432151  1.175331e+02  1.918913e+02
75%                606.835280   714.806254  9.735925e+02  1.115490e+03
skewness             2.301621     2.217432  2.048285e+00  1.857661e+00
kurtosis             4.838612     4.458010  3.809485e+00  2.815567e+00
lag               4500.000000  5000.000000  5.500000e+03  6.000000e+03

                         6500         7000         7500         8000  \
count             3480.000000  3662.000000  3634.000000  3896.000000
avg semivariance   457.266098   467.704088   516.927018   541.153067
std               1364.146646  1383.711428  1389.344310  1428.070471
min                  0.000068     0.000026     0.000001     0.000117
max               7663.801433  7732.330663  7813.736128  7973.467258
25%                 10.418563    12.876976    18.949910    19.034676
median             204.072924   302.430256   359.129638   433.632557
75%               1193.259201  1214.830447  1490.008769  1563.982741
skewness             1.838347     1.937889     1.554949     1.528514
kurtosis             3.106947     3.556362     1.747874     1.750673
lag               6500.000000  7000.000000  7500.000000  8000.000000

                         8500         9000         9500
count             3858.000000  3948.000000  3730.000000
avg semivariance   547.505631   570.137994   585.303445
std               1468.639605  1447.853281  1511.243633
min                  0.000002     0.000030     0.000238
max               7673.916124  6551.827118  7855.885675
25%                 23.704184    37.257304    39.582661
median             448.199026   532.439156   571.200226
75%               1405.225225  1588.239008  1571.700352
```

(continues on next page)

(continued from previous page)

```
skewness              1.597757     1.462847     1.579469
kurtosis              1.819097     1.232785     1.831717
lag                8500.000000  9000.000000  9500.000000
```

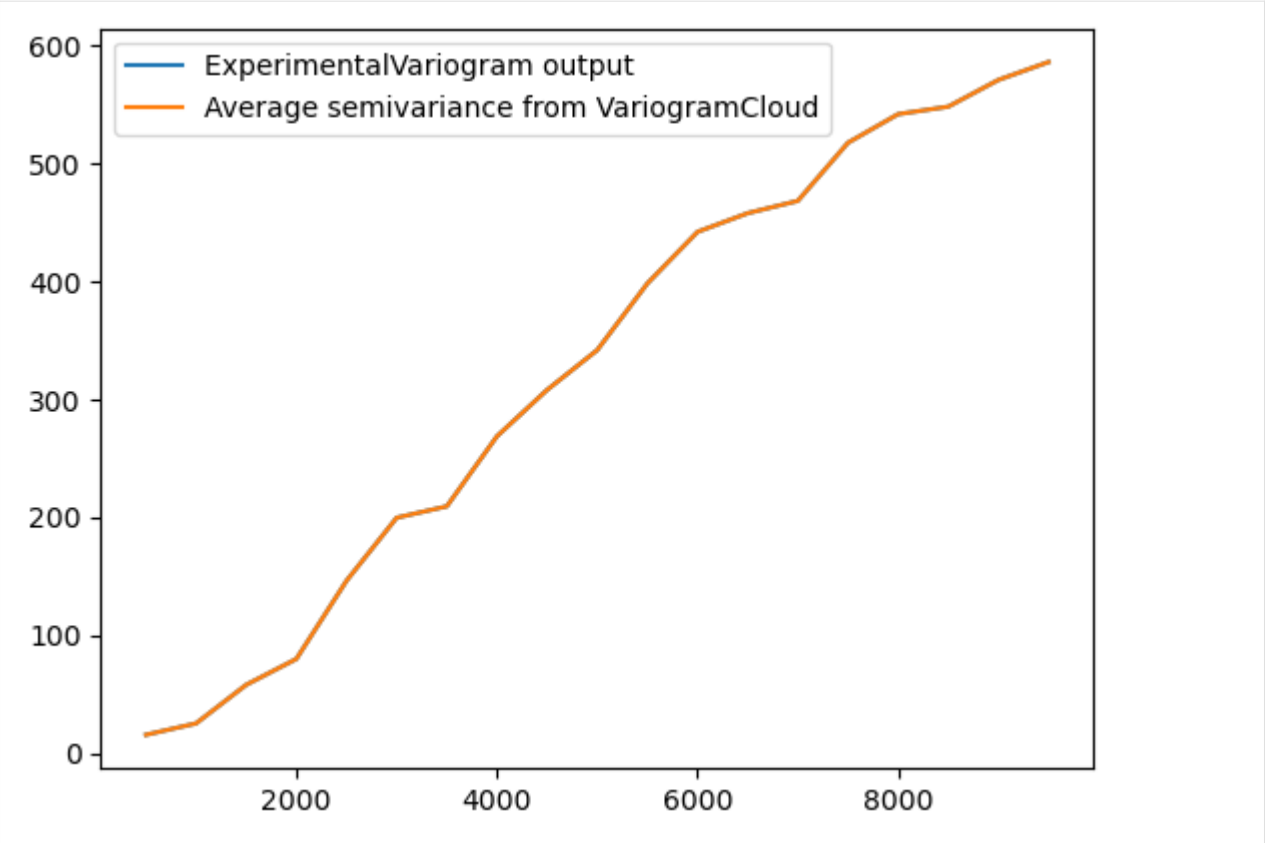
With this detailed table, we can analyze the variogram in detail and decide: - if lags are correctly placed, - if there are enough points per lag, - if the maximum range is too low or too high, - is distribution close to normal,

And based on the answers to those questions, we can change semivariogram parameters slightly.

The interesting row is `avg semivariance`. Let's plot it against lags, and let's plot the output from `ExperimentalVariogram` in the same figure:

[22]:
```
# Re-calculate experimental variogram

max_range = 10000
step_size = 500

experimental_variogram = ExperimentalVariogram(
    input_array=dem,
    step_size=step_size,
    max_range=max_range
)
```

[23]:
```
# Plot

plt.figure()
plt.plot(experimental_variogram.experimental_semivariance_array[:, 0],
         experimental_variogram.experimental_semivariance_array[:, 1])
plt.plot(desc.loc['avg semivariance'])
plt.legend(['ExperimentalVariogram output', 'Average semivariance from VariogramCloud'])
plt.show()
```



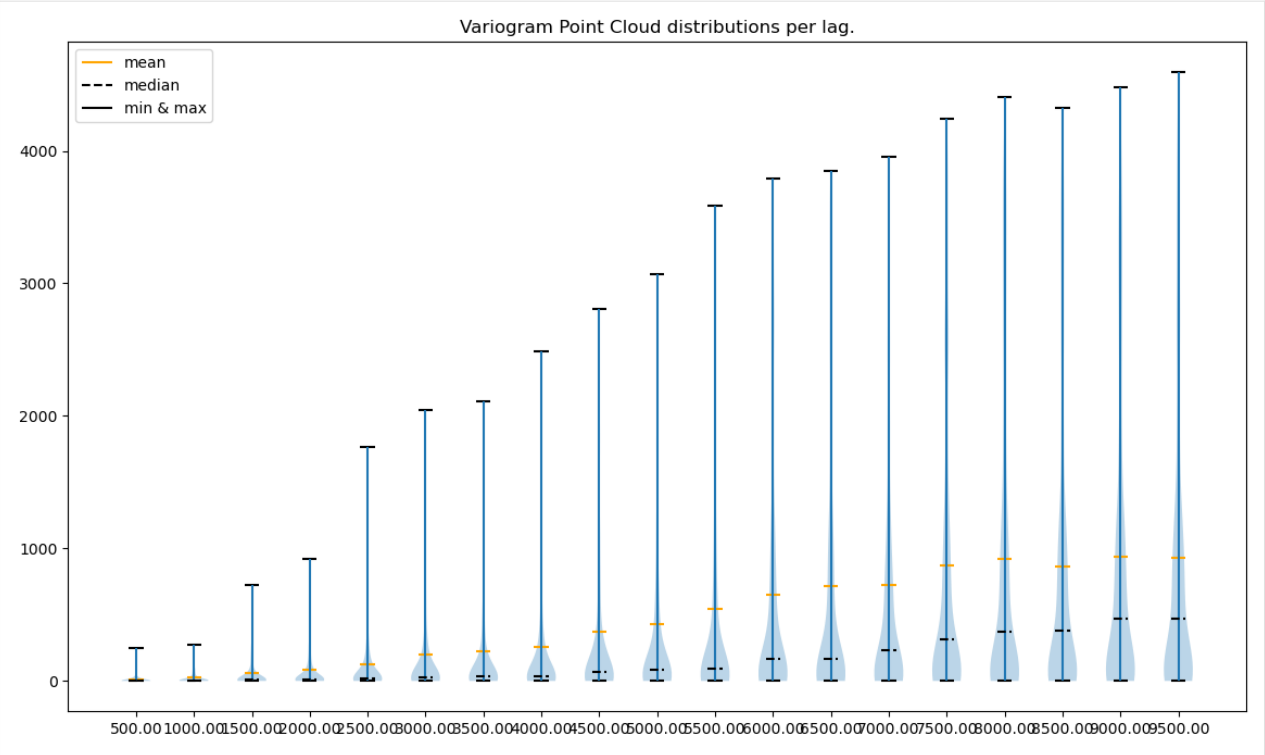
The result is the same, and it shouldn't be a surprise. The experimental variogram averages all semivariances per lag. We stepped back in computations and focused on all point pairs within a lag.

There is one last method within the `VariogramCloud` class: `.remove_outliers()`. It cleans our variogram from anomalous readings. The cleaning default algorithm uses the z-score, but we can also use the interquartile range analysis. Both methods cut outliers from the largest or the lowest intervals. For data that deviates greately from normal distribution, it is better to use an inter-quartile range for cleaning:

[24]:
```
new_variogram_cloud = variogram_cloud.remove_outliers(method='iqr', iqr_lower_limit=3,␣
↪iqr_upper_limit=2)
```

Now we can check what has happend with the variogram:

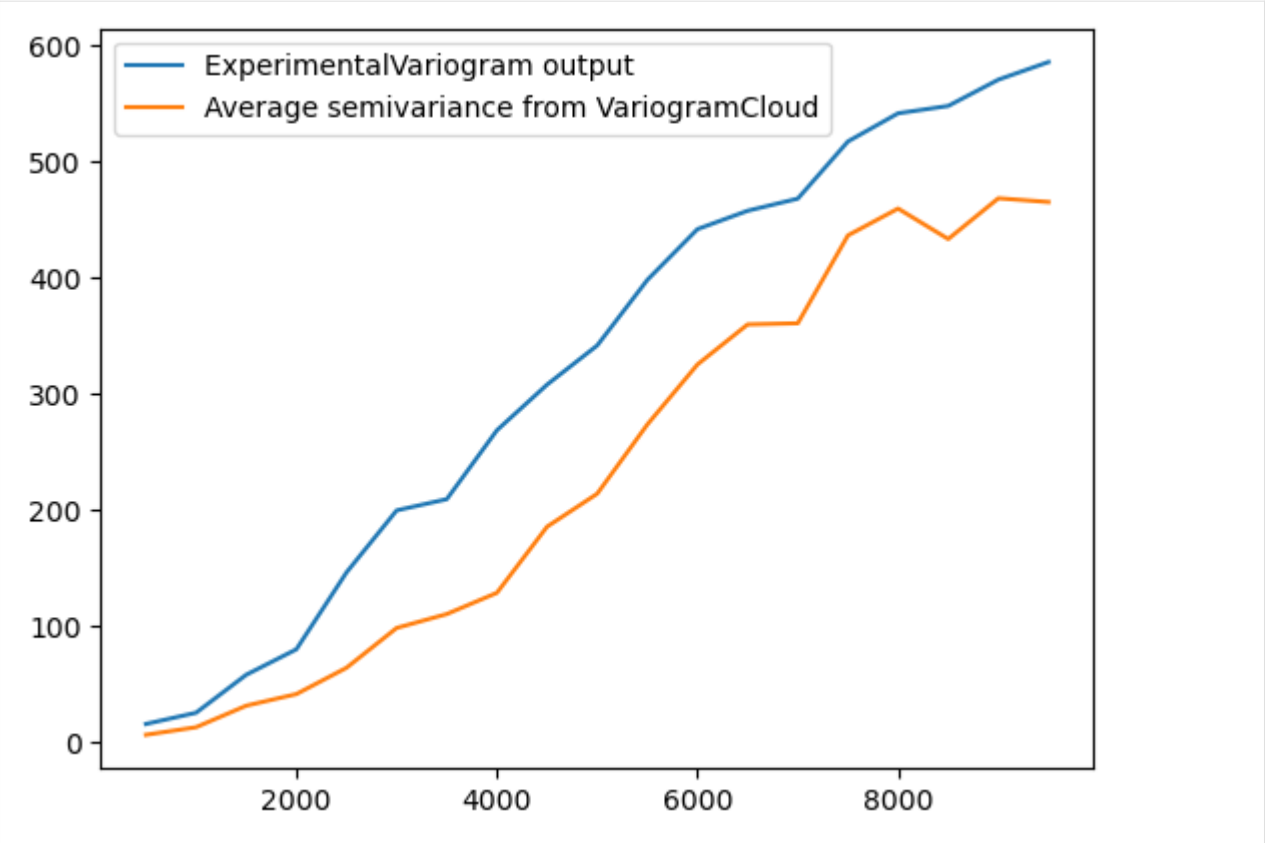
[25]:
```
new_variogram_cloud.plot(kind='violin')
```



[25]:
```
True
```

Maximum semivariances are much lower than before. How does look the averaged semivariogram? Now we will use the `.calculate_experimental_variogram()` method to calculate the variogram directly and compare it to the experimental variogram retrieved from the `ExperimentalVariogram` class.

[26]:
```
# Plot
exp_var_from_point_cloud = new_variogram_cloud.calculate_experimental_variogram()

plt.figure()
plt.plot(experimental_variogram.experimental_semivariance_array[:, 0],
         experimental_variogram.experimental_semivariance_array[:, 1])
plt.plot(exp_var_from_point_cloud[:, 0],
         exp_var_from_point_cloud[:, 1])
plt.legend(['ExperimentalVariogram output', 'Average semivariance from VariogramCloud'])
plt.show()
```



Now we see that the shape of the variogram persisted, but maximum values per lag are lower. It could be helpful for some models, especially if we eliminate outliers with extreme values.

Summary

After all those steps, you have better insights into the `ExperimentalVariogram` and `VariogramCloud` classes. You may check your dataset and compare results to those in the tutorial.

The main takeaways from this tutorial are:

- API of `ExperimentalVariogram` and `VariogramCloud` classes,
- differences and similarities between `ExperimentalVariogram` and `VariogramCloud` classes,
- full variogram analysis and preparation before spatial interpolation.

Where to go from here?

- B.1.1 Ordinary and Simple Kriging
- B.3.1 Directional Ordinary Kriging
- C.1.1 Outliers and Kriging Model

Changelog

Date	Change description	Author
2023-08-22	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2023-01-23	The first release of the tutorial	@SimonMolinsky

[ ]:

2.3.2 Intermediate

B.1.1 Ordinary and Simple Kriging

Table of Contents:

1. Read point data,
2. Set semivariogram model,
3. Set Ordinary Kriging and Simple Kriging models,
4. Predict values at unknown locations.

Introduction

This tutorial will teach us how to perform spatial interpolation with Ordinary and Simple Kriging. We will compare a different number of ranges, and test outcomes of processing with the root mean squared error.

Ordinary and Simple Kriging is the simplest form of Kriging, but they're still powerful techniques.

We use DEM data which is stored in a file `samples/point_data/txt/pl_dem_epsg2180.txt`.

Import packages

[1]:
```
from typing import List

import numpy as np

from pyinterpolate import read_txt, build_experimental_variogram, build_theoretical_
↪variogram, kriging
```

1) Read point data

[2]:
```
dem = read_txt('samples/point_data/txt/pl_dem_epsg2180.txt')
dem[: 3]
```

[2]:
```
array([[2.37685325e+05, 5.45416708e+05, 5.12545509e+01],
       [2.37674140e+05, 5.45209671e+05, 4.89582825e+01],
       [2.37449255e+05, 5.41045935e+05, 1.68178635e+01]])
```

In the beginning, we remove 70 % of our points from the dataset, and we will leave them as a test set to estimate how good our models are.

[3]:
```
def create_train_test(dataset: np.ndarray, training_set_ratio=0.3):
    """
    Function divides base dataset into a training and a test set.

    Parameters
    ----------
    dataset : np.ndarray

    training_set_ratio : float, default = 0.3

    Returns
    -------
    training_set, test_set : List[np.ndarray]
    """

    np.random.seed(101)  # To ensure that we will get the same results every time

    indexes_of_training_set = np.random.choice(range(len(dataset) - 1), int(training_set_
↪ratio * len(dataset)), replace=False)
    training_set = dataset[indexes_of_training_set]
    validation_set = np.delete(dataset, indexes_of_training_set, 0)
    return training_set, validation_set


train_set, test_set = create_train_test(dem)
```

We have removed a subset of points from a dataset to be sure that Kriging is working. In this scenario, 70% of available points are removed, but in real-world cases, you will probably have even fewer points to perform estimations, down to the 1% of known values.

Function `create_train_test()` divides our dataset into two subsets:

- **training set** used for semivariogram model derivation,
- **test set** used for the model error calculation.

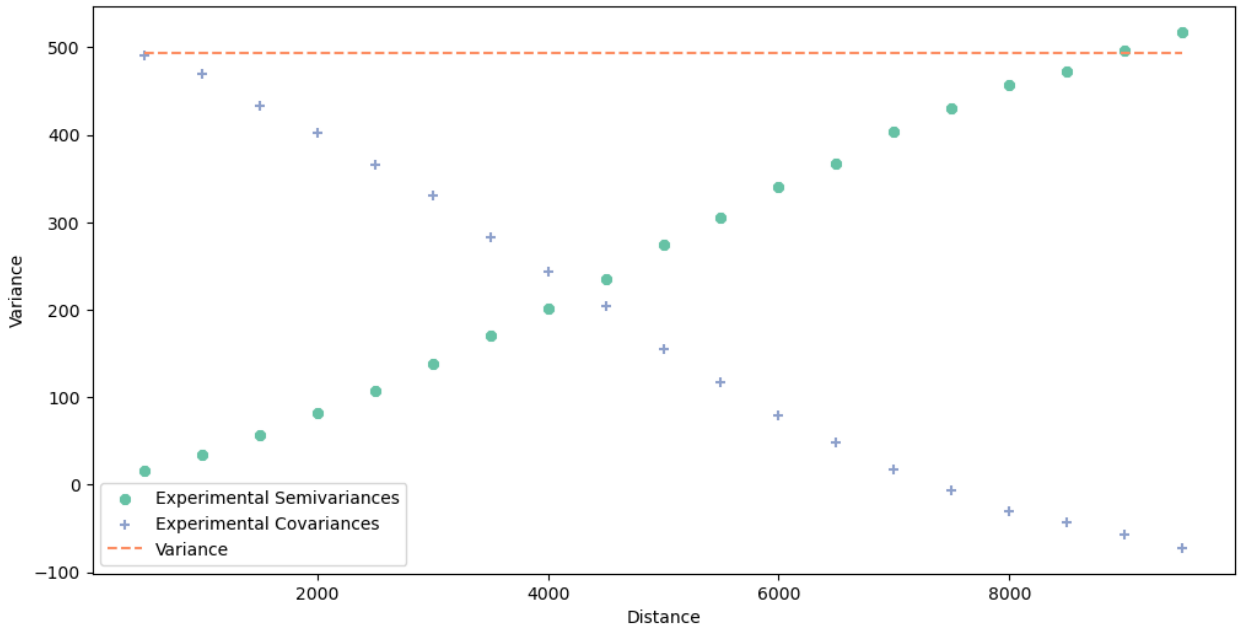
Points for each set are chosen randomly to avoid bias related to the geographical location. Let's imagine we have a sorted list of Digital Elevation Model points. The western part of our measurements covers a mountain, and the eastern part is plain. When we use the part of the west for modeling and the east part for tests, we are going directly into a catastrophe! That's why performing multiple random sampling and testing various realizations from our data is better. We will prepare only one realization with a fixed random seed, but you should be aware that in the real-world analysis, we must test more realizations (e.g., set multiple random seeds).

2) Set Semivariogram model

In this step, we are going to create experimental and theoretical semivariograms.

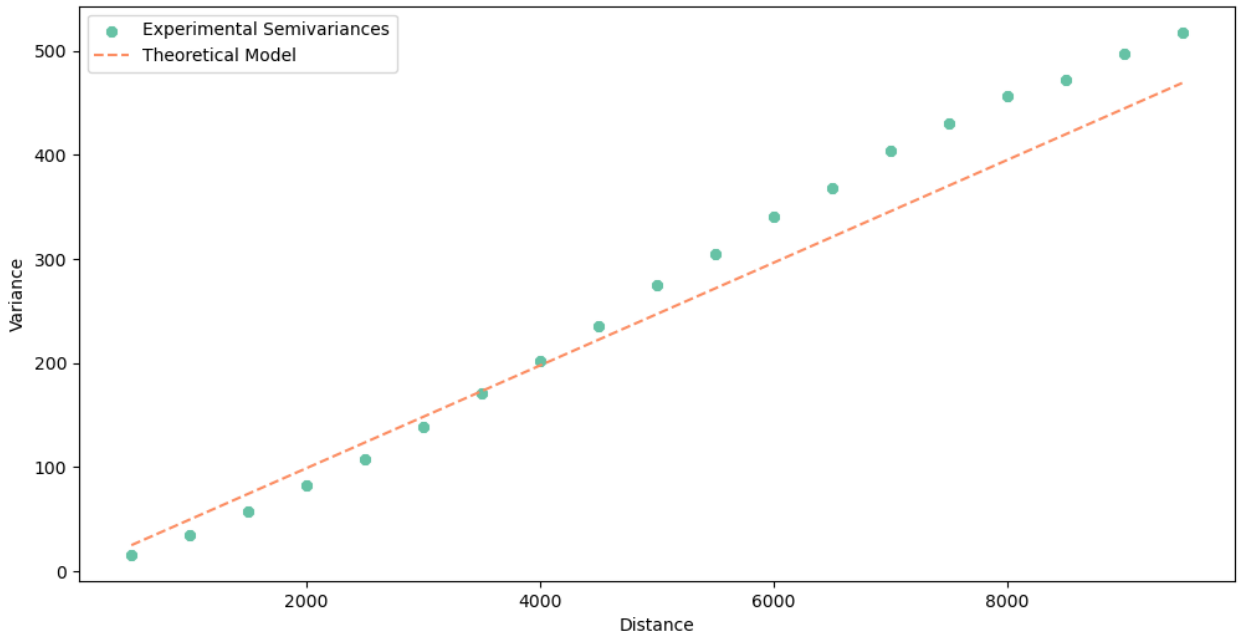
[4]:
```
# Prepare experimental semivariogram

step_radius = 500  # meters
max_range = 10000  # meters

exp_semivar = build_experimental_variogram(input_array=train_set, step_size=step_radius,␣
↪max_range=max_range)
```

[5]:
```
# Plot experimental semivariogram

exp_semivar.plot()
```



[7]:
```
# Fit data into a theoretical model

semivar = build_theoretical_variogram(experimental_variogram=exp_semivar,
                                      model_name='linear',
                                      sill=exp_semivar.variance,
                                      rang=max_range)
```

[8]:
```
semivar.plot()
```



[9]:
```
print(semivar)
```

```
* Selected model: Linear model
* Nugget: 0.0
* Sill: 493.778772754868
* Range: 10000
* Spatial Dependency Strength is Unknown
* Mean Bias: 21.92700357030904
* Mean RMSE: 37.17324824914244
* Error-lag weighting method: None


+--------+--------------------+--------------------+----------------------+
|  lag   |    theoretical     |    experimental    |     bias (y-y')      |
+--------+--------------------+--------------------+----------------------+
| 500.0  |  24.6889386377434  | 15.614084077885668 |  -9.074854559857734  |
| 1000.0 |  49.3778772754868  | 34.151521896711735 | -15.226355378775068  |
| 1500.0 |  74.0668159132302  | 56.61060460335831  |  -17.45621130987189  |
| 2000.0 |  98.7557545509736  | 81.75250857008155  |  -17.00324598089206  |
| 2500.0 |  123.444693188717  | 107.39147411240353 | -16.053219076313468  |
| 3000.0 | 148.1336318264604  | 138.01527174633642 | -10.118360080123978  |
| 3500.0 | 172.82257046420378 | 170.17760461535644 | -2.6449658488473347  |
| 4000.0 | 197.5115091019472  | 201.7643955690108  |  4.2528864670635755  |
| 4500.0 | 222.2004477396906  | 235.6737606045218  |   13.4733128648312   |
| 5000.0 |  246.889386377434  | 274.49115425599933 |  27.60176787856534   |
| 5500.0 | 271.5783250151774  | 304.81328096735956 |  33.23495595218213   |
| 6000.0 | 296.2672636529208  | 341.13970843945583 |  44.87244478653503   |
| 6500.0 | 320.9562022906642  | 367.8150566677254  |  46.85885437706122   |
| 7000.0 | 345.64514092840756 | 403.5663320998641  |  57.92119117145654   |
| 7500.0 |  370.334079566151  | 430.51175648403176 |  60.177676917880774  |
| 8000.0 | 395.0230182038944  | 456.5353159209564  |  61.51229771706198   |
```

(continues on next page)

(continued from previous page)

```
| 8500.0 | 419.7119568416378  | 472.6792052919712  |  52.967248450333386 |
| 9000.0 | 444.4008954793812  | 496.87388696572737 |  52.47299148634619  |
| 9500.0 | 469.08983411712455 | 517.9344861183605  |  48.84465200123594  |
+--------+--------------------+--------------------+---------------------+
```

3) Set Ordinary Kriging and Simple Kriging models

This is the essential step of our tutorial. We've set our semivariogram model and can now predict unknown values. We will "predict" a known and arbitrary point in the first run. It is a test of Kriging, which should work as **an unbiased linear estimator**. Thus it should return the **exact** value if we pass into it a point used for training. In the second step, we will try to guess values at unknown locations and calculate the **R**oot **M**ean **S**quared **E**rror (RMSE) of interpolated values.

We can use the same `kriging()` function for both kriging types. It takes those arguments:

- `observations`: array with known points,
- `theoretical_model`: fitted `TheoreticalVariogram` model,
- `points`: points to interpolate values,
- `how`: `ok` - ordinary kriging, `sk` - simple kriging,
- `neighbors_range`: `None` or `float`, the maximum distance where we search for point neighbors. If `None` is given, the range is selected from the `theoretical_model` `rang` attribute.
- `no_neighbors`: `int`, number of neighbors to estimate unknown value.
- `use_all_neighbors_in_range`: `bool`, default is `False`. `True`: if the number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors`, then take all of them for modeling.
- `sk_mean`: `None` or `float`, the mean value of a process over a study area. It should be known before processing, and that's why Simple Kriging has a limited number of applications. You must have multiple samples and a well-known area to use this parameter.
- `allow_approx_solutions`: Allows the approximation of kriging weights based on the OLS algorithm. Not recommended to set it to `True` if you don't know what you are doing! By default, it is set to `False`,
- `number_of_workers`: if we pass more than 10k points to interpolate, setting this parameter to the number of your CPU workers (or -1) will speed up calculations.

We use only the first four parameters and `sk_mean` when we perform Simple Kriging. Let's start! The first step is an interpolation of the value known by our model.

[10]:
```
# Select one known value

known_value = train_set[10]
known_value
```

[10]:
```
array([2.49304390e+05, 5.40766289e+05, 2.03649998e+01])
```

[11]:
```
# Predict with Ordinary Kriging

ok_interpolation = kriging(train_set, semivar, [known_value[:-1]])
ok_interpolation
```

```
100%|| 1/1 [00:00<00:00, 372.13it/s]
```

[11]:
```
array([[2.03649998e+01, 0.00000000e+00, 2.49304390e+05, 5.40766289e+05]])
```

The first value is our prediction: it is precisely the same as the input in the training set! So far, the algorithm has worked well. The second value is prediction variance error, equal to zero - we are sure it is the exact value.

Simple Kriging is slightly different than Ordinary Kriging, and we must set the process mean to retrieve valid results. It is rarely the case. That's why Ordinary Kriging is the first choice for many applications. We know the global mean because we have the whole dataset, but in the real-world scenario, we cannot divide the set into training and test sets and then get the mean from the entire dataset - it is an information leak from the test set into a model!

[12]:
```
sk_interpolation = kriging(train_set, semivar, [known_value[:-1]], how='sk', sk_
↪mean=float(np.mean(dem)))
sk_interpolation
```

```
100%|| 1/1 [00:00<00:00, 576.93it/s]
```

[12]:
```
array([[2.03649998e+01, 0.00000000e+00, 2.49304390e+05, 5.40766289e+05]])
```

The Simple Kriging algorithm returns the same output as the Ordinary Kriging: `[prediction, error variance, pt x, pt y]`. And as with Ordinary Kriging, Simple Kriging has returned the same value as the actual value fed to the algorithm.

4) Predict values at unknown locations

Using kriging for interpolation of known points values is pointless, and it is an excellent tool for a testing algorithm but nothing more. Here we will interpolate with Kriging, and we will interpolate values at unknown locations. Additionally, we will control the `no_neighbors` parameter to check how it influences predictions.

In the first step, we will create a simple function to test our kriging results and calculate RMSE.

[13]:
```
def test_kriging(train_data, variogram_model, ktype, test_values, number_of_neighbors,␣
↪sk_mean_value=None):
    predictions = kriging(observations=train_data,
                          theoretical_model=variogram_model,
                          points=test_values[:, :-1],
                          how=ktype,
                          no_neighbors=number_of_neighbors,
                          number_of_workers=1,
                          sk_mean=sk_mean_value)
    mse = np.mean((predictions[:, 0] - test_values[:, -1])**2)
    rmse = np.sqrt(mse)
    return rmse
```

[14]:
```
# Number of neighbors

no_of_n = [4, 8, 16, 32, 64, 128, 256]
print('Ordinary Kriging: tests')
print('')

for nn in no_of_n:
    print('Number of neighbors:', nn)
    rmse_pred = test_kriging(train_data=train_set, variogram_model=semivar, ktype='ok',␣
↪test_values=test_set, number_of_neighbors=nn)
```

(continues on next page)

(continued from previous page)

```
    print('RMSE:', rmse_pred)
    print('')
```

```
Ordinary Kriging: tests

Number of neighbors: 4
100%|| 4826/4826 [00:01<00:00, 3356.06it/s]
RMSE: 3.404341396262182

Number of neighbors: 8
100%|| 4826/4826 [00:01<00:00, 3344.89it/s]
RMSE: 3.2909488461696372

Number of neighbors: 16
100%|| 4826/4826 [00:01<00:00, 3243.19it/s]
RMSE: 3.2674275543792795

Number of neighbors: 32
100%|| 4826/4826 [00:19<00:00, 253.42it/s]
RMSE: 3.2608735889738663

Number of neighbors: 64
100%|| 4826/4826 [00:55<00:00, 86.70it/s]
RMSE: 3.2565732386644757

Number of neighbors: 128
100%|| 4826/4826 [01:05<00:00, 73.89it/s]
RMSE: 3.2547868359770336

Number of neighbors: 256
100%|| 4826/4826 [00:42<00:00, 114.38it/s]
RMSE: 3.2550170696805005
```

[15]:
```
print('Simple Kriging: tests')
print('')

sk_mean = np.mean(dem[:, -1])

for nn in no_of_n:
    print('Number of neighbors:', nn)
    rmse_pred = test_kriging(train_data=train_set, variogram_model=semivar, ktype='sk',␣
↪test_values=test_set, number_of_neighbors=nn, sk_mean_value=sk_mean)
    print('RMSE:', rmse_pred)
    print('')
```

```
Simple Kriging: tests

Number of neighbors: 4
100%|| 4826/4826 [00:01<00:00, 3898.96it/s]
RMSE: 8.697161939348147

Number of neighbors: 8
100%|| 4826/4826 [00:01<00:00, 3866.67it/s]
RMSE: 3.650066190084651

Number of neighbors: 16
100%|| 4826/4826 [00:01<00:00, 3833.67it/s]
RMSE: 3.348632949010878

Number of neighbors: 32
100%|| 4826/4826 [00:06<00:00, 729.77it/s]
RMSE: 3.2785837126433313

Number of neighbors: 64
100%|| 4826/4826 [00:10<00:00, 477.52it/s]
RMSE: 3.261651162610457

Number of neighbors: 128
100%|| 4826/4826 [00:09<00:00, 510.67it/s]
RMSE: 3.255919635781099

Number of neighbors: 256
100%|| 4826/4826 [00:26<00:00, 185.51it/s]
RMSE: 3.254332343473019
```

Usually, Simple Kriging will give us worse results than Ordinary Kriging because we need to know the process mean to build a valid Simple Kriging model. Only if we know the process mean we can use Simple Kriging with a large number of neighbors. We see that the results of Simple Kriging are better when we pass more values into it. You shouldn't be shocked. Simple Kriging *discovers* the global mean and utilizes this information when building a Kriging system.

On the other hand, Ordinary Kriging is a Swiss knife in our geostatistical toolbox. It works well even if we don't know the process mean, and it should be our first-choice technique.

Let's look into processing times with a growing number of neighbors. We can theoretically take more and more neighbors, but is it worth it?

- **Small dataset**: with small datasets (up to 1000 points), we may consider using many neighbors. It shouldn't be a problem for processing time.
- **Large dataset**: more than 5000 points? Then we must take into account two scenarios:

1. A single report or only a one-time analysis - then use as many neighbors as it is possible (within range);
2. Near-real time and systematic analyses - use fewer neighbors, between 8 - 128. I usually use 32 neighbors for datasets with more than 1000 point pairs.

Where to go from here?

- B.1.2 Kriging Benchmarking
- B.1.3 Outliers and Kriging Model
- B.2.1 Directional Ordinary Kriging
- C.1.1 Blocks to Points Interpolation with Ordinary Kriging

Changelog

Date	Change description	Author
2023-08-22	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@Simon-Molinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@Simon-Molinsky
2022-08-20	The tutorial is updated to the version 0.3.0 of the package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class & error variance estimated in a correct way	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-12-11	Behavior of `prepare_kriging_data()` function has benn changed	@Simon-Molinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@Simon-Molinsky
2021-04-03	Simple Kriging `global_mean` parameter update.	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated point data source.	@Simon-Molinsky

[ ]:

B.1.2 Kriging Benchmarking

Table of Contents:

1. Read point data,
2. Divide dataset into two sets: modeling and validation set,
3. Perform IDW and evaluate it,
4. Perform variogram modeling on the modeling set,
5. Validate Kriging and compare Kriging and IDW validation results,

6. Bonus scenario: only 2% of values are known!

Introduction

In this tutorial, we will learn how to validate our Kriging model. We'll compare it to the **Inverse Distance Weighting** function, where the unknown point value is interpolated as the weighted mean of its neighbors. Weights are inversely proportional to the distance between neighboring points, dropping faster with a higher power.

(1) **GENERAL FORM OF IDW**

$$z(u) = \frac{\sum_i \lambda_i * z_i}{\sum_i \lambda_i}$$

Where:

- $z(u)$: is the value at an unknown location,
- i: is an i-th known location,
- z_i: is a value at known location i,
- λ_i: is a weight assigned to the known location i.

(2) **WEIGHTING PARAMETER**

$$\lambda_i = \frac{1}{d_i^p}$$

Where:

- d: is a distance from the known point z_i to the unknown point $z(u)$,
- p: is a hyperparameter that controls how strong a relationship is between a known and unknown point. You may set large p to establish a strong relationship between the closest points and the weak influence of distant points. On the other hand, you may set a small p to emphasize that the distance between points matters a little.

As you noticed **IDW** is a simple but powerful technique. Unfortunately, it has a significant drawback: **we must set ``p`` - power - manually**, which isn't derived from the data and variogram. That's why it can be used for other tasks. An example is to use IDW as a baseline for comparison to the different techniques.

Import packages

[1]:
```
import numpy as np

from pyinterpolate import inverse_distance_weighting  # function for idw
from pyinterpolate import read_txt
from pyinterpolate import build_experimental_variogram  # experimental semivariogram
from pyinterpolate import build_theoretical_variogram, TheoreticalVariogram  #␣
↪theoretical models
from pyinterpolate import kriging  # kriging models
```

1) Read point data

[4]:
```
dem = read_txt('samples/point_data/txt/pl_dem_epsg2180.txt')
dem[:3]
```

[4]:
```
array([[2.37685325e+05, 5.45416708e+05, 5.12545509e+01],
       [2.37674140e+05, 5.45209671e+05, 4.89582825e+01],
       [2.37449255e+05, 5.41045935e+05, 1.68178635e+01]])
```

2) Divide the dataset into two sets: modeling and validation set

In this step, we will divide our dataset into two sets:

- modeling set (10%): points used for variogram modeling,
- validation set (90%): points used for prediction and results validation.

The baseline dataset will be divided randomly.

[5]:
```
# Create modeling and validation sets

def create_model_validation_sets(dataset: np.array, training_set_ratio=0.5, rnd_
↪seed=101):

    np.random.seed(rnd_seed)

    indexes_of_training_set = np.random.choice(range(len(dataset) - 1), int(training_set_
↪ratio * len(dataset)), replace=False)
    training_set = dataset[indexes_of_training_set]
    validation_set = np.delete(dataset, indexes_of_training_set, 0)
    return training_set, validation_set

known_points, unknown_points = create_model_validation_sets(dem)
```

3) Perform IDW and evaluate it

Inverse Distance Weighting doesn't require variogram modeling or other steps. We pass power that affects weights in a denominator. Things to remember are:

- Large `power` -> closer neighbors are more important,
- `power` which is close to the **zero** -> all neighbors are important, and we assume that the distant process has the same effect on our variable as the closest events.

[6]:
```
IDW_POWER = 2
NUMBER_OF_NEIGHBOURS = 128

idw_predictions = []

for pt in unknown_points:
    idw_result = inverse_distance_weighting(known_points, pt[:-1], NUMBER_OF_NEIGHBOURS,␣
↪IDW_POWER)
    idw_predictions.append(idw_result)
```

[7]:
```
# Evaluation

idw_rmse = np.mean(np.sqrt((unknown_points[:, -1] - np.array(idw_predictions))**2))
print(f'Root Mean Squared Error of prediction with IDW is {idw_rmse}')
```

```
Root Mean Squared Error of prediction with IDW is 2.5045203528088513
```

Clarification: Obtained Root Mean Squared Error could serve as a baseline for further model development. To build a better reference, we create four IDW models of powers:

1. **0.5**
2. **1**
3. **2**
4. **4**

[8]:
```
IDW_POWERS = [0.5, 1, 2, 4]
idw_rmse = {}

for pw in IDW_POWERS:
    results = []
    for pt in unknown_points:
        idw_result = inverse_distance_weighting(known_points, pt[:-1], NUMBER_OF_
↪NEIGHBOURS, pw)
        results.append(idw_result)
    idw_rmse[pw] = np.mean(np.sqrt((unknown_points[:, -1] - np.array(results))**2))
```

[9]:
```
for pw in IDW_POWERS:
    print(f'Root Mean Squared Error of prediction with IDW of power {pw} is {idw_
↪rmse[pw]:.4f}')
```

```
Root Mean Squared Error of prediction with IDW of power 0.5 is 4.4434
Root Mean Squared Error of prediction with IDW of power 1 is 3.8513
Root Mean Squared Error of prediction with IDW of power 2 is 2.5045
Root Mean Squared Error of prediction with IDW of power 4 is 1.8198
```

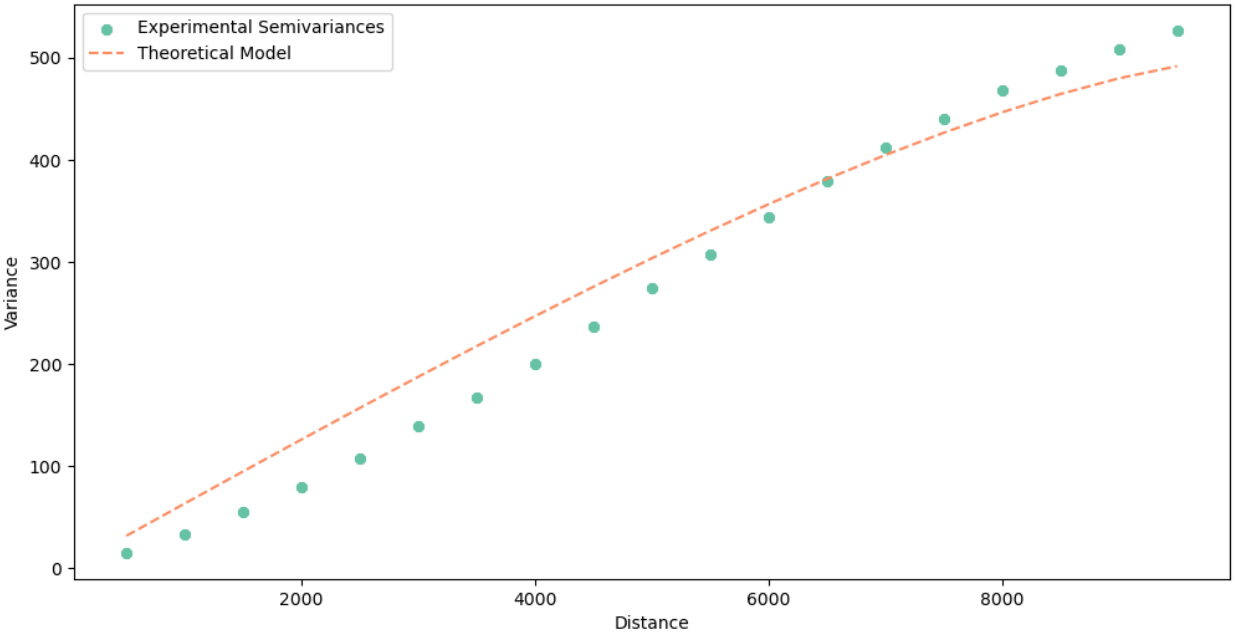
4) Perform variogram modeling on the modeling set

In this step, we will go through semivariogram modeling for Kriging interpolation.

[10]:
```
max_range = 10000
step_size = 500

exp_semivar = build_experimental_variogram(known_points, step_size=step_size, max_
↪range=max_range)
```

[11]:
```
semivar = build_theoretical_variogram(exp_semivar, model_name='circular', sill=exp_
↪semivar.variance, rang=max_range)
```

[12]:
```
semivar.plot()
```



5) Validate Kriging and compare Kriging and IDW validation results

Lastly, we perform Kriging interpolation and compare results to the **IDW** models. We use all points to weight values at unknown locations and the semivariogram model we chose in the previous step.

[13]:
```
# Set Kriging model

predictions = kriging(observations=known_points, theoretical_model=semivar,␣
↪points=unknown_points[:, :-1], no_neighbors=128)
```

```
100%|| 3447/3447 [01:17<00:00, 44.31it/s]
```

[14]:
```
# Evaluation

kriging_rmse = np.mean(np.sqrt((unknown_points[:, -1] - np.array(predictions[:, 0]))**2))
print(f'Root Mean Squared Error of prediction with Kriging is {kriging_rmse}')
```

```
Root Mean Squared Error of prediction with Kriging is 1.6638061333290732
```

Clarification: Kriging is better than any of the IDW models, and we may assume that our modeling approach gives us better insights into the spatial process we are observing. But this is not the end. Let's consider a more complex scenario!

6) Bonus scenario: only 2% of values are known!

Data sampled from the real world is less good than the sample from the tutorial. It is too expensive to densely sample every location, and usually, we get only a tiny percent of the area covered by data. That's why it is good to compare **IDW vs. Kriging** in this scenario! We repeat steps 1-5 with a change in the division for the modeling/validation set. (I encourage you to try it alone and compare your code and results with those in this notebook).

[15]:
```
# Data preparation

known_points, unknown_points = create_model_validation_sets(dem, 0.02, rnd_seed=112)
```

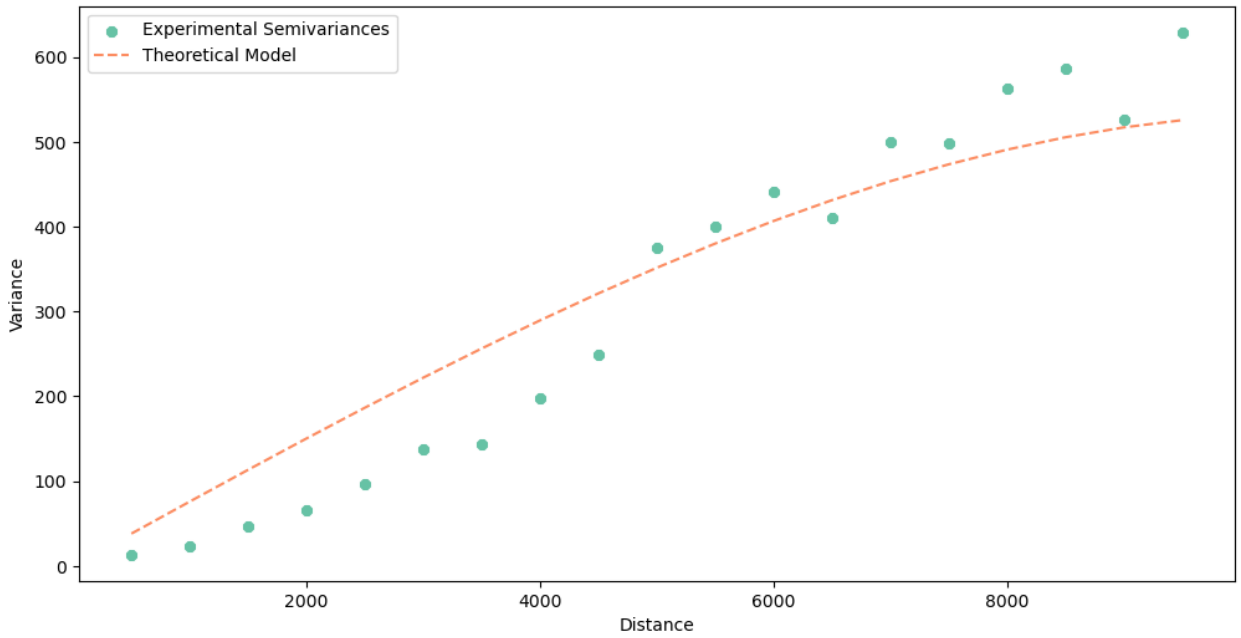
[16]:
```
IDW_POWERS = [0.5, 1, 2, 4, 6, 8]
idw_rmse = {}

for pw in IDW_POWERS:
    results = []
    for pt in unknown_points:
        idw_result = inverse_distance_weighting(known_points, pt[:-1], NUMBER_OF_
↪NEIGHBOURS, pw)
        results.append(idw_result)
    idw_rmse[pw] = np.mean(np.sqrt((unknown_points[:, -1] - np.array(results))**2))
```

[17]:
```
for pw in IDW_POWERS:
    print(f'Root Mean Squared Error of prediction with IDW of power {pw} is {idw_
↪rmse[pw]:.4f}')
```

```
Root Mean Squared Error of prediction with IDW of power 0.5 is 17.1027
Root Mean Squared Error of prediction with IDW of power 1 is 13.8879
Root Mean Squared Error of prediction with IDW of power 2 is 7.1886
Root Mean Squared Error of prediction with IDW of power 4 is 5.1183
Root Mean Squared Error of prediction with IDW of power 6 is 5.2371
Root Mean Squared Error of prediction with IDW of power 8 is 5.3415
```

[18]:
```
# Variogram

max_range = 10000
step_size = 500

exp_semivar = build_experimental_variogram(known_points, step_size=step_size, max_
↪range=max_range)
```

[19]:
```
semivar = TheoreticalVariogram()
semivar.autofit(exp_semivar, model_name='spherical')
semivar.plot()
```



[20]:
```
# Set Kriging model

predictions = kriging(observations=known_points, theoretical_model=semivar,␣
↪points=unknown_points[:, :-1], no_neighbors=128, use_all_neighbors_in_range=False)
```

```
100%|| 6756/6756 [00:58<00:00, 115.93it/s]
```

[21]:
```
# Evaluation

kriging_rmse = np.mean(np.sqrt((unknown_points[:, -1] - np.array(predictions[:, 0]))**2))
print(f'Root Mean Squared Error of prediction with Kriging is {kriging_rmse}')
```

```
Root Mean Squared Error of prediction with Kriging is 4.886647202272596
```

[22]:
```
# Comparison

for pw in IDW_POWERS:
    print(f'Root Mean Squared Error of prediction with IDW of power {pw} is {idw_
↪rmse[pw]:.4f}')
print(f'Root Mean Squared Error of prediction with Kriging is {kriging_rmse:.4f}')
```

```
Root Mean Squared Error of prediction with IDW of power 0.5 is 17.1027
Root Mean Squared Error of prediction with IDW of power 1 is 13.8879
Root Mean Squared Error of prediction with IDW of power 2 is 7.1886
Root Mean Squared Error of prediction with IDW of power 4 is 5.1183
Root Mean Squared Error of prediction with IDW of power 6 is 5.2371
Root Mean Squared Error of prediction with IDW of power 8 is 5.3415
Root Mean Squared Error of prediction with Kriging is 4.8866
```

Your results may be different (if you set another random seed), but in most cases, Kriging will be better than IDW or very close to the best results from IDW. Even more important is that for the single data source with a low number of samples, we don't have the opportunity to perform the validation step, and we're unable to guess how big the power parameter should be. With Kriging, we model variogram, and *voila!* - model works.

Where to go from here?

- B.1.3 Outliers and Kriging Model
- B.2.1 Directional Ordinary Kriging
- C.1.1 Blocks to Points Interpolation with Ordinary Kriging
- C.1.2 Semivariogram Regularization

Changelog

Date	Change description	Author
2023-08-23	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@Simon-Molinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@Simon-Molinsky
2022-08-23	The new version of tutorial for the 0.3.0 version of a package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-12-08	Behavior of `prepare_kriging_data()` function has changed	@Simon-Molinsky
2021-05-12	First version of tutorial	@Simon-Molinsky

[ ]:

B.1.3 Outliers and Kriging

Table of Contents:

1. Read point data and take 10% of it as a sample for the further analysis (dataset A),
2. Check if outliers are present in a data and create additional dataset without outliers (dataset B),
3. Create the Variogram Point Cloud model for the datasets A and B,
4. Remove outliers from datasets A and B,
5. Create four Ordinary Kriging models and compare their performance.

Introduction

Outliers may affect our analysis and the final interpolation results. In this tutorial, we learn about their influence on the final model and compare interpolation errors for different scenarios where data is cleaned differently.

We can remove too high or too low values at the preprocessing stage (check part 2 of the tutorial) or remove outliers directly from the variogram point cloud (part 4). Results from each type of preprocessing (and a raw dataset analysis) are different, and we will compare them.

We use:

- DEM data, which is stored in a file `samples/point_data/txt/pl_dem_epsg2180.txt`.

Import packages

[2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from pyinterpolate import calc_point_to_point_distance, read_txt, kriging,␣
↪TheoreticalVariogram, VariogramCloud
```

1) Read point data and divide it into the training and test set

[3]:
```
# Read data from file

dem = read_txt('samples/point_data/txt/pl_dem_epsg2180.txt')
```

[4]:
```
# Divide data into training and test set

def create_train_test(data, training_fraction):
    idxs = np.arange(0, len(data))
    number_of_training_samples = int(len(data) * training_fraction)
    training_idxs = np.random.choice(idxs, size=number_of_training_samples,␣
↪replace=False)
    test_idxs = [i for i in idxs if i not in training_idxs]
    training_set = data[training_idxs, :]
    test_set = data[test_idxs, :]

    return training_set, test_set
```

[5]:
```
train, test = create_train_test(dem, 0.1)
```

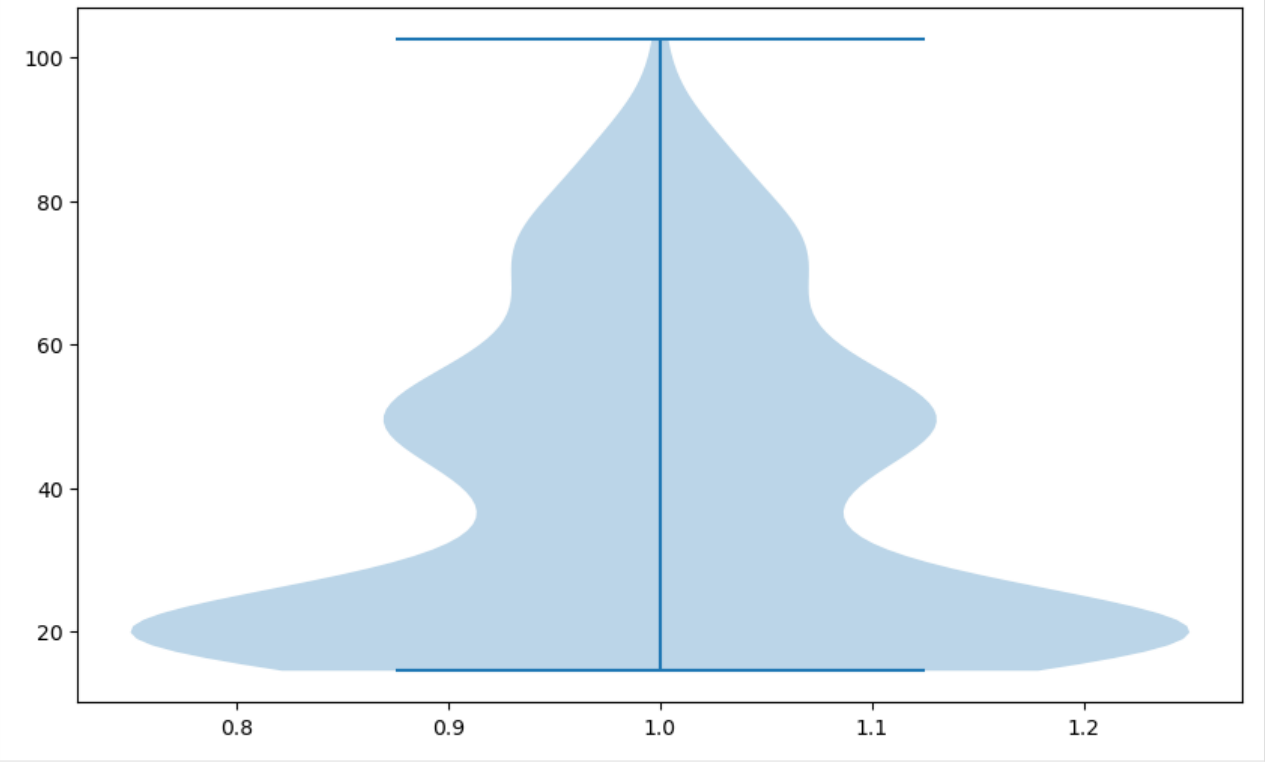
[6]:
```
train[:3]
```

[6]:
```
array([[2.51238103e+05, 5.50262413e+05, 8.71362305e+01],
       [2.45382683e+05, 5.28767297e+05, 5.84672661e+01],
       [2.54519766e+05, 5.51709669e+05, 7.63842850e+01]])
```

2) Check outliers: analyze the distribution of the values

We will inspect all values in the `train` set to determine if our dataset contains outliers. In the beginning, we plot data distribution with the `violinplot`.

[7]:
```
# Distribution plot

plt.figure(figsize=(10, 6))
plt.violinplot(train[:, -1])
plt.show()
```



> **NOTE:** Your plot may differ from the tutorial's. Why is that? Because we take a random sample of 10% of values, and after each iteration, the algorithm takes different points for the analysis.

Clarification:

Investigation of the plot tells us that our data is:

- grouped around the lowest values, and most of the values are below 50 meters,
- has (probably) three different distributions mixed, which can be a sign that the Digital Elevation Model covers three different types of elevation. One is grouped around 20 meters, the next roughly 50 meters and the furthest is visible around 70 meters.

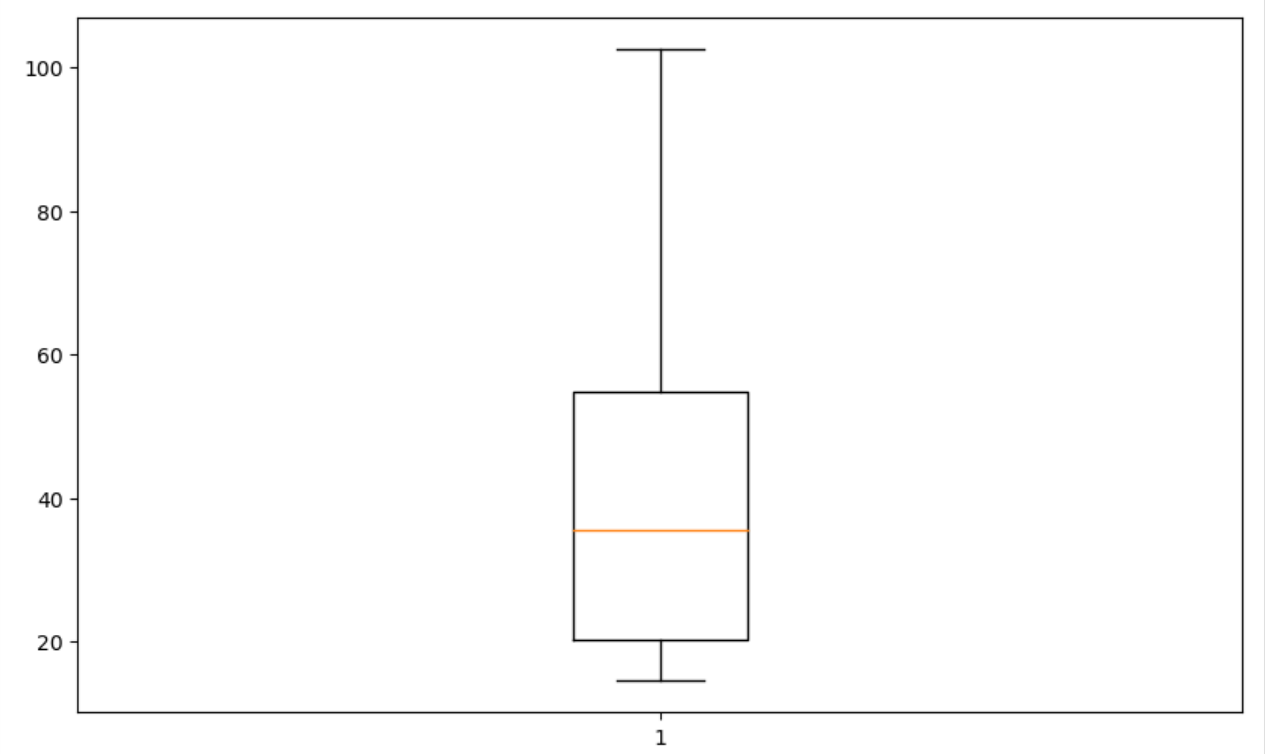
Violinplot is suitable for the distribution analysis. Finding outliers in this plot may be challenging because we see mostly distributions. We should change a plot type to understand if outliers exist in a dataset. The excellent choice is the `boxplot`:

[8]:
```
# Boxplot

plt.figure(figsize=(10, 6))
```

(continues on next page)

(continued from previous page)

```
plt.boxplot(train[:, -1])
plt.show()
```



Boxplot is a unique and handy data visualization tool. Let's analyze this plot from the bottom up to the top.

> **NOTE:** Boxplot represents values sorted in ascending order and their statistical properties: quartiles, median, and outliers.

- The bottom whisker (horizontal line) represents the lower range of values in our dataset,
- The box lower line is the first quartile of data or, in other words, 25% of values of our dataset are below this point. We name it Q1.
- The middle line is the median of our dataset, and we name it Q2 or median.
- The upper line is the third quartile of a data or, in other words, 75% of values are below this point,
- The top whisker represents the upper range of values in our dataset. We name it Q3.
- Individual points (if they exist, then we see them as points below the bottom whisker or above the top whisker) are considered outliers. They could be outliers in the upper range as well as the lower range of our data. The significant distance between Q1 and the bottom whisker or between Q3 and the top whisker indicates potential outliers. Points below or above this distance are treated as outliers. The outlier distance is calculated as the $weight * (Q3 - Q1)$ where we can manually set the `weight`, but other parameters are read directly from the data.

We use this knowledge to remove outliers from the dataset, assuming that *outliers are anomalies rather than unbiased readings*. We will remove outliers from the data, assuming that outliers are placed one standard deviation from Q1 (down) and Q3 (up).

[9]:
```
# Create training set without outliers
```

(continues on next page)

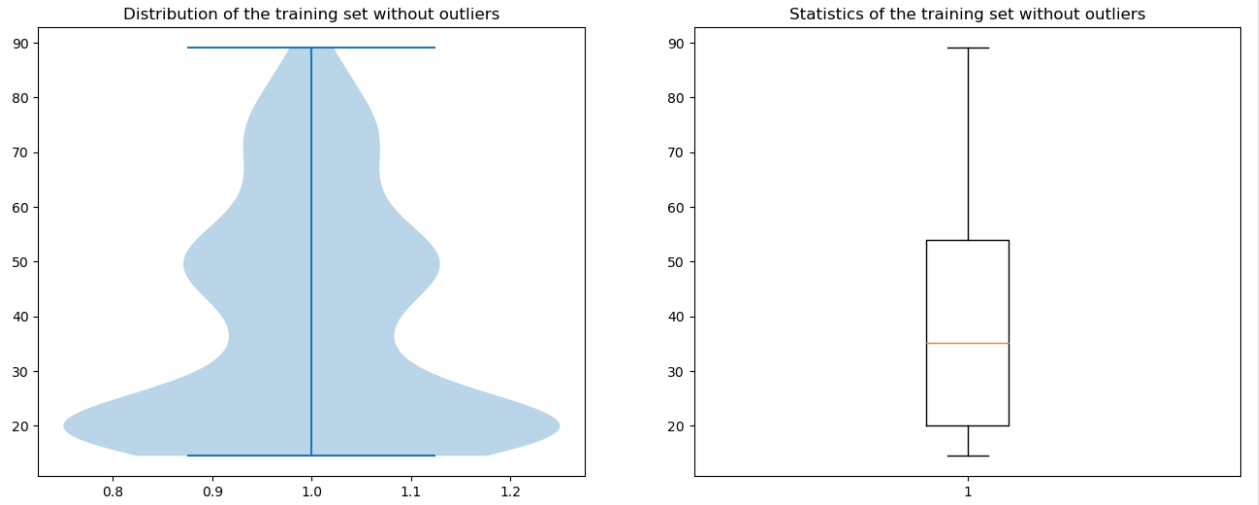
(continued from previous page)

```
q1 = np.quantile(train[:, -1], 0.25)
q3 = np.quantile(train[:, -1], 0.75)

top_limit = q3 + (q3 - q1)

train_without_outliers = train[train[:, -1] < top_limit]
```

[10]:
```
print('Length of the full training set is {} records'.format(len(train)))
print('Length of the pre-processed training set is {} records'.format(len(train_without_
↪outliers)))
```

```
Length of the full training set is 689 records
Length of the pre-processed training set is 680 records
```

[11]:
```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(16, 6))
ax[0].violinplot(train_without_outliers[:, -1])
ax[0].set_title('Distribution of the training set without outliers')
ax[1].boxplot(train_without_outliers[:, -1])
ax[1].set_title('Statistics of the training set without outliers')
plt.show()
```



Clarification: We have cut some records from the baseline training dataset. The distribution plot (`violinplot`) has a shorter tail and ends more abruptly upwards. The `boxplot` of the new data doesn't have any outliers. The one crucial thing to notice is that the observations are still skewed, but this is not a problem for this concrete tutorial.

> **NOTE**: if you are eager to know how to deal with skewed datasets, we recommend the article Transforming Skewed Data.

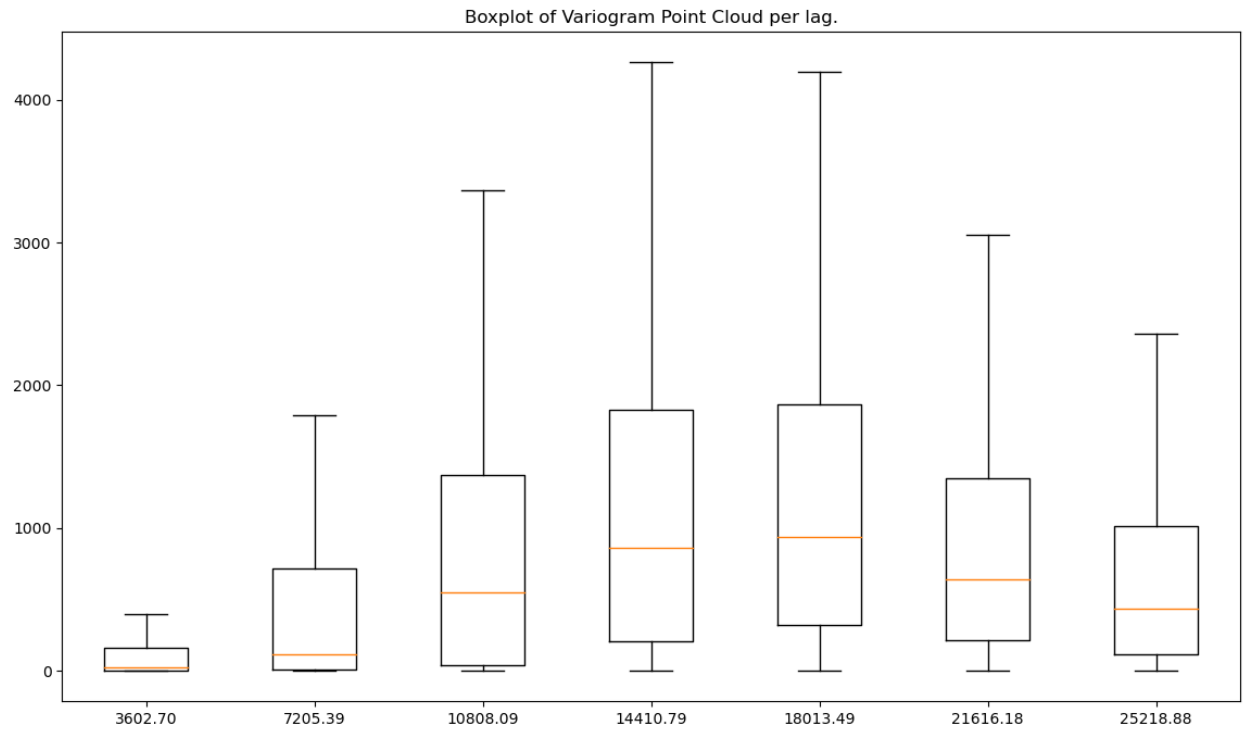
3) Create the Variogram Point Cloud model for datasets A and B

Now, we are going one step further, and we will transform both datasets with- and without- outliers and calculate variogram point clouds. Then, we compare both clouds.

[12]:
```
def get_variogram_point_cloud(dataset, max_range, number_of_lags=8):
    step_size = max_range / number_of_lags
    cloud = VariogramCloud(input_array=dataset, step_size=step_size, max_range=max_range,
↪ calculate_on_creation=True)
    return cloud
```

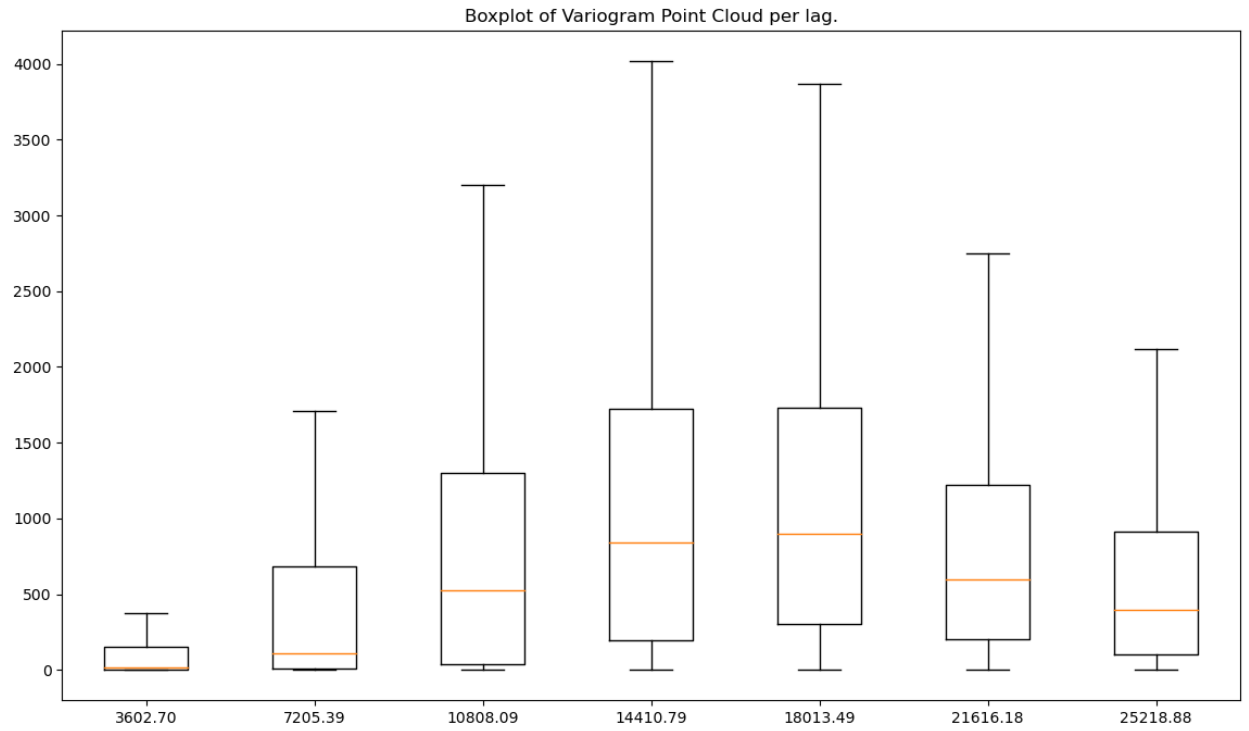
[13]:
```
whole_distance = np.max(calc_point_to_point_distance(train[:, :-1]))

cloud_raw = get_variogram_point_cloud(train, whole_distance)
cloud_processed = get_variogram_point_cloud(train_without_outliers, whole_distance)
```

[14]:
```
# Show variogram cloud: initial training dataset

cloud_raw.plot('box')
```



[14]:
```
True
```

[15]:
```
# Show variogram cloud: pre-processed training dataset

cloud_processed.plot('box')
```



[15]:
```
True
```

Clarification: a quick look into the results shows that calculated semivariances are skewed into significant positive values for each lag, but most profoundly within middle lags (~15:21 km). The processed dataset has lower semivariances than the raw readings, and both variograms have a similar shape.

In the next cell, we will check a standard deviation of the per lag variances.

[16]:
```
for k, v in cloud_raw.experimental_point_cloud.items():
    print('Lag {:.2f}'.format(k))

    v_raw = int(np.std(v))
    v_pro = int(np.std(cloud_processed.experimental_point_cloud[k]))
    v_smape = 100 * (np.abs(v_raw - v_pro) / (0.5 * (v_raw + v_pro)))

    print('Standard Deviation raw dataset:', v_raw)
    print('Standard Deviation processed dataset:', v_pro)
    print('Symmetric Mean Absolute Percentage Error of Variances: {:.2f}'.format(v_
↪smape))
    print('')
```

```
Lag 3602.70
Standard Deviation raw dataset: 562
Standard Deviation processed dataset: 559
Symmetric Mean Absolute Percentage Error of Variances: 0.54

Lag 7205.39
Standard Deviation raw dataset: 1078
Standard Deviation processed dataset: 1005
Symmetric Mean Absolute Percentage Error of Variances: 7.01
```

(continues on next page)

(continued from previous page)

```
Lag 10808.09
Standard Deviation raw dataset: 1298
Standard Deviation processed dataset: 1185
Symmetric Mean Absolute Percentage Error of Variances: 9.10

Lag 14410.79
Standard Deviation raw dataset: 1345
Standard Deviation processed dataset: 1204
Symmetric Mean Absolute Percentage Error of Variances: 11.06

Lag 18013.49
Standard Deviation raw dataset: 1269
Standard Deviation processed dataset: 1136
Symmetric Mean Absolute Percentage Error of Variances: 11.06

Lag 21616.18
Standard Deviation raw dataset: 974
Standard Deviation processed dataset: 913
Symmetric Mean Absolute Percentage Error of Variances: 6.47

Lag 25218.88
Standard Deviation raw dataset: 747
Standard Deviation processed dataset: 712
Symmetric Mean Absolute Percentage Error of Variances: 4.80
```

Clarification: The differences (sMAPE) per lag vary greatly. We can see that the preprocessing of raw values introduces the information lost, and it is especially painful for the closest neighbors. It doesn't mean that the preprocessing of raw observations is not recommended, but it is a good idea to include the **spatial component** in the outliers' detection process. By **spatial component**, we mean working with variograms instead of the raw points.

The raw data cleaning has lowered the semivariances dispersion for the middle lags (where we have the largest number of point pairs for the analysis).

At this point, we cannot judge which dataset is better for the modeling. Instead, we will remove outliers from both **variograms** (instead of the **raw data**).

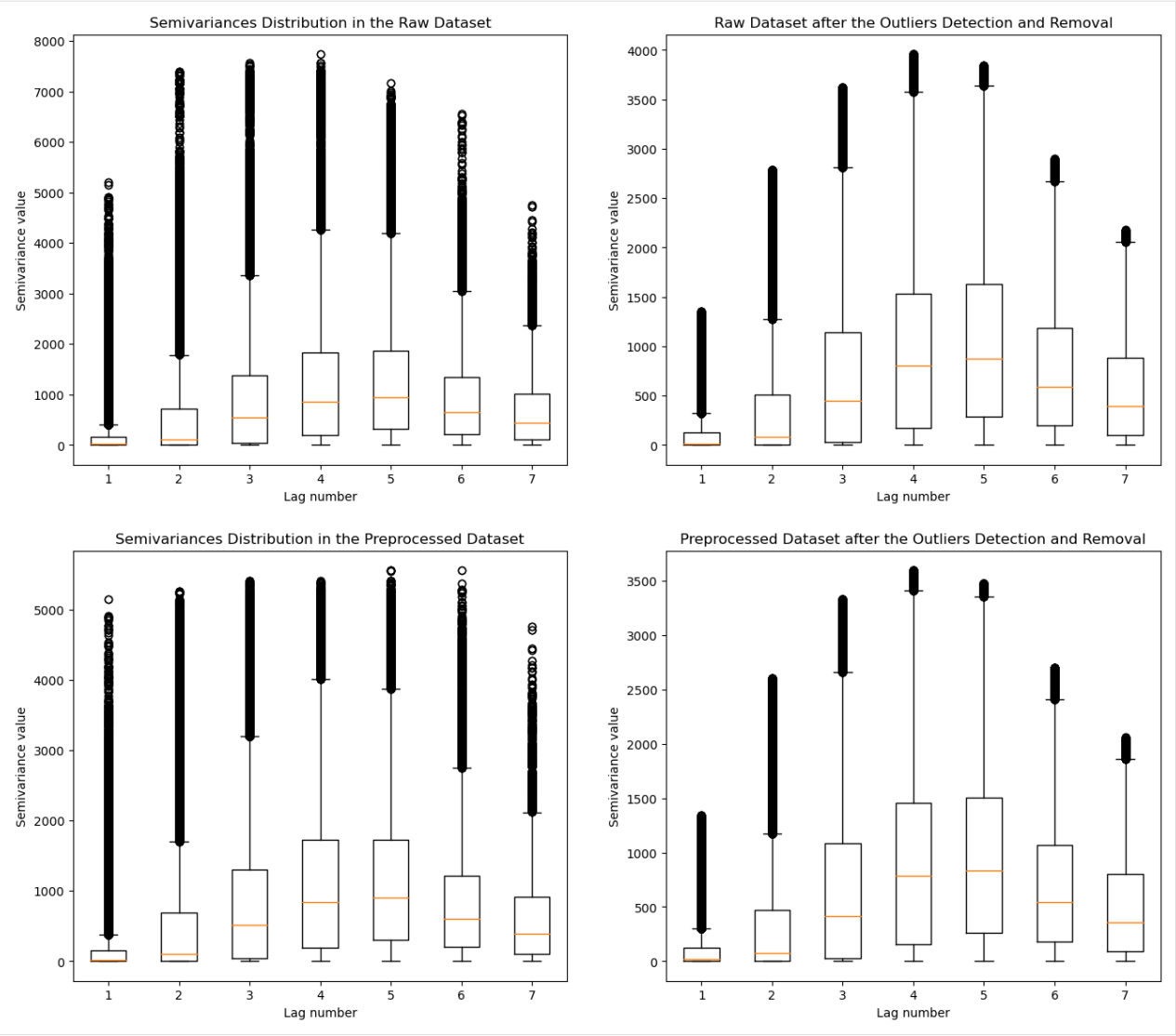
4) Remove outliers from the variograms

In this step, we will use **pyinterpolate's** function `remove_outliers()` to build two additional variogram point clouds from the raw and processed datasets. We delete the top part outliers of the **semivariance values** rather than the raw readings.

[17]:
```
raw_without_outliers = cloud_raw.remove_outliers(method='zscore', z_lower_limit=-2, z_
↪upper_limit=2, inplace=False)
prep_without_outliers = cloud_processed.remove_outliers(method='zscore', z_lower_limit=-
↪2, z_upper_limit=2, inplace=False)
```

[18]:
```
data_raw = [x for x in cloud_raw.experimental_point_cloud.values()]
data_raw_not_out = [x for x in raw_without_outliers.experimental_point_cloud.values()]
data_prep = [x for x in cloud_processed.experimental_point_cloud.values()]
data_prep_not_out = [x for x in prep_without_outliers.experimental_point_cloud.values()]
```

[19]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(16, 14))

ax[0, 0].boxplot(data_raw)
ax[0, 0].set_title('Semivariances Distribution in the Raw Dataset')
ax[0, 0].set_xlabel('Lag number')
ax[0, 0].set_ylabel('Semivariance value')

ax[0, 1].boxplot(data_raw_not_out)
ax[0, 1].set_title('Raw Dataset after the Outliers Detection and Removal')
ax[0, 1].set_xlabel('Lag number')
ax[0, 1].set_ylabel('Semivariance value')

ax[1, 0].boxplot(data_prep)
ax[1, 0].set_title('Semivariances Distribution in the Preprocessed Dataset')
ax[1, 0].set_xlabel('Lag number')
ax[1, 0].set_ylabel('Semivariance value')

ax[1, 1].boxplot(data_prep_not_out)
ax[1, 1].set_title('Preprocessed Dataset after the Outliers Detection and Removal')
ax[1, 1].set_xlabel('Lag number')
ax[1, 1].set_ylabel('Semivariance value')

plt.show()
```



Clarification: Comparison of multiple variogram clouds could be challenging. We see that the largest semivariances are present in the raw data. Heavily processed data has the lowest number of outliers. The medians in each dataset are distributed over a similar pattern. How is it similar? We can check if we transform the variogram point cloud into the experimental semivariogram. Pyinterpolate has a method for it: `.calculate_experimental_variogram()`. We use it and compare four plots of semivariances to gain more insight into the transformations.

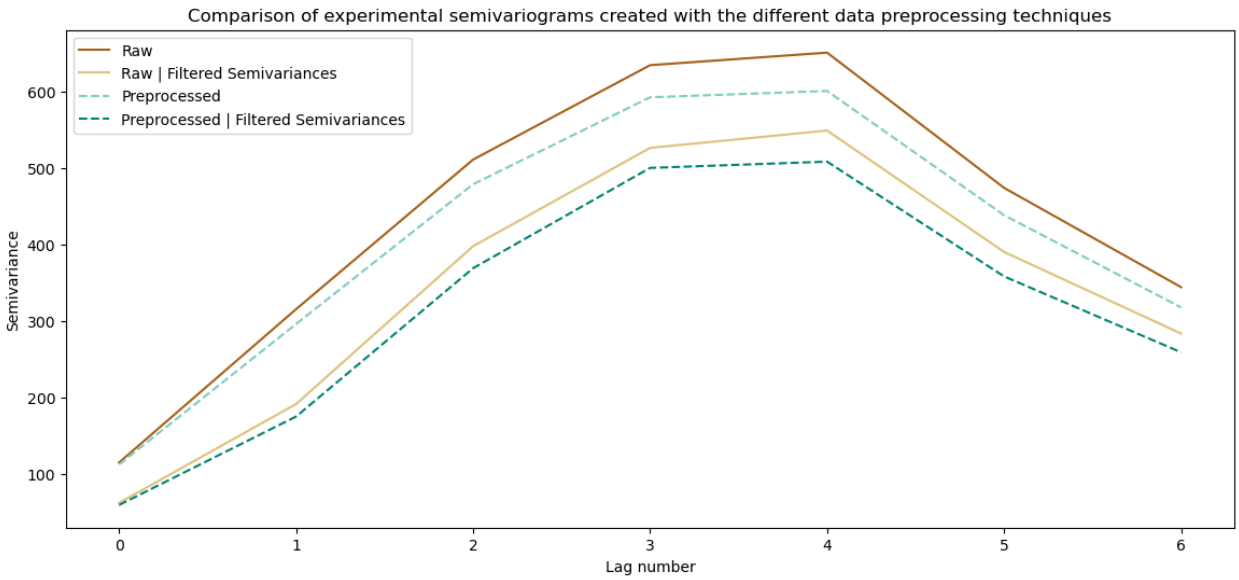
[20]:
```
raw_semivar = cloud_raw.calculate_experimental_variogram()
raw_semivar_not_out = raw_without_outliers.calculate_experimental_variogram()
prep_semivar = cloud_processed.calculate_experimental_variogram()
prep_semivar_not_out = prep_without_outliers.calculate_experimental_variogram()
```

[21]:
```
plt.figure(figsize=(14, 6))
plt.plot(raw_semivar[:, 1], c='#a6611a')
plt.plot(raw_semivar_not_out[:, 1], c='#dfc27d')
plt.plot(prep_semivar[:, 1], '--', c='#80cdc1')
plt.plot(prep_semivar_not_out[:, 1], '--', c='#018571')
plt.title('Comparison of experimental semivariograms created with the different data␣
```

(continues on next page)

(continued from previous page)

```
↪preprocessing techniques')
plt.ylabel('Semivariance')
plt.xlabel('Lag number')
plt.legend(['Raw', 'Raw | Filtered Semivariances', 'Preprocessed', 'Preprocessed |␣
↪Filtered Semivariances'])
plt.show()
```



Clarification: An understanding of those plots is a challenging task. Let's divide reasoning into multiple points:

- Raw dataset and preprocessed raw dataset show a similar pattern. The differences are more pronounced for the distant lags than for the closest point pairs,
- Datasets with the cleaned variograms are different from the raw data. The absolute semivariance values per lag are smaller, and the semivariogram pattern is slightly different. Interestingly, the possible two distributions within the dataset are more visible in the cleaned variograms (one with a peak around the 6th lag and the other with a peak around the 13th lag).
- The differences between semivariograms are primarily visible at larger distances, and differences are minor for the closest point pairs.
- **Filtering semivariance** sharply lowers the weights per lag, and **removing outliers** from the input data only slightly corrects the variogram.

Let's check how different variograms work with real-world data.

5) Create Four Ordinary Kriging models based on the four Variogram Point Clouds and compare their performance

[23]:
```
# Fit different semivariogram models into prepared datasets and variograms

# Raw
raw_theo = TheoreticalVariogram()
raw_theo.autofit(experimental_variogram=raw_semivar, return_params=False, nugget=0)
```

(continues on next page)

(continued from previous page)

```
# Raw with cleaned variogram
raw_theo_no_out = TheoreticalVariogram()
raw_theo_no_out.autofit(experimental_variogram=raw_semivar_not_out, return_params=False,␣
↪nugget=0)

# Preprocessed
prep_theo = TheoreticalVariogram()
prep_theo.autofit(experimental_variogram=prep_semivar, return_params=False, nugget=0)

# Preprocessed with cleaned variogram
prep_theo_no_out = TheoreticalVariogram()
prep_theo_no_out.autofit(experimental_variogram=prep_semivar_not_out, return_
↪params=False, nugget=0)
```

[24]:
```
# Set Kriging models

# Raw
raw_model = kriging(observations=train, theoretical_model=raw_theo, points=test[:, :-1],␣
↪allow_approx_solutions=True)

# Raw & cleaned
c_raw_model = kriging(observations=train, theoretical_model=raw_theo_no_out,␣
↪points=test[:, :-1], allow_approx_solutions=True)

# Preprocessed
prep_model = kriging(observations=train, theoretical_model=prep_theo, points=test[:, :-
↪1], allow_approx_solutions=True)

# Preprocessed & cleaned
c_prep_model = kriging(observations=train, theoretical_model=prep_theo_no_out,␣
↪points=test[:, :-1], allow_approx_solutions=True)
```

```
  0%|          | 0/6204 [00:00<?, ?it/s]/home/szymon/Documents/Programming/Repositories/
↪pyinterpolate-environment/pyinterpolate/pyinterpolate/kriging/utils/process.py:124:␣
↪UserWarning: 'Kriging system solution is based on the approximate solution, output may␣
↪be incorrect!'
  warnings.warn(LeastSquaresApproximationWarning().__str__())
100%|| 6204/6204 [00:01<00:00, 4084.98it/s]
100%|| 6204/6204 [00:01<00:00, 4905.61it/s]
100%|| 6204/6204 [00:01<00:00, 5030.64it/s]
100%|| 6204/6204 [00:01<00:00, 4967.68it/s]
```

[25]:
```
# Build test function

def calculate_model_deviation(modeled_values, test_set):
    mse = ((test_set[:, -1] - modeled_values[:, 0])**2)
    return mse
```

[26]:
```
r_test = calculate_model_deviation(raw_model, test)
```

[27]:
```
cr_test = calculate_model_deviation(c_raw_model, test)
```

[28]:
```
p_test = calculate_model_deviation(prep_model, test)
```

[29]:
```
cp_test = calculate_model_deviation(c_prep_model, test)
```

[30]:
```
df = pd.DataFrame(data=np.array([r_test, cr_test, p_test, cp_test]).transpose(),
                  columns=['Raw', 'Raw-cleaned', 'Preprocessed', 'Preprocessed-cleaned'])
```

We will use `.describe()` method from **pandas** to get the errors statistic:

[31]:
```
df.describe()
```

[31]:

	Raw	Raw-cleaned	Preprocessed	Preprocessed-cleaned
count	6.204000e+03	6.204000e+03	6.204000e+03	6.204000e+03
mean	4.112253e+03	1.125479e+03	1.616366e+03	9.951413e+03
std	1.496551e+05	2.407578e+04	4.314202e+04	7.176860e+05
min	7.560711e-08	3.693612e-07	4.051043e-07	2.012364e-07
25%	9.082049e-01	8.223928e-01	9.574270e-01	8.203482e-01
50%	7.449470e+00	7.218670e+00	7.740437e+00	7.111464e+00
75%	5.507547e+01	5.230695e+01	5.615120e+01	5.161227e+01
max	1.004389e+07	1.049016e+06	2.625928e+06	5.651596e+07

The view of this table may be different each time you run this tutorial (and it is related to the training-test division process). In some cases, you may get better results for the cleaned dataset; for others, results will be better for raw data. That's why we should always have a validation set to compare different scenarios. The other idea is to use all of the models and create a distribution of predictions and errors per point - we can use it later for decisions.

> **NOTE:** The example in this tutorial is related to the Digital Elevation Model, which the data provider preprocessed (*Copernicus Land Monitoring Services*). You shouldn't get the impression that raw data preprocessing and filtering are not required for the analysis. There are cases where the sensor may produce unreliable and biased results, such as a satellite camera's saturated pixel. Removing it with the specific noise-filtering algorithm before the variogram point cloud development is better.

Where to go from here?

- B.2.1 Directional Ordinary Kriging
- C.1.1 Blocks to Points Interpolation with Ordinary Kriging
- C.1.2 Semivariogram Regularization

Changelog

Date	Change description	Author
2023-08-23	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@SimonMolinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@SimonMolinsky
2022-10-08	The tutorial updated to the version 0.3.2 of the package	@SimonMolinsky
2022-08-27	The tutorial updated to the version 0.3.0 of the package	@SimonMolinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@SimonMolinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@SimonMolinsky
2021-12-11	Behavior of `prepare_kriging_data()` function has benn changed	@SimonMolinsky
2021-10-13	Refactored TheoreticalSemivariogram (name change of class attribute) and refactored `calc_semivariance_from_pt_cloud()` function to protect calculations from `NaN's`.	@ethmtrgt & @SimonMolinsky
2021-08-22	Initial release	@SimonMolinsky

[ ]:

B.2.1 Directional Ordinary Kriging

Table of Contents:

1. Read *meuse* dataset, transform data,
2. Create directional semivariograms,
3. Create directional kriging models,
4. Compare results of interpolation.

Introduction

In this tutorial, we will learn about Ordinary Kriging with directions. The classic Kriging works with an isotropic semivariogram. The similarity between points has the same weights in all directions. Actual geographical data is rarely isotropic, and we distinguish leading directions in spatial processes. The example may be a temperature, where differences in the North-South axis are greater than in the West-East axis.

1. Read *meuse* dataset and transform data

We use *meuse* dataset and interpolate zinc concentrations. Dataset is provided in:

> Pebesma, Edzer. (2009). The meuse data set: a tutorial for the gstat R package -> link to the publication

[1]:
```
import geopandas as gpd
import numpy as np
import pandas as pd
import pyinterpolate as ptp
from pyinterpolate.variogram.empirical.experimental_variogram import DirectionalVariogram
import matplotlib.pyplot as plt
```

[2]:
```
MEUSE_FILE = 'samples/point_data/csv/meuse/meuse.csv'
MEUSE_GRID = 'samples/point_data/csv/meuse/meuse_grid.csv'

# Variogram parameters
STEP_SIZE = 100
MAX_RANGE = 1600
```

[3]:
```
df = pd.read_csv(MEUSE_FILE, usecols=['x', 'y', 'zinc'])
```

[4]:
```
df.head()
```

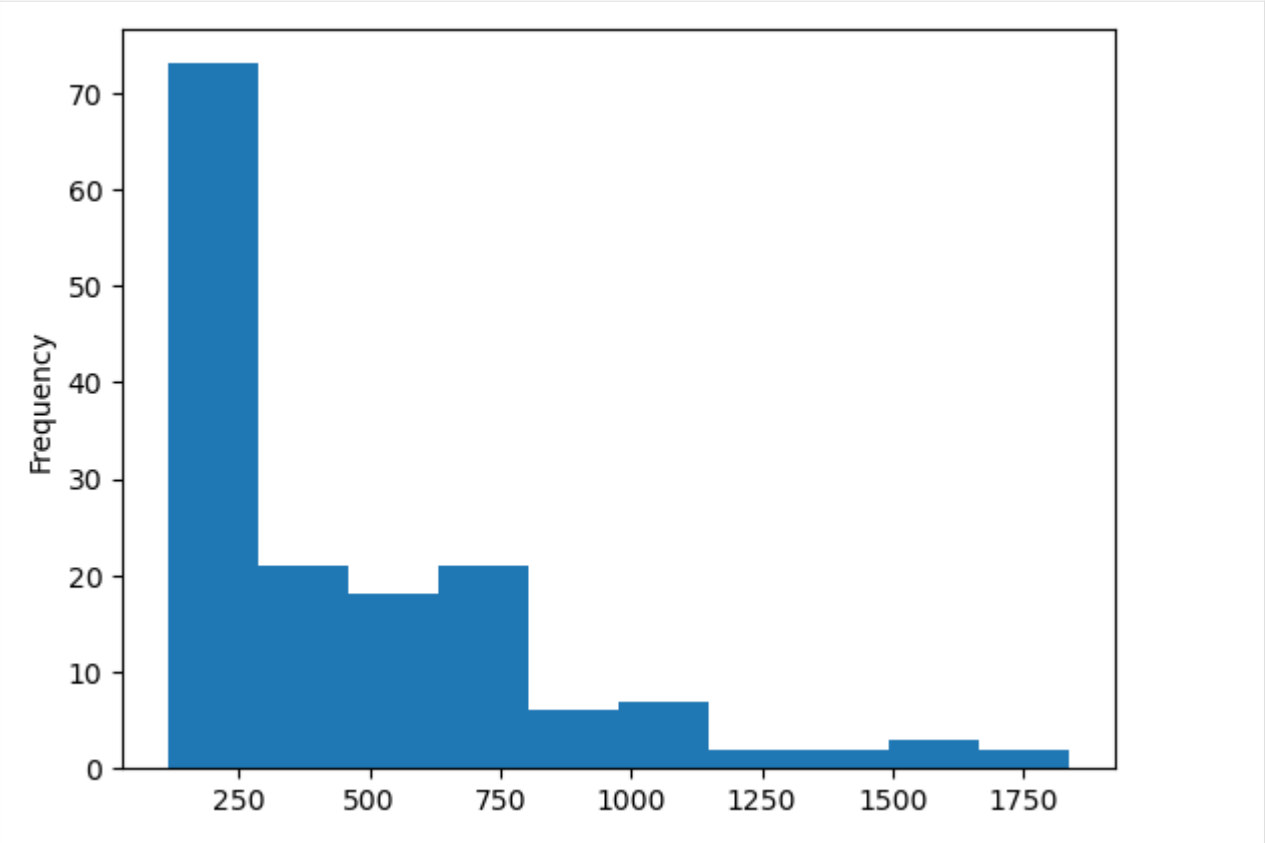
[4]:

	x	y	zinc
0	181072	333611	1022
1	181025	333558	1141
2	181165	333537	640
3	181298	333484	257
4	181307	333330	269

Let's check the distribution of zinc values:

[5]:
```
df['zinc'].plot(kind='hist')
```

[5]:
```
<Axes: ylabel='Frequency'>
```

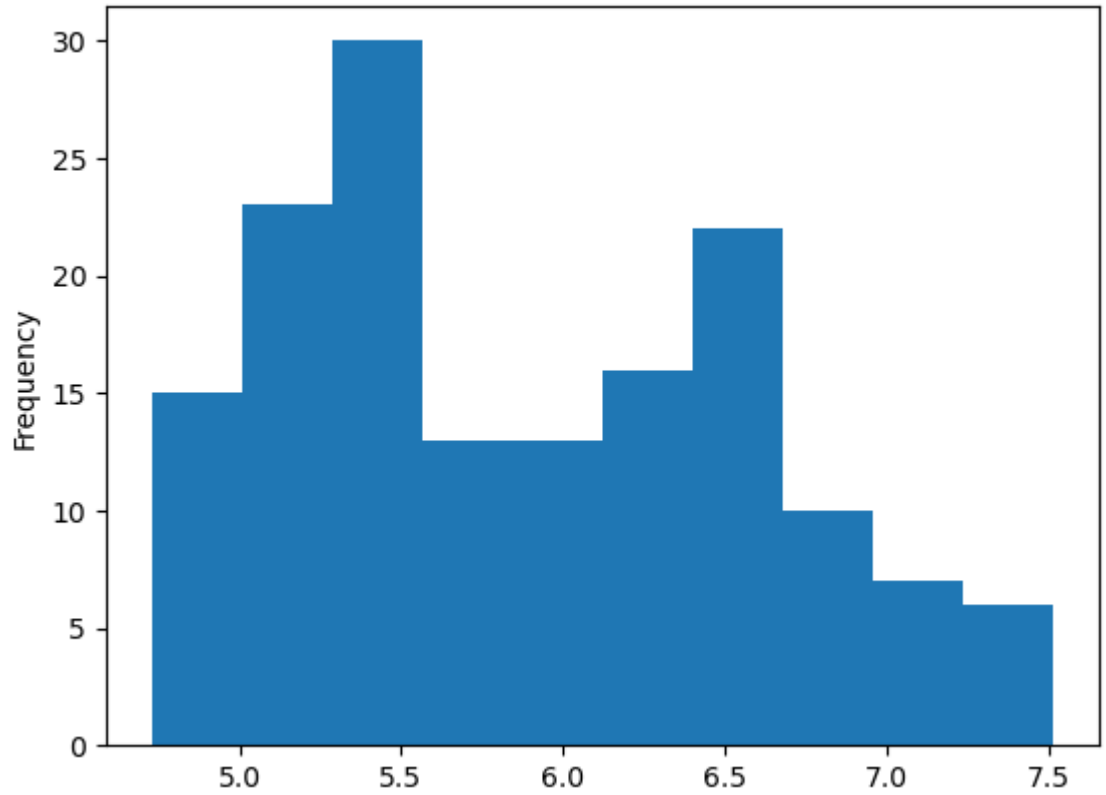


Our zinc concentrations are highly skewed. We should take a natural logarithm of values to make the distribution closer to the normal.

[6]:
```
df['log-zinc'] = np.log(df['zinc'])
```

[7]:
```
df['log-zinc'].plot(kind='hist')
```

[7]:
```
<Axes: ylabel='Frequency'>
```

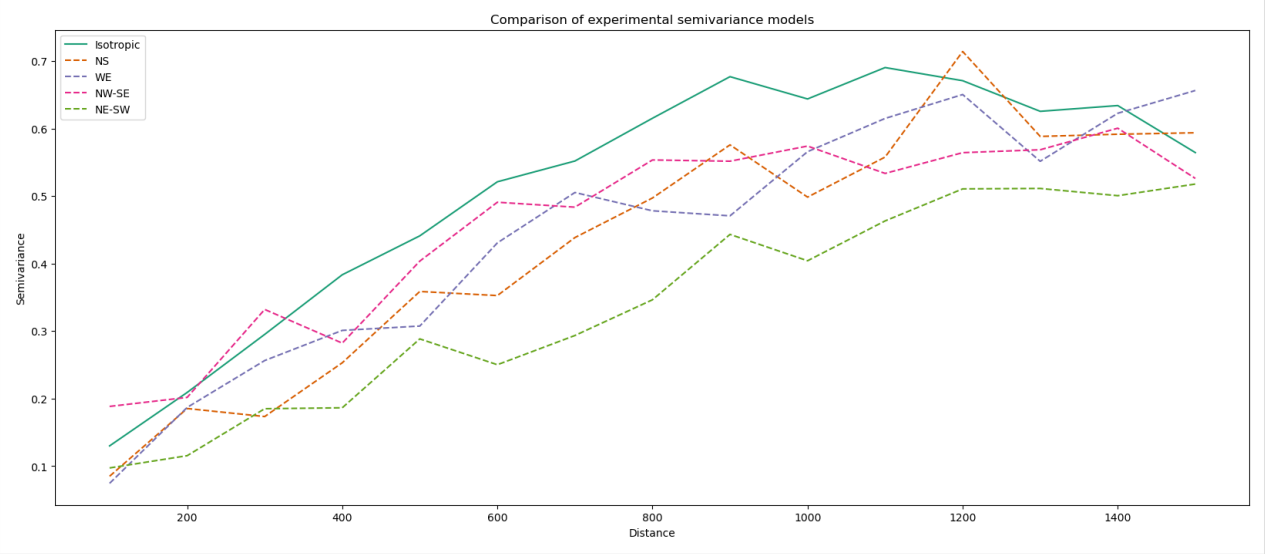


The histogram is still not close to the normal distribution; it has two modes, but the tail is shorter. We will build models based on this histogram.

2. Create directional variograms

Data is prepared, we can model it's semivariance. We will perform an isotropic modeling and a full directional modeling in *N-S*, *W-E*, *NE-SW*, and *NW-SE* axes.

[8]:
```
dirvar = DirectionalVariogram(input_array=df[['x', 'y', 'log-zinc']],
                              step_size=STEP_SIZE,
                              max_range=MAX_RANGE,
                              tolerance=0.1,
                              method='e')
dirvar.show()
```



We have five experimental variograms for further modeling. Let's build theoretical models from those.

[11]:
```
MODEL_NAME = 'spherical'
DIRECTIONS = ['ISO', 'NS', 'WE', 'NE-SW', 'NW-SE']
```

[12]:
```
theoretical_models = {}
variograms = dirvar.get()

for direction in DIRECTIONS:
    theoretical_model = ptp.TheoreticalVariogram()
    theoretical_model.autofit(experimental_variogram=variograms[direction],
                              model_name=MODEL_NAME,
                              nugget=0)
    theoretical_models[direction] = theoretical_model
```

3. Create directional Ordinary Kriging models

With a set of directional variograms we will perform spatial interpolation over a defined grid.

[13]:
```
grid = pd.read_csv(MEUSE_GRID, usecols=['x', 'y'])
grid.head()
```

[13]:

	x	y
0	181180	333740
1	181140	333700
2	181180	333700
3	181220	333700
4	181100	333660

[16]:
```
kriged_results = {}

for direction in DIRECTIONS:
    result = ptp.kriging(observations=df[['x', 'y', 'log-zinc']].values,
                         theoretical_model=theoretical_models[direction],
```

(continues on next page)

(continued from previous page)

```
                        points=grid.values,
                        no_neighbors=16,
                        use_all_neighbors_in_range=True)
    result = pd.DataFrame(result, columns=['pred', 'err', 'x', 'y'])
    rgeometry = gpd.points_from_xy(result['x'], result['y'])
    rgeometry.name = 'geometry'
    result = gpd.GeoDataFrame(result, geometry=rgeometry)

    kriged_results[direction] = result
```

```
100%|| 3103/3103 [00:03<00:00, 776.43it/s]
100%|| 3103/3103 [00:00<00:00, 4160.84it/s]
100%|| 3103/3103 [00:00<00:00, 3911.17it/s]
100%|| 3103/3103 [00:00<00:00, 3740.61it/s]
100%|| 3103/3103 [00:00<00:00, 4165.74it/s]
```

[17]:
```
kriged_results['ISO'].head()
```

[17]:

	pred	err	x	y	geometry
0	6.523941	0.283231	181180.0	333740.0	POINT (181180.000 333740.000)
1	6.664006	0.198841	181140.0	333700.0	POINT (181140.000 333700.000)
2	6.541937	0.224790	181180.0	333700.0	POINT (181180.000 333700.000)
3	6.407741	0.254611	181220.0	333700.0	POINT (181220.000 333700.000)
4	6.809450	0.109191	181100.0	333660.0	POINT (181100.000 333660.000)

With all models calculated, we can compare the results of interpolation. We will show interpolation and variance errors across the models.

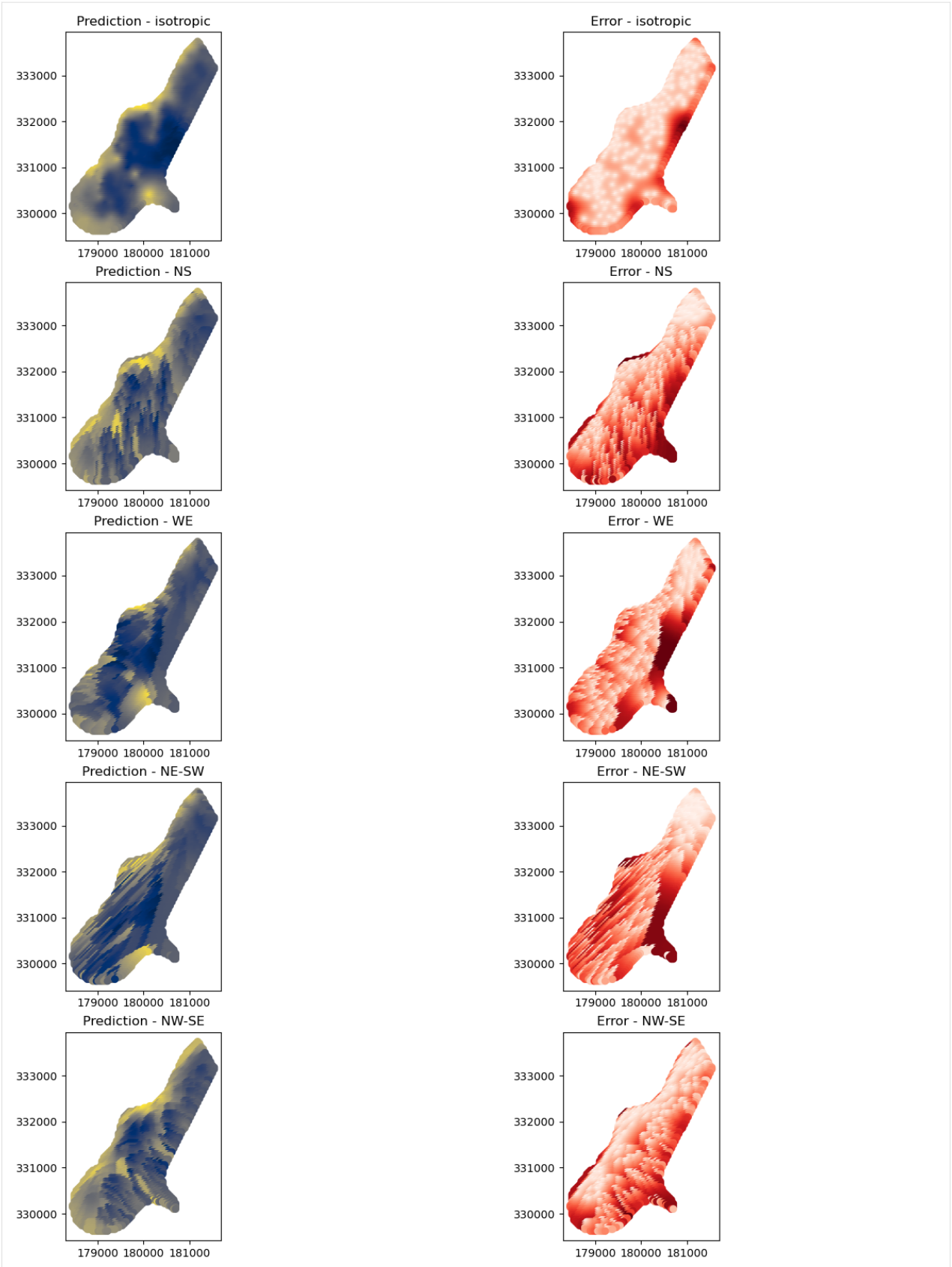
4. Compare interpolated results

[18]:
```
fig, ax = plt.subplots(nrows=5, ncols=2, figsize=(15, 20))

kriged_results['ISO'].plot(ax=ax[0, 0], column='pred', cmap='cividis')
kriged_results['ISO'].plot(ax=ax[0, 1], column='err', cmap='Reds')
ax[0, 0].set_title('Prediction - isotropic')
ax[0, 1].set_title('Error - isotropic')

kriged_results['NS'].plot(ax=ax[1, 0], column='pred', cmap='cividis')
kriged_results['NS'].plot(ax=ax[1, 1], column='err', cmap='Reds')
ax[1, 0].set_title('Prediction - NS')
ax[1, 1].set_title('Error - NS')

kriged_results['WE'].plot(ax=ax[2, 0], column='pred', cmap='cividis')
kriged_results['WE'].plot(ax=ax[2, 1], column='err', cmap='Reds')
ax[2, 0].set_title('Prediction - WE')
ax[2, 1].set_title('Error - WE')

kriged_results['NE-SW'].plot(ax=ax[3, 0], column='pred', cmap='cividis')
kriged_results['NE-SW'].plot(ax=ax[3, 1], column='err', cmap='Reds')
ax[3, 0].set_title('Prediction - NE-SW')
ax[3, 1].set_title('Error - NE-SW')
```

(continues on next page)

(continued from previous page)

```
kriged_results['NW-SE'].plot(ax=ax[4, 0], column='pred', cmap='cividis')
kriged_results['NW-SE'].plot(ax=ax[4, 1], column='err', cmap='Reds')
ax[4, 0].set_title('Prediction - NW-SE')
ax[4, 1].set_title('Error - NW-SE')

plt.show()
```

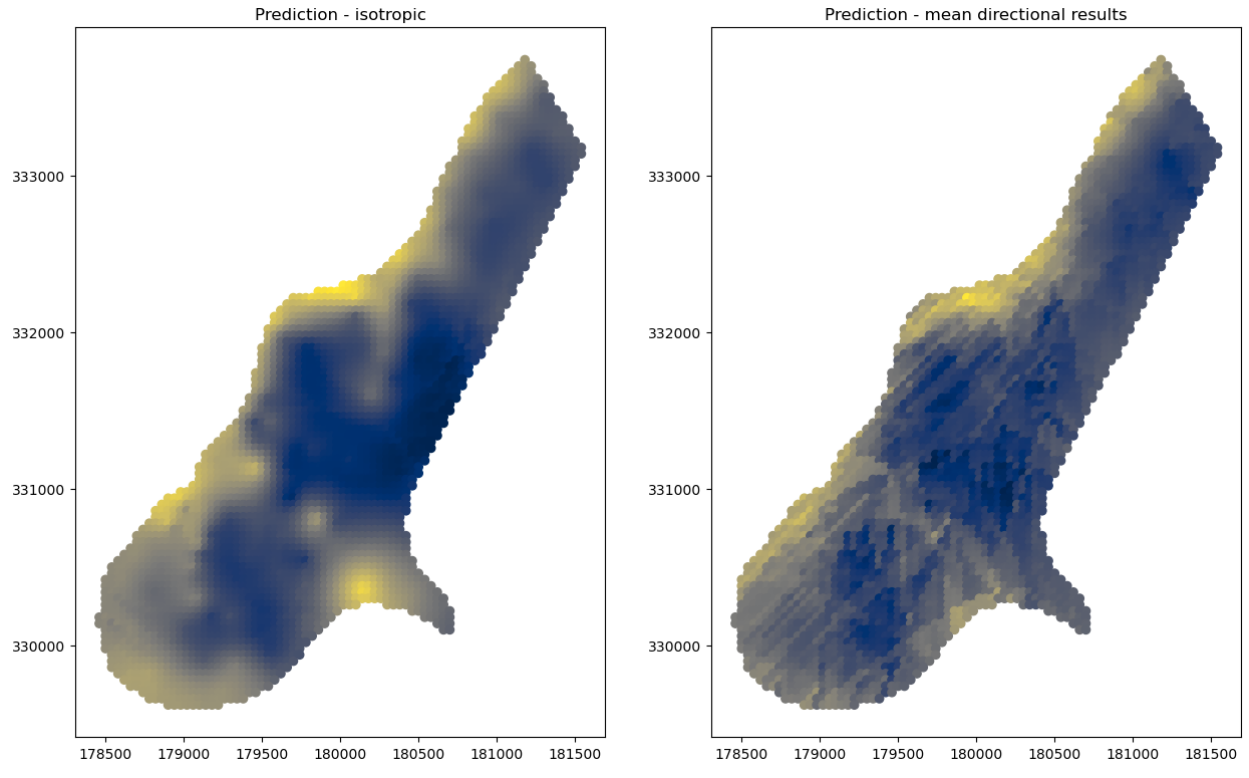

Prediction - isotropic
Error - isotropic
Prediction - NS
Error - NS
Prediction - WE
Error - WE
Prediction - NE-SW
Error - NE-SW
Prediction - NW-SE
Error - NW-SE
333000
332000
331000
330000
179000 180000 181000

Directional Kriging clearly shows the leading direction and is even more pronounced in the variance error maps. At the end, let's compare the mean of directional interpolations to the isotropic interpolation result:

[19]:
```
mean_directional_results = (kriged_results['NS']['pred'] +
                            kriged_results['WE']['pred'] +
                            kriged_results['NE-SW']['pred'] +
                            kriged_results['NW-SE']['pred']) / 4
```

[20]:
```
kriged_results['ISO']['mean_dir_pred'] = mean_directional_results
```

[21]:
```
_, ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 20))

kriged_results['ISO'].plot(ax=ax[0], column='pred', cmap='cividis')
ax[0].set_title('Prediction - isotropic')
kriged_results['ISO'].plot(ax=ax[1], column='mean_dir_pred', cmap='cividis')
ax[1].set_title('Prediction - mean directional results')
plt.show()
```



Where to go from here?

- C.1.1 Blocks to Points Interpolation with Ordinary Kriging
- C.1.2 Semivariogram Regularization

Changelog

Date	Change description	Author	Package Version
2023-08-23	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky	0.5.0
2023-04-15	Update: version 0.4.1	@Simon-Molinsky	0.4.1
2022-11-19	The first version of tutorial	@Simon-Molinsky	0.3.6

[ ]:

2.3.3 Advanced

C.1.1 Blocks to points Ordinary Kriging

Table of Contents:

1. Read block data,
2. Detect and remove outliers,
3. Create semivariogram model,
4. Read point data canvas,
5. Build map of interpolated values,
6. Show map of interpolated values with choropleth map of the breast cancer rates.

Introduction

This tutorial is based on the question of user *@ikey* from *Discord* channel of the package (02/04/2021). Thank you *@ikey* for your engagement!

In short:

> Are we able to perform Semivariogram Regularization and Poisson Kriging if our point support represents only points without any values?

No, it's not possible. But there is a hack that we can use - we can interpolate missing values from areal centroids with Ordinary and Simple Kriging just like we do with *regular points*. Why is it not possible with unknown points?

The idea behind semivariogram regularization of areal aggregates is to use semivariogram of point support. Point support must reflect an ongoing process. For example, in the case of epidemiology:

1. We have areal counts of infections divided by POPULATION over some areas.
2. We take POPULATION blocks (points) with a number of inhabitants per block and use those as support for our map. Thus we transform the choropleth map into the point map, and only to the points where someone is living ==> someone may be infected. We skip large parts of the map where the risk of infection is not present because no one is living there.
3. We finally obtain the population-at-risk map of points or smoothed choropleth map of infection risk without large visual bias where large/small areas seem to be more important.

This works when we have specific **counts over area divided by another parameter** (time, population, probability, volume, etc.). In this scenario, we can find a function that links our areal counts with those underlying variables and/or processes. Without support values, we are not able to perform semivariogram regularization.

What we want to achieve may be done with Ordinary Kriging (because our points are unknown), and semivariogram regularization is not required (and not possible without any values).

In summary, we will learn how to:

- transform areal data into points,
- detect and remove outliers from data,
- create an interpolated map from unknown points,
- visualize and save the created map.

We use:

- for point canvas: `samples/point_data/shapefile/regular_grid_points.shp`,
- for areal centroids: Breast cancer rates data is stored in the shapefile `samples/regularization/cancer_data.gpkg`.

This tutorial requires understanding concepts presented in **Semivariogram Estimation** and **Variogram Point Cloud** tutorials along with **Ordinary and Simple Kriging** notebook and (optionally) **Semivariogram Regularization** tutorial.

Import packages

[1]:
```
import numpy as np
import geopandas as gpd

import matplotlib.pyplot as plt

from pyinterpolate import Blocks, VariogramCloud, TheoreticalVariogram, kriging
```

1) Read areal data

The block data represents the breast cancer rates in the Northeastern part of the United States. Rate is the number of new cases divided by the population in a county, and this fraction is multiplied by a constant factor of 100,000.

We read data from `geopackage` and use only `rate`, `centroid.x`, and `centroid.y` columns.

[2]:
```
# Read and prepare data

DATASET = 'samples/regularization/cancer_data.gpkg'
POLYGON_LAYER = 'areas'
```

(continues on next page)

(continued from previous page)

```
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'
MAX_RANGE = 400000
STEP_SIZE = 40000

AREAL_INPUT = Blocks()
AREAL_INPUT.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
↪name=POLYGON_LAYER)

AREAL_INPUT.data.head()
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

[2]:

```
    FIPS                                           geometry   rate  \
0  25019  MULTIPOLYGON (((2115688.816 556471.240, 211569...  192.2
1  36121  POLYGON ((1419423.430 564830.379, 1419729.721 ...  166.8
2  33001  MULTIPOLYGON (((1937530.728 779787.978, 193751...  157.4
3  25007  MULTIPOLYGON (((2074073.532 539159.504, 207411...  156.7
4  25001  MULTIPOLYGON (((2095343.207 637424.961, 209528...  155.3

     centroid_x     centroid_y
0  2.132630e+06  557971.155949
1  1.442153e+06  550673.935704
2  1.958207e+06  766008.383446
3  2.082188e+06  556830.822367
4  2.100747e+06  600235.845891
```

[3]:

```
areal_centroids = AREAL_INPUT.data[['centroid_x', 'centroid_y', 'rate']].values
areal_centroids[0]
```

[3]:

```
array([2.13262960e+06, 5.57971156e+05, 1.92200000e+02])
```

2) Detect and remove outliers

We check and clean data before the semivariogram fitting and the Ordinary Kriging interpolation. Variogram Point Cloud is the best way to analyze data and detect outliers. We inspect and compare different lags and step sizes and their respective point clouds. Then we will create multiple variograms and Kriging models.

[4]:

```
# Create analysis parameters

maximum_range = 300000

number_of_lags = [4, 8, 16]
step_sizes = [maximum_range / x for x in number_of_lags]

variogram_clouds = []

for step_size in step_sizes:
    vc = VariogramCloud(input_array=areal_centroids, step_size=step_size, max_
```

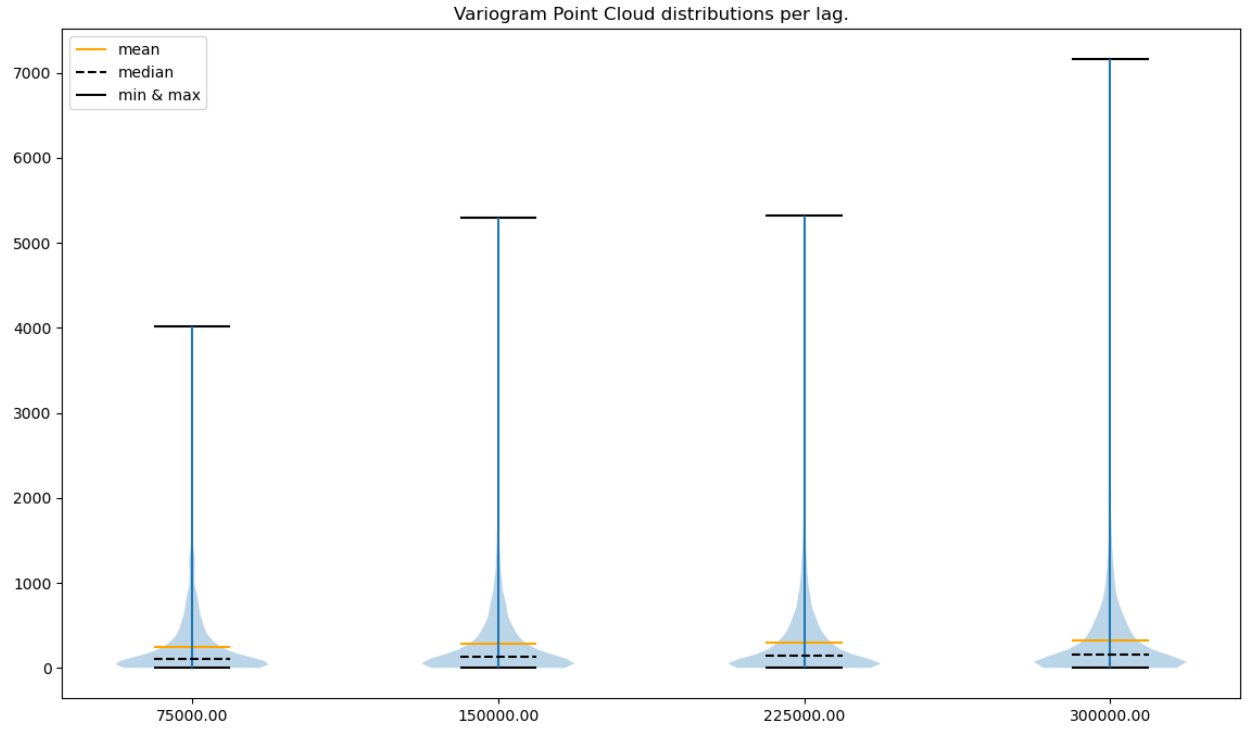
(continues on next page)

(continued from previous page)

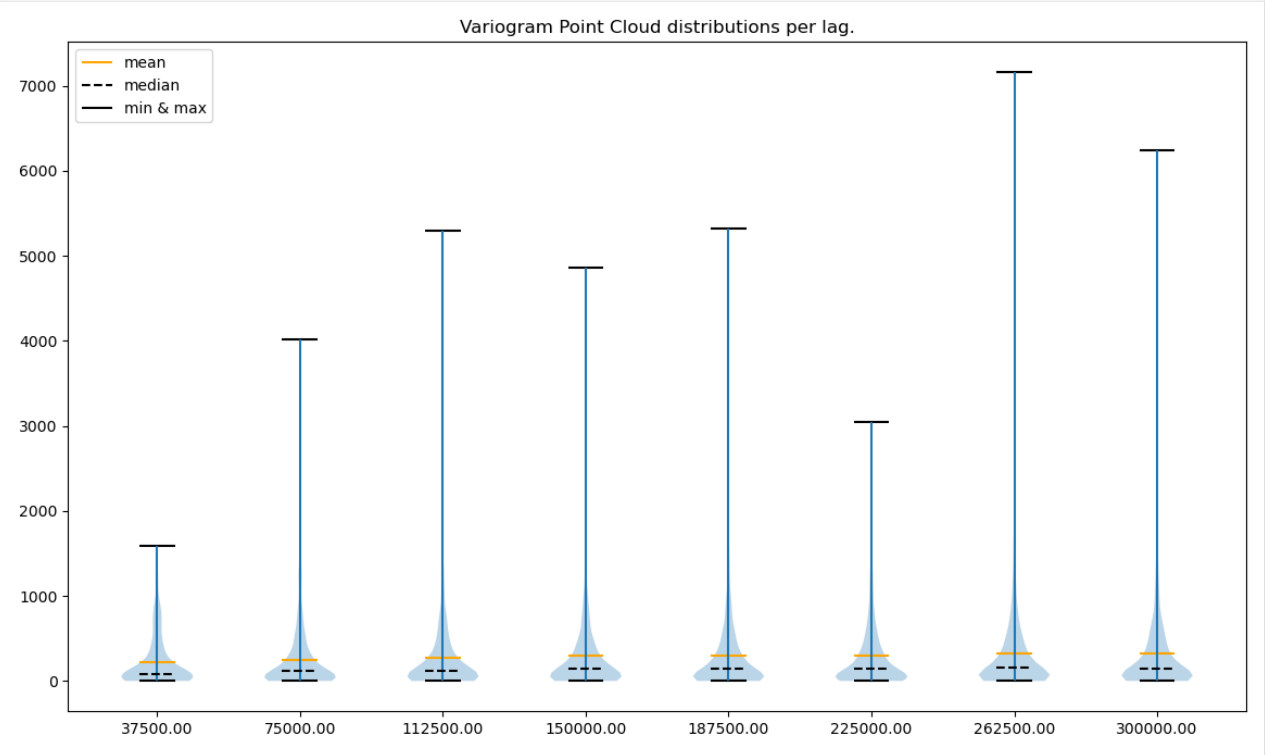
```
    ↪range=maximum_range+step_size)
        variogram_clouds.append(vc)
```

[5]:
```
for vc in variogram_clouds:
    print(f'Lags per area: {len(vc.lags)}')
    print('')
    vc.plot('violin')
    print('\n###########\n')
```

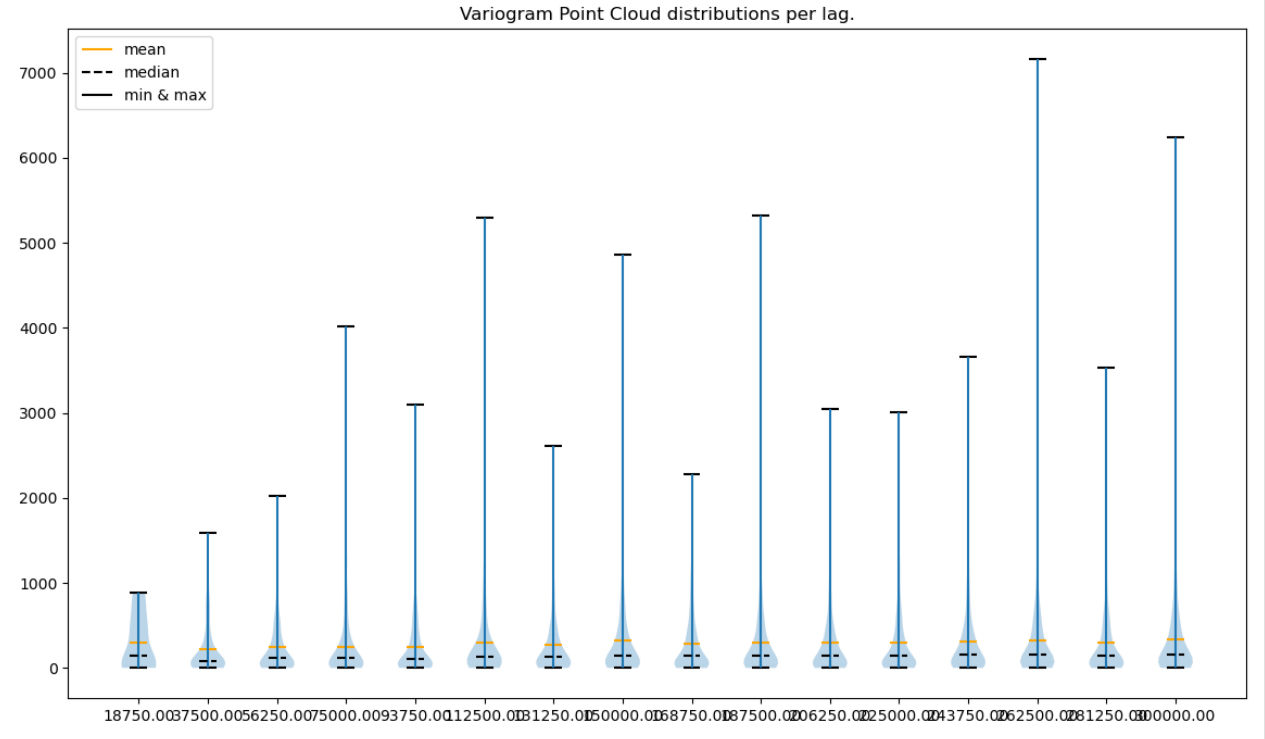
```
Lags per area: 4
```




```
###########

Lags per area: 8
```




```
###########

Lags per area: 16
```



```
###########
```

What does a fast check tell us?

- point to point errors per lag are skewed toward positive values (the mean is always greater than the median),
- less lags == less variability,
- variability in the last case (32 lags) seems to be too high,
- variability in the first case (4 lags) seems to be too small,
- there are extreme semivariance values, and the largest extremities are present for the most distant lags.

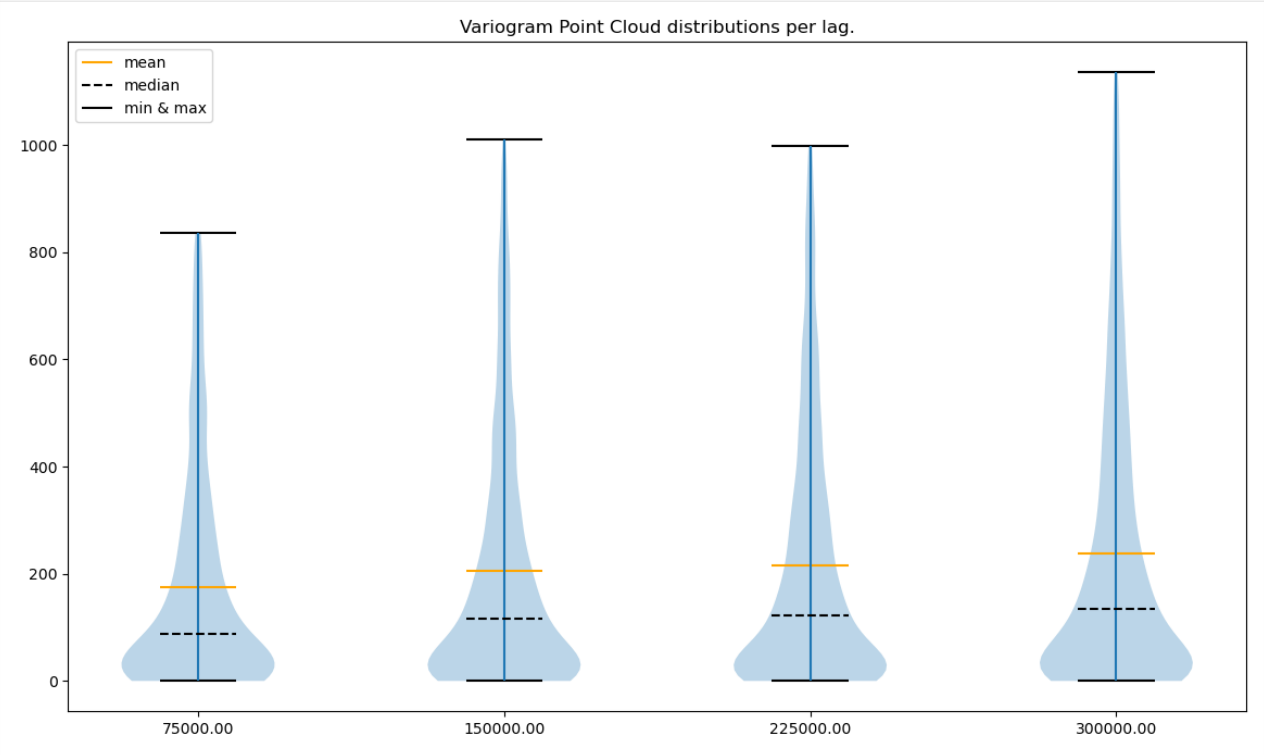
What can we do?

- remove outliers (extremely high semivariances) per each lag,
- use a small number of neighbors and a range close to 200,000 to avoid unnecessary computations.

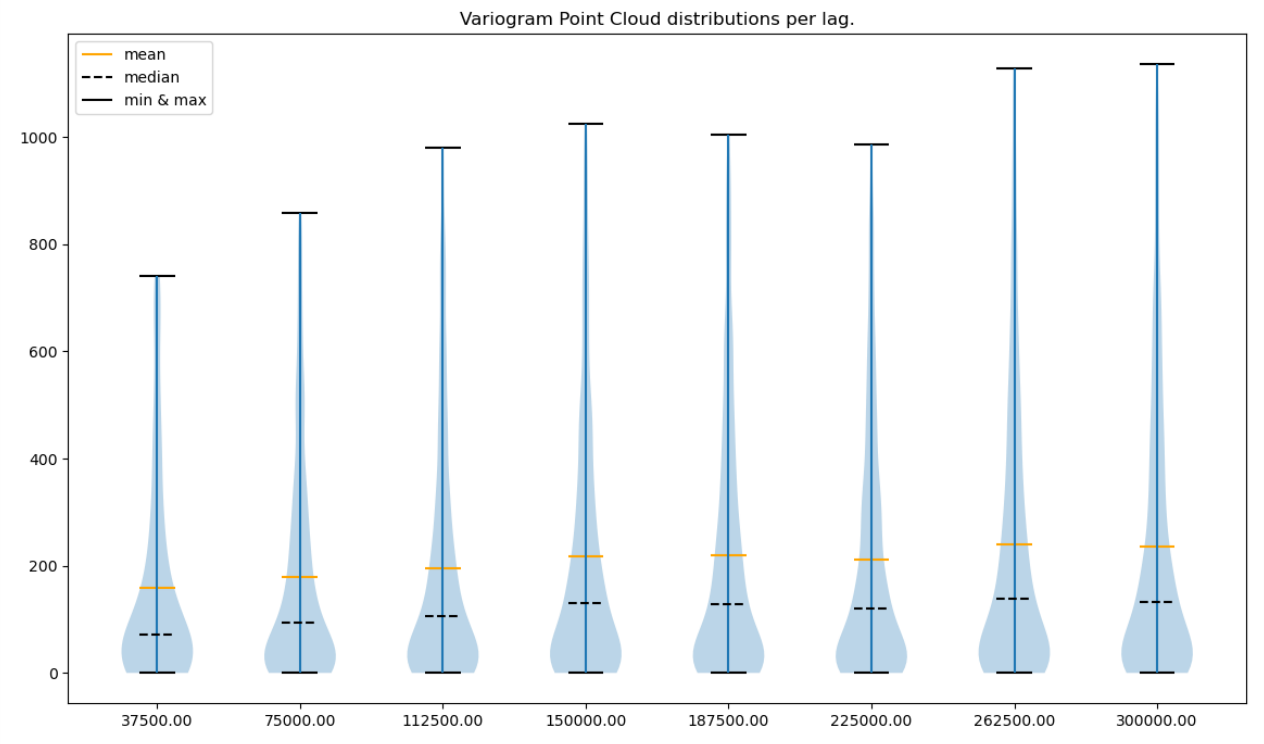
[6]:
```
# Now remove outliers from each lag

_ = [vc.remove_outliers(method='iqr', iqr_lower_limit=3, iqr_upper_limit=1.5,␣
↪inplace=True) for vc in variogram_clouds]
```

[7]:
```
# And show data without outliers

for vc in variogram_clouds:
    print(f'Lags per area: {len(vc.lags)}')
    print('')
    vc.plot('violin')
    print('\n###########\n')
```

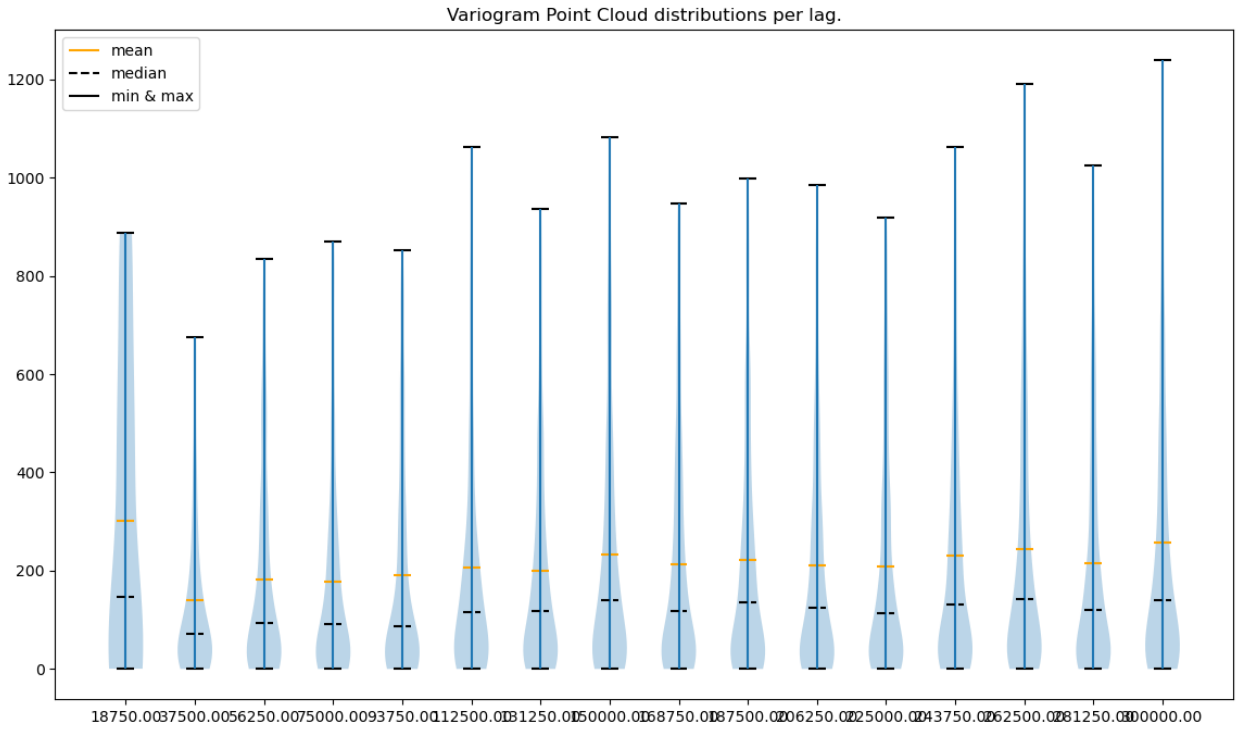
```
Lags per area: 4
```




```
###########

Lags per area: 8
```



```
###########

Lags per area: 16
```




```
###########
```

If we compare the **y axis** before and after removing outliers, we may see that values are now smaller and densely populated. After outlier removal, we see also that 32 lags are too much, and especially for the first lag, it generates too much noise. We can expect that model based on this division will work poorly.

3. Create a theoretical semivariogram model

Semivariogram may be fitted manually or automatically. In this case, we fit it automatically - we test multiple models, and it's easier to use the `autofit()` method of the `TheoreticalVariogram` object. We didn't force any model to see how different results we would get.

[8]:
```
theoretical_semivariograms = []

for idx, vc in enumerate(variogram_clouds):

    print(f'Semivariance calculated for {vc.lags} lags.')
    print('')
    # Calculate experimental model
    exp_model = vc.calculate_experimental_variogram()

    # Assign experimental model and data to TheoreticalSemivariogram
```

(continues on next page)

(continued from previous page)

```
    theo_semi = TheoreticalVariogram()
    theo_semi.autofit(experimental_variogram=exp_model, nugget=0)
    theoretical_semivariograms.append(theo_semi)
    print('')
    print('Model parameters:')
    print('Model type:', theo_semi.name)
    print('Nugget:', theo_semi.nugget)
    print('Sill:', theo_semi.sill)
    print('Range:', theo_semi.rang)
    print('Model error:', theo_semi.rmse)
    print('')
    print('#####')
```

```
Semivariance calculated for [ 75000. 150000. 225000. 300000.] lags.


Model parameters:
Model type: spherical
Nugget: 0
Sill: 104.18060667237297
Range: 118000.0
Model error: 7.462726772043129

#####
Semivariance calculated for [ 37500.  75000. 112500. 150000. 187500. 225000. 262500.␣
↪300000.] lags.


Model parameters:
Model type: spherical
Nugget: 0
Sill: 109.59322531046718
Range: 70000.0
Model error: 9.709346473449811

#####
Semivariance calculated for [ 18750.  37500.  56250.  75000.  93750. 112500. 131250.␣
↪150000. 168750.
 187500. 206250. 225000. 243750. 262500. 281250. 300000.] lags.


Model parameters:
Model type: spherical
Nugget: 0
Sill: 111.06413642423938
Range: 30000.0
Model error: 21.29533062806228

#####
```

```
/home/szymon/Documents/Programming/Repositories/pyinterpolate-environment/pyinterpolate/
↪pyinterpolate/variogram/theoretical/semivariogram.py:509: UserWarning: If you provide␣
↪experimental variogram as a numpy array you must remember that the direction parameter␣
```

(continues on next page)

(continued from previous page)

```
↪must be set if it is a directional variogram. Otherwise, algorithm assumes that␣
↪variogram is isotropic.
  warnings.warn(msg)
```

Each case has a different model type, a different sill, and a different range! How do we choose the model parameters appropriately in this scenario? Error rises with the number of lags but is it a good indicator of the semivariogram fit? No, and we should be careful when choosing variograms with a few lags instead of variograms with multiple lags. We may miss some spatial patterns that will be averaged with a smaller number of lags. The 8-lag variogram seems to be the best because RMSE is the lowest. On the other hand, 32-lags variograms work poorly!

4. Read point data canvas

Our variogram model is ready to load the point file as the **GeoDataFrame** object, which stores the geometry column and data features as tables.

[9]:
```
points = 'samples/point_data/shapefile/regular_grid_points.shp'  # file with grid for␣
↪analysis
gdf_pts = gpd.read_file(points)
gdf_pts.set_index('id', inplace=True)
gdf_pts['x'] = gdf_pts.geometry.x
gdf_pts['y'] = gdf_pts.geometry.y
gdf_pts.head()
```

[9]:

id	geometry	x	y
81.0	POINT (1277277.671 441124.507)	1.277278e+06	441124.5068
82.0	POINT (1277277.671 431124.507)	1.277278e+06	431124.5068
83.0	POINT (1277277.671 421124.507)	1.277278e+06	421124.5068
84.0	POINT (1277277.671 411124.507)	1.277278e+06	411124.5068
85.0	POINT (1277277.671 401124.507)	1.277278e+06	401124.5068

5. Build a map of interpolated values

We can model all values in a batch. If we set the `number_of_workers` parameter to `-1` or a positive integer, then the algorithm will use parallel processing functions to speed up calculations.

Based on the learning from a semivariogram modeling step, we will set a number of neighboring areas to 8 and range to 200000. We will append interpolated values and errors to an existing data frame and plot them to compare results.

[10]:
```
preds_cols = []
errs_cols = []

for semi_model in theoretical_semivariograms:
    kriged = kriging(observations=areal_centroids,
                     theoretical_model=semi_model,
                     points=gdf_pts[['x', 'y']].values,
                     neighbors_range=200000,
                     no_neighbors=8,
                     number_of_workers=1,
                     use_all_neighbors_in_range=True)
```

(continues on next page)

(continued from previous page)

```
    # Interpolate missing values and uncertainty
    pred_col_name = 'p ' + semi_model.name[:3] + ' ' + str(len(semi_model.lags))
    uncertainty_col_name = 'e ' + semi_model.name[:3] + ' ' + str(len(semi_model.lags))
    gdf_pts[pred_col_name] = kriged[:, 0]
    gdf_pts[uncertainty_col_name] = kriged[:, 1]
    preds_cols.append(pred_col_name)
    errs_cols.append(uncertainty_col_name)
```

```
100%|| 5419/5419 [00:54<00:00, 99.68it/s]
100%|| 5419/5419 [00:26<00:00, 206.25it/s]
100%|| 5419/5419 [00:22<00:00, 238.09it/s]
```

[11]:
```
gdf_pts.head()
```

[11]:

id	geometry	x	y	p sph 4	e sph 4	p sph 8	e sph 8	p sph 16	e sph 16
81.0	POINT (1277277.671 441124.507)	1.277278e+06	441124.5068	132.133667	65.848568	133.061501	100.081621	131.969565	115.893012
82.0	POINT (1277277.671 431124.507)	1.277278e+06	431124.5068	130.583635	64.566536	131.102959	100.171994	130.604348	115.893012
83.0	POINT (1277277.671 421124.507)	1.277278e+06	421124.5068	129.607074	63.776407	130.397073	99.884160	130.604348	115.893012
84.0	POINT (1277277.671 411124.507)	1.277278e+06	411124.5068	128.854301	63.223374	130.168520	99.475313	130.604348	115.893012
85.0	POINT (1277277.671 401124.507)	1.277278e+06	401124.5068	128.549723	62.707759	130.652012	98.828762	130.860000	115.506702

This is the somewhat complicated function in which we interpolate missing values as new columns of **GeoDataFrame**. *Uncertainty* is assigned to interpolated results for further analysis.

Now we can save our **GeoDataFrame** as a shapefile.

[12]:
```
# Save interpolation results

gdf_pts.to_file('output/interpolation_results_areal_to_point.shp')
```

Now we will check the results directly in the notebook.

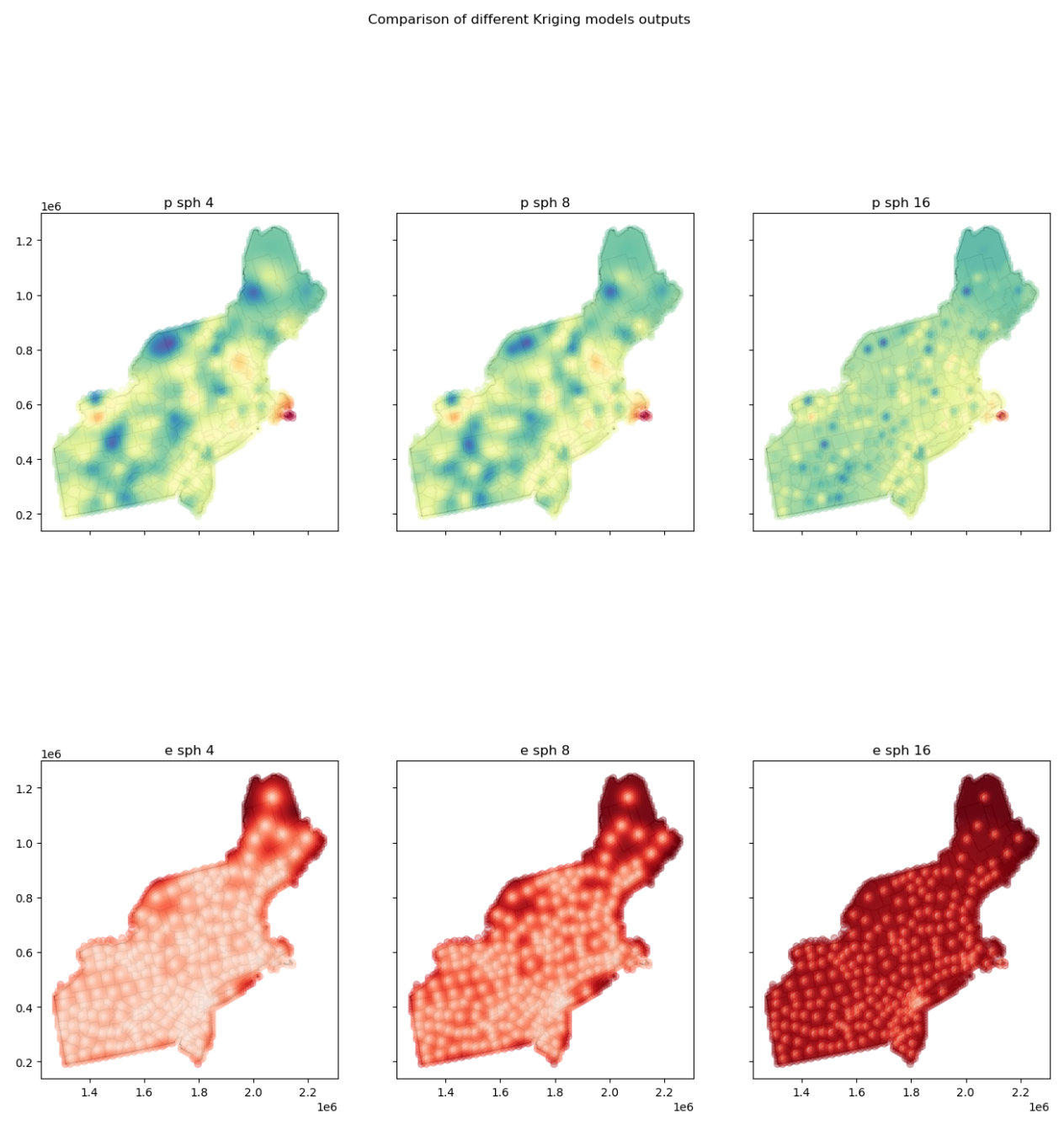
6. Show a map of interpolated values with a choropleth map of the breast cancer rates

[13]:
```
# Now compare results to choropleth maps
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(16, 16), sharex='all', sharey='all')

base1 = AREAL_INPUT.data.plot(ax=axes[0, 0], legend=True, edgecolor='black', color='white
↪')
base2 = AREAL_INPUT.data.plot(ax=axes[0, 1], legend=True, edgecolor='black', color='white
↪')
base3 = AREAL_INPUT.data.plot(ax=axes[0, 2], legend=True, edgecolor='black', color='white
↪')
```

(continues on next page)

(continued from previous page)

```
base4 = AREAL_INPUT.data.plot(ax=axes[1, 0], legend=True, edgecolor='black', color='white
↪')
base5 = AREAL_INPUT.data.plot(ax=axes[1, 1], legend=True, edgecolor='black', color='white
↪')
base6 = AREAL_INPUT.data.plot(ax=axes[1, 2], legend=True, edgecolor='black', color='white
↪')

gdf_pts.plot(ax=base1, column=preds_cols[0], cmap='Spectral_r', alpha=0.3)
gdf_pts.plot(ax=base2, column=preds_cols[1], cmap='Spectral_r', alpha=0.3)
gdf_pts.plot(ax=base3, column=preds_cols[2], cmap='Spectral_r', alpha=0.3)
gdf_pts.plot(ax=base4, column=errs_cols[0], cmap='Reds', alpha=0.3)
gdf_pts.plot(ax=base5, column=errs_cols[1], cmap='Reds', alpha=0.3)
gdf_pts.plot(ax=base6, column=errs_cols[2], cmap='Reds', alpha=0.3)

axes[0, 0].set_title(preds_cols[0])
axes[0, 1].set_title(preds_cols[1])
axes[0, 2].set_title(preds_cols[2])
axes[1, 0].set_title(errs_cols[0])
axes[1, 1].set_title(errs_cols[1])
axes[1, 2].set_title(errs_cols[2])

plt.suptitle('Comparison of different Kriging models outputs')
plt.show()
```



Visual inspection shows that:

> The circular model and the exponential model have created the smoothest results. An interesting pattern emerges on uncertainty maps. The variance errors are smoother on the first map (the circular model). The middle map produces a higher variance in output, but it still works as a filter. The last model doesn't perform well, has too strict distance constraints, and the final outcome is not smooth. Sharply increasing variance errors on the uncertainty map of the cubic model tell us that maybe we have chosen the wrong model - we are not able to retrieve meaningful information from it.

Are absolute errors between maps large or not? Let's check it in the last step:

[14]:
```
for pcol1 in preds_cols:
    print('Column:', pcol1)
    for pcol2 in preds_cols:
        if pcol1 == pcol2:
            pass
        else:
            mad = gdf_pts[pcol1] - gdf_pts[pcol2]
            mad = np.abs(np.mean(mad))
            print(f'Mean Absolute Difference with {pcol2} is {mad:.4f}')
    print('')
```

```
Column: p sph 4
Mean Absolute Difference with p sph 8 is 0.2395
Mean Absolute Difference with p sph 16 is 0.6387

Column: p sph 8
Mean Absolute Difference with p sph 4 is 0.2395
Mean Absolute Difference with p sph 16 is 0.3993

Column: p sph 16
Mean Absolute Difference with p sph 4 is 0.6387
Mean Absolute Difference with p sph 8 is 0.3993
```

That differences are tiny, so we shouldn't throw away one model after another. Semivariogram models are comparable. I prefer the **8-lag** theoretical variogram with an exponential model, and it has the best bias-to-variance ratio and a low RMSE error.

Changelog

Date	Change description	Author
2023-08-23	The tutorial was refreshed and set along with the 0.5.0 version of the package	@SimonMolinsky
2023-04-15	Tutorial updated for the 0.4.1 version of the package	@SimonMolinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@SimonMolinsky
2022-10-21	Tutorial updated for the 0.3.4 version of the package	@SimonMolinsky
2022-08-23	Tutorial updated for the 0.3.0 version of the package	@SimonMolinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@SimonMolinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@SimonMolinsky
2021-12-06	Behavior of `prepare_kriging_data()` function has changed	@SimonMolinsky
2021-10-13	Refactored TheoreticalSemivariogram (name change of class attribute) and refactored `calc_semivariance_from_pt_cloud()` functions to protect calculations from `NaN's`.	@ethmtrgt & @SimonMolinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@SimonMolinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@SimonMolinsky

[ ]:

C.1.2 Semivariogram Regularization

Table of Contents:

1. Prepare areal and point data,
2. Set semivariogram parameters,
3. Perform regularization,
4. Visualize regularized semivariogram and analyze algorithm performance,
5. Export semivariogram to json.

Introduction

We will learn how to regularize a semivariogram of a dataset that consists of irregularly shaped polygons. The procedure of the semivariogram regularization is described here: (1) Goovaerts P., Kriging and Semivariogram Deconvolution in the Presence of Irregular Geographical Units, Mathematical Geology 40(1), 101-128, 2008.

The main idea is to retrieve the point support semivariogram from the semivariogram of blocks of different shapes and sizes. This is the case in the mining industry, where aggregated blocks are deconvoluted into smaller units, and in epidemiology, where data is aggregated over big administrative units. Or in ecology, where species observations are aggregated over areas or time windows.

In this tutorial, we use block data of Breast Cancer incidence rates in Northeastern counties of the U.S. and U.S. Census 2010 data for population blocks.

> The breast cancer rates data and the point support population counts are in the geopackage in a directory: `samples/regularization/cancer_data.gpkg`.

Import packages

[1]:
```
from pyinterpolate import build_experimental_variogram, Blocks, PointSupport,␣
↪Deconvolution
```

1) Prepare areal and point data

Data structures for semivariogram regularization are rather complex, and that's why **Pyinterpolate** has classes to prepare this data for processing. There are two structures:

1. `Blocks()`: stores polygon and block data, transforms it, and retrieves centroids,
2. `PointSupport()`: performs spatial joins between polygons and points support data and manages internal indexing of those datasets.

A researcher can prepare data manually - the `Deconvolution` class processes `Dict`, `numpy array`, or `DataFrame` types of objects. For more data types and how to prepare them, we can look at the API documentation.

[2]:
```
DATASET = 'samples/regularization/cancer_data.gpkg'
OUTPUT = 'samples/regularization/regularized_variogram.json'
POLYGON_LAYER = 'areas'
POPULATION_LAYER = 'points'
POP10 = 'POP10'
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'
```

[3]:
```
blocks = Blocks()
blocks.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
↪name=POLYGON_LAYER)

point_support = PointSupport()
point_support.from_files(point_support_data_file=DATASET,
                         blocks_file=DATASET,
                         point_support_geometry_col=GEOMETRY_COL,
                         point_support_val_col=POP10,
```

(continues on next page)

(continued from previous page)

```
                        blocks_geometry_col=GEOMETRY_COL,
                        blocks_index_col=POLYGON_ID,
                        use_point_support_crs=True,
                        point_support_layer_name=POPULATION_LAYER,
                        blocks_layer_name=POLYGON_LAYER)
```

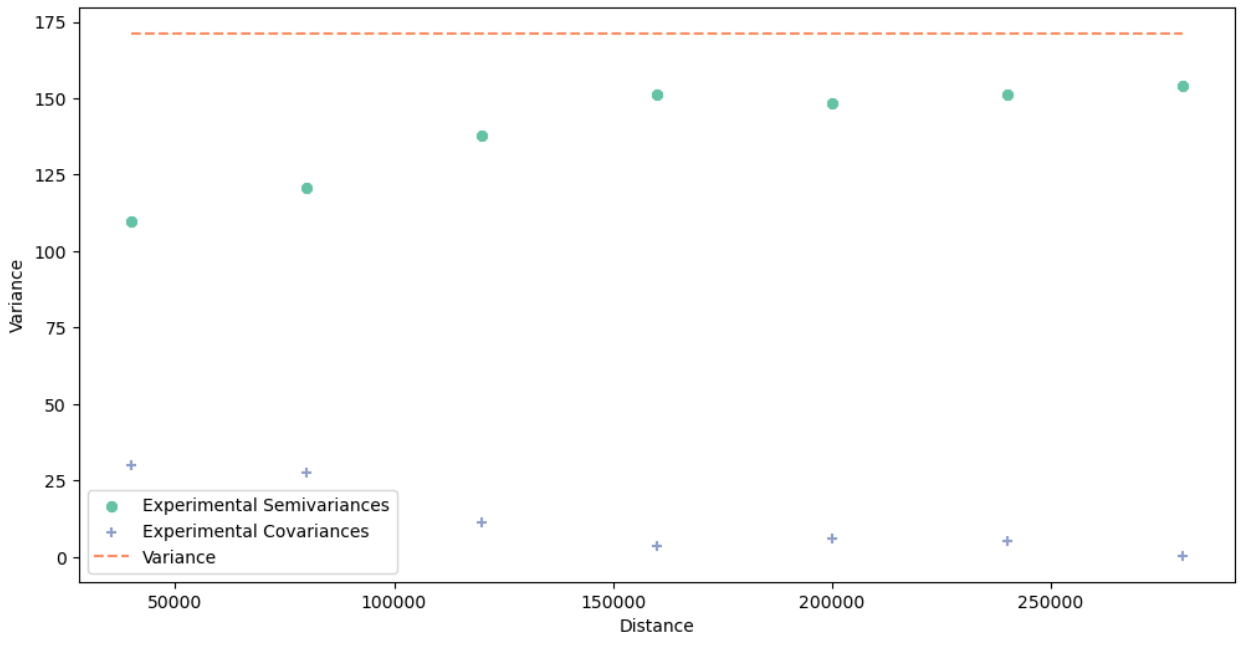
```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

2) Set semivariogram parameters

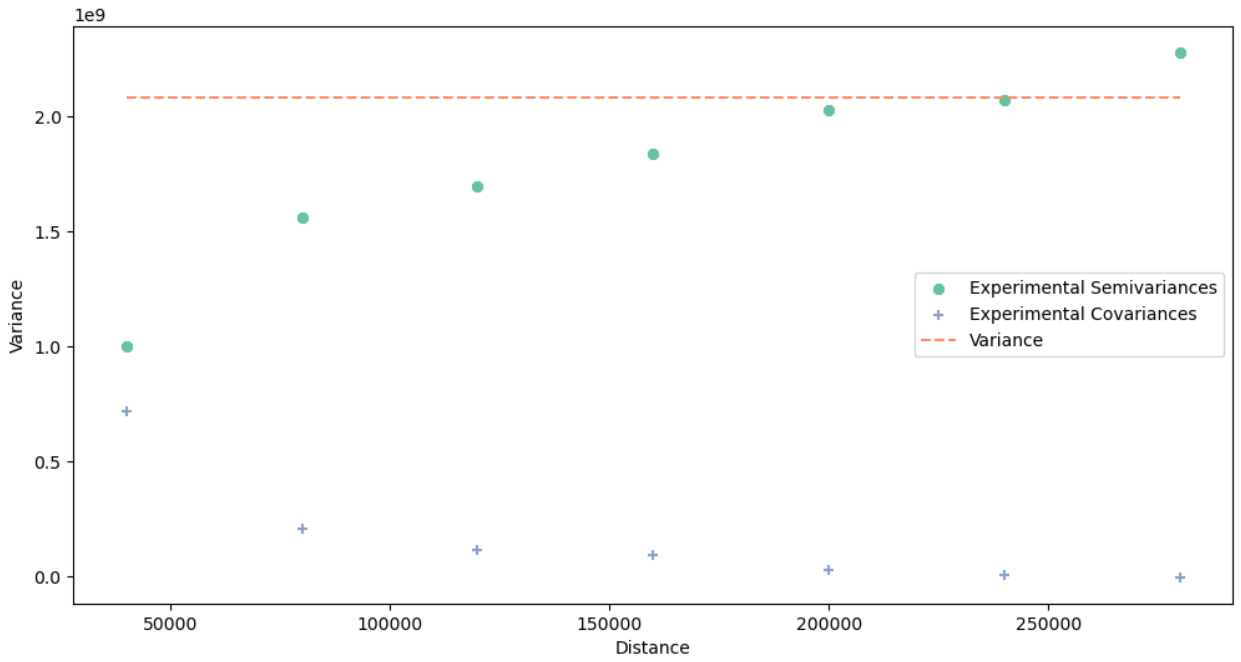
Now, we must set parameters for the areal semivariogram AND point semivariogram. It is essential to understand data well to set the variogram parameters properly. That's why you should always check the experimental semivariograms of block data and its point support. We do it for prepared areal and point datasets.

The *step size* and the *maximum search radius* parameters depend on the block data. But we should check the point's semivariogram too. We won't create meaningful results if point support data is spatially independent.

[4]:
```
# Check experimental semivariogram of areal data - this cell may be run multiple times
# before you find optimal parameters

maximum_range = 300000
step_size = 40000

dt = blocks.data[[blocks.cx, blocks.cy, blocks.value_column_name]]  # x, y, val
exp_semivar = build_experimental_variogram(input_array=dt, step_size=step_size, max_
↪range=maximum_range)

# Plot experimental semivariogram

exp_semivar.plot()
```



[5]:
```
# Check experimental semivariogram of point data - this cell may be run multiple times
# before you find optimal parameters

maximum_point_range = 300000
step_size_points = 20000


pt = point_support.point_support[[point_support.x_col, point_support.y_col, point_
↪support.value_column]].values
exp_semivar = build_experimental_variogram(input_array=pt, step_size=step_size, max_
↪range=maximum_range)

# Plot experimental semivariogram

exp_semivar.plot()
```



We see that block and point support variograms follow a spatial-dependency pattern. In this case, we can move to the next step - deconvolution. The next step is to create the `Deconvolution` object. We have multiple parameters to choose from, and it is hard to find the best fit initially, so try to avoid multiple loops because it is time-consuming.

The program is designed to first `fit()` the model and later to `transform()` it. There is a way to perform both steps simultaneously with the `fit_transform()` method. However, I encourage you to divide the process into two parts because transformation can take a long time to end. It is better to check how the model behaves after the first iteration (`fit()`) rather than wait until the end of processing.

When you `fit()` a model, you have multiple parameters to control, but only a few are necessary for regular purposes (**in bold**): - **agg_dataset**: Blocks with aggregated data. `Blocks()` class object or `GeoDataFrame` and `DataFrame` with columns: `centroid.x, centroid.y, ds, index` or `numpy array`: `[[block index, centroid x, centroid y, value]]`. - **point_support_dataset** : `PointSupport()` object, or `Dict`: `{block id: [[point x, point y, value]]}`, or `numpy array`: `[[block id, x, y, value]]`, or `DataFrame` and `GeoDataFrame`: `columns={x, y, ds, index}`. - **agg_step_size** : Step size between lags. - **agg_max_range** : Maximal distance of analysis. - **agg_nugget** : The nugget of semivariogram. We will set it to 0. - *agg_direction* : Direction of semivariogram, values from 0 to 360 degrees. - *agg_tolerance* : (see docs). - **variogram_weighting_method**

: Method used to weight error at a given lags. Available methods: * equal: no weighting, * **closest**: Lags at a close range have bigger weights, * distant: lags that further away have bigger weights, * dense: error is weighted by the number of point pairs within a lag - more pairs, smaller weight. - **model_name**: the name of the semivariogram function. Two terms aggregate multiple models: `all` and `safe`, and the latter is recommended to use. - **model_types** : semivariogram model types to test. You can pass here a list of theoretical models.

We will weight lags, and we do not store semivariograms and semivariogram models.

After fitting, we perform `transform()`. This function has few parameters to control the regularization process, but we leave them as default with one exception: we set the **max_iters** parameter to 5.

This process of fitting and transforming takes some time, so it's a good idea to run it and do something else...

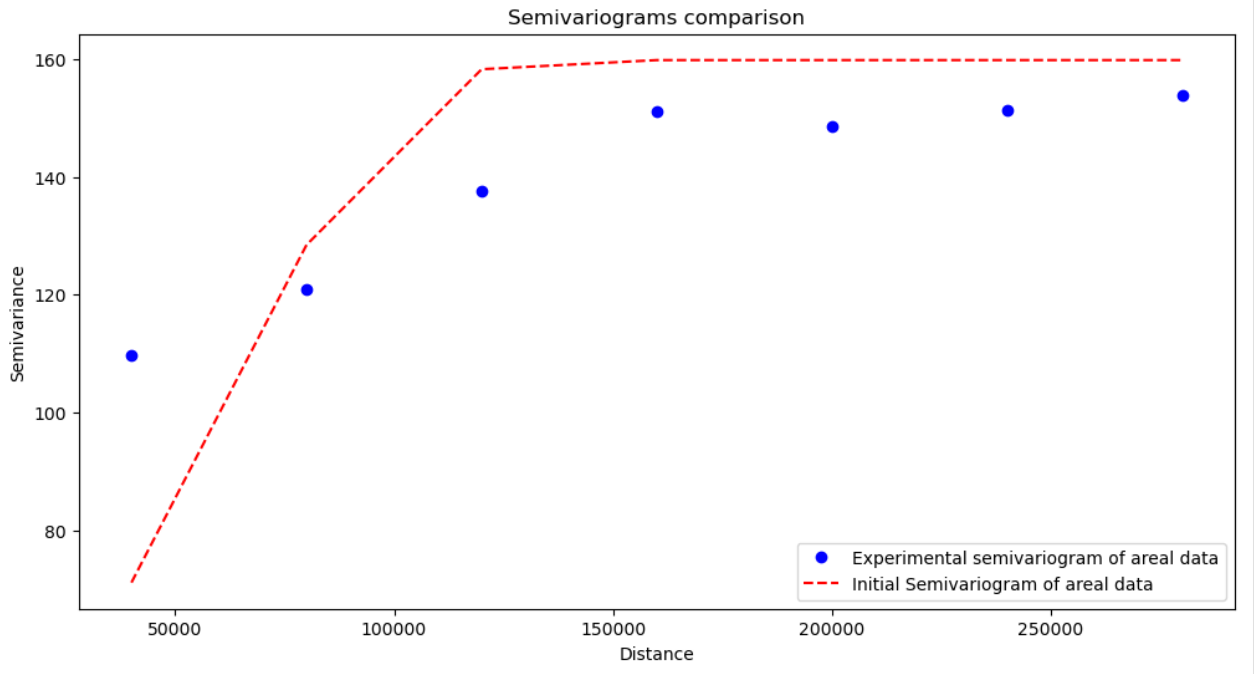
3) Regularize semivariogram

[6]:
```
reg_mod = Deconvolution(verbose=True)
```

[7]:
```
reg_mod.fit(agg_dataset=blocks,
            point_support_dataset=point_support,
            agg_step_size=step_size,
            agg_max_range=maximum_range,
            agg_nugget=0.,
            variogram_weighting_method='closest',
            model_name='safe')
```

```
Regularization fit process starts
Regularization fit process ends
```

[8]:
```
# Check initial experimental, theoretical and regularized semivariograms

reg_mod.plot_variograms()
```



After the first step, we see that the variogram can be regularized. There is a big difference between the theoretical curve and the experimental values. Let's run the transformation process.

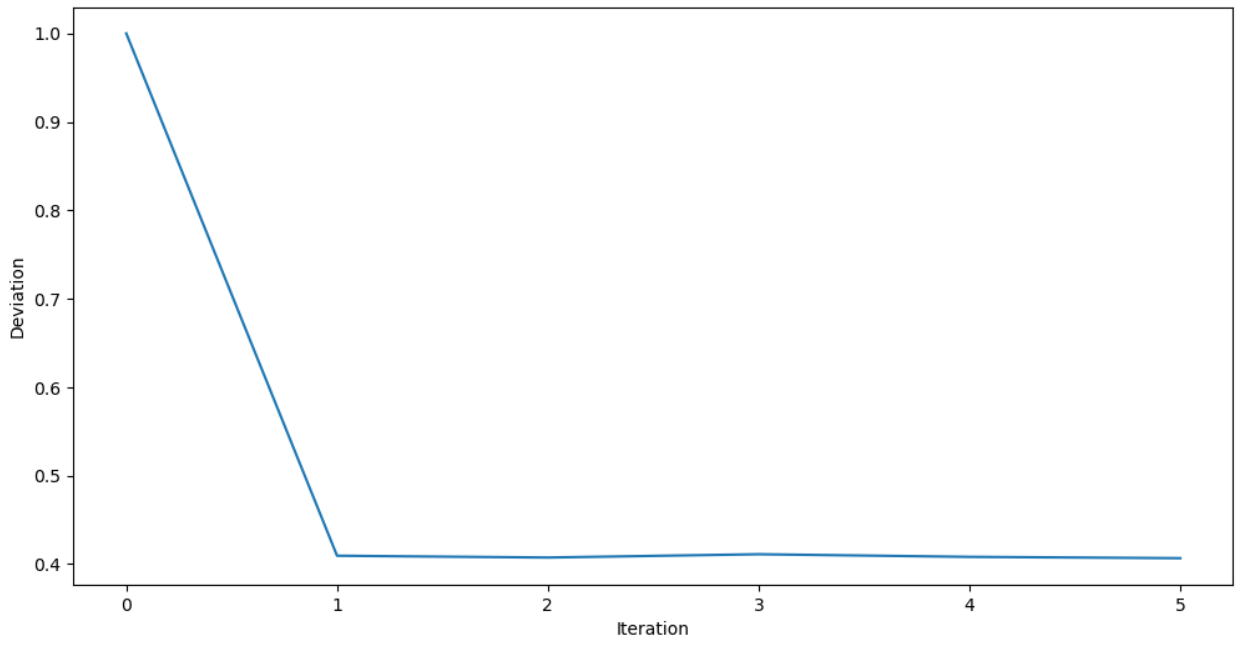
[9]:
```
reg_mod.transform(max_iters=5)
```

```
Transform procedure starts
100%|| 5/5 [00:18<00:00,  3.74s/it]
```

4) Visualize and check semivariogram

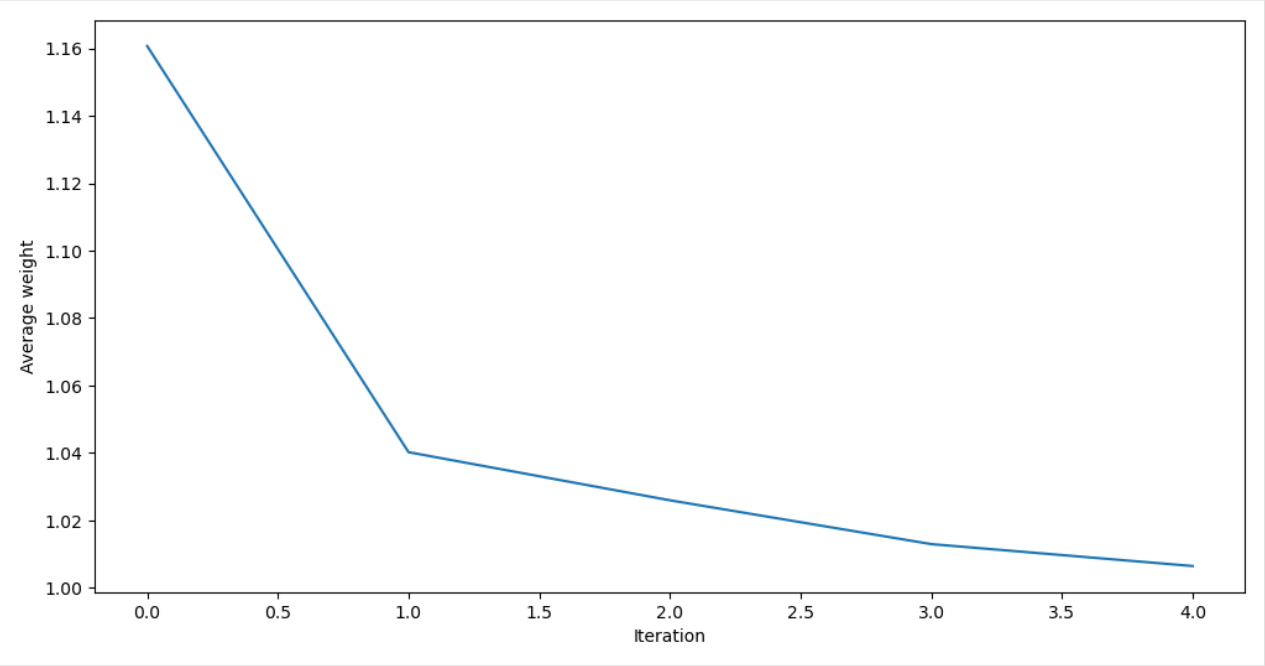
The process is fully automatic, but we can check how it behaved through each iteration. We can analyze deviation change (the most important variable, the mean absolute difference between regularized and theoretical models) with the built-in method. Still, if you are more interested in the algorithm stability, you can also analyze weight change over each iteration.

[10]:
```
# First analyze deviation

reg_mod.plot_deviations()
```



Clarification: The deviation between a regularized semivariogram and a theoretical model decreases. However, a significant improvement can be seen after the first step, and then it slows down. That's why semivariogram regularization usually does not require many steps. However, if there are many lags, optimization may require more steps to achieve a meaningful result.

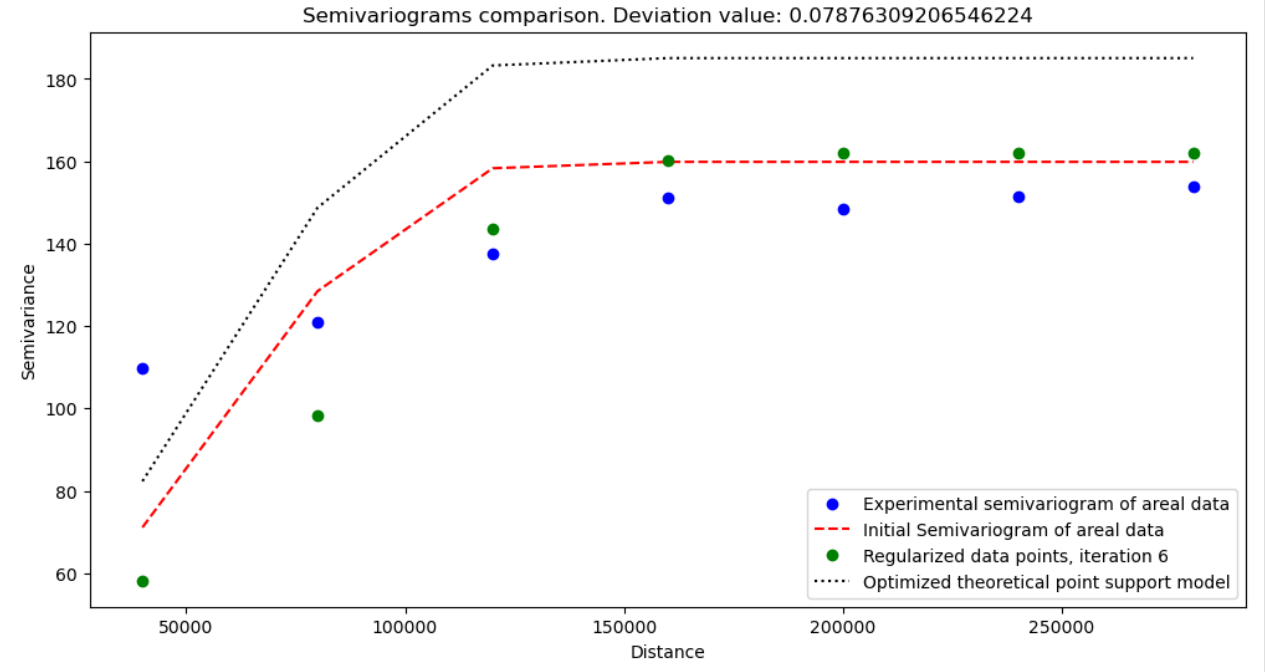
[11]:
```
# Check weights - it is important to track problems with algorithm, esepcially if sum of␣
↪weights is oscillating
# then it may be a sign of problems with data, model or (hopefully not!) algorithm␣
↪itself.

reg_mod.plot_weights()
```



NOTE: Weights are smaller with each iteration. It is the expected behavior of the algorithm. The general trend goes downward, and small oscillations may occur due to the optimization process.

The most important part is to compare semivariograms! You can see that the Regularized Model is different from the initial semivariogram. It is the result of regularization. The population (point support) is considered in the cases of variogram development and further kriging interpolation.

[12]:
```
reg_mod.plot_variograms()
```



The regularized model works well for our dataset and follows general trends. It assigns smaller weights for closest

lags, and for more significant distances, weights are bigger. It is related to the semivariogram weighting method - we penalize more poorly performing models for the shortest distances from the origin.

5) Export semivariogram to text file

[13]:
```
# Export semivariogram to the text file. This is important step because calculations are␣
↪slow...
# and it is better to not repeat them.
# We will use built-in method: export_model() where we pass only filename and our␣
↪semivariogram
# parameters are stored for other tasks.
reg_mod.export_model('output/regularized_model.json')
```

Where to go from here?

- C.1.3 Poisson Kriging Centroid-based
- C.1.4 Poisson Kriging Area to Area
- C.1.5 Poisson Kriging Area to Point

Changelog

Date	Change description	Author
2023-08-24	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-04-15	Tutorial debugged and updated to the 0.4.1 version of the package	@Simon-Molinsky
2022-11-05	Tutorial updated for the 0.3.5 version of the package	@Simon-Molinsky
2022-10-08	Update for the 0.3.4 version of the package	@Simon-Molinsky
2022-10-17	Updated variogram models selection (0.3.3 version of the package)	@Simon-Molinsky
2022-10-08	Update for the 0.3.2 version of the package	@Simon-Molinsky
2022-08-27	Update for the 0.3.0 version of the package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-05-28	Updated paths for input/output data	@Simon-Molinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@Simon-Molinsky
2021-04-04	Ranges parameter removed from regularize semivariogram class	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@Simon-Molinsky

[13]:

C.1.3 Poisson Kriging - centroid based approach

Table of Contents:

1. Load areal and point data,
2. Load semivariogram (regularized),
3. Remove 10% of areal dataset,
4. Predict values at unknown locations,
5. Analyse Forecast Bias and Root Mean Squared Error of prediction.

Introduction

Before starting this tutorial, be sure you have understood concepts in the **Ordinary and Simple Kriging** and **Semivariogram Regularization** tutorials.

The Poisson Kriging technique is used to model spatial count data. We will analyze a particular case where values are counted over blocks, and those blocks may have irregular shapes and sizes. We will try to predict the breast cancer rates in the Northeastern counties of the U.S. We will use U.S. Census 2010 data to create point support for blocks.

> The breast cancer rates data and the point support population counts are located in the geopackage in a directory: `samples/regularization/cancer_data.gpkg`

Even if our areal data has units with irregular shapes and sizes, for the case of this tutorial, we assume that each region may be represented by its centroid. This is a huge simplification; we can do it at our own risk. There are cases when we can use this method instead of the Area-to-Area Kriging. Those cases are:

- We must do our calculations fast, and the processing time is crucial,
- or blocks have similar shapes and sizes.

Centroid-based Poisson Kriging is much faster than Area-to-Area or Area-to-Point Poisson Kriging, and for some applications, it may be the desired property.

The tutorial covers the following steps:

1. Read and explore data,
2. Load semivariogram model,
3. Prepare training and test data,
4. Predict values of unknown locations and calculate forecast bias and root mean squared error,
5. Analyze error metrics.

1) Read and explore data

[1]:
```
import numpy as np
import pandas as pd

from pyinterpolate import TheoreticalVariogram
from pyinterpolate import Blocks, PointSupport
from pyinterpolate import centroid_poisson_kriging
```

[2]:
```
DATASET = 'samples/regularization/cancer_data.gpkg'
OUTPUT = 'samples/regularization/regularized_variogram.json'
POLYGON_LAYER = 'areas'
POPULATION_LAYER = 'points'
POP10 = 'POP10'
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'

blocks = Blocks()
blocks.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
↪name=POLYGON_LAYER)
```

(continues on next page)

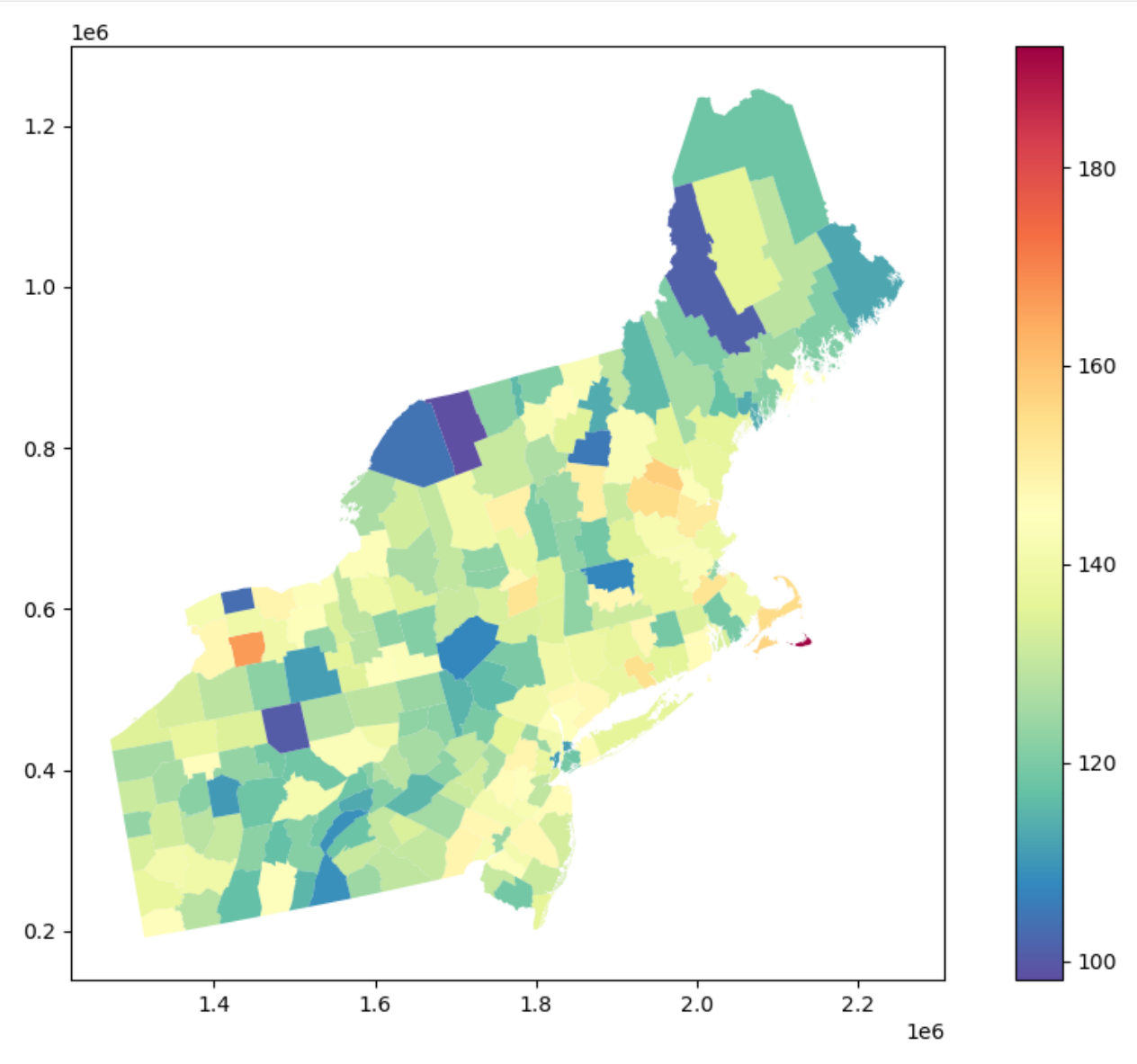
(continued from previous page)

```
point_support = PointSupport()
point_support.from_files(point_support_data_file=DATASET,
                         blocks_file=DATASET,
                         point_support_geometry_col=GEOMETRY_COL,
                         point_support_val_col=POP10,
                         blocks_geometry_col=GEOMETRY_COL,
                         blocks_index_col=POLYGON_ID,
                         use_point_support_crs=True,
                         point_support_layer_name=POPULATION_LAYER,
                         blocks_layer_name=POLYGON_LAYER)
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

[3]:
```
# Lets take a look into a map of areal counts

blocks.data.plot(column=blocks.value_column_name, cmap='Spectral_r', legend=True,␣
↪figsize=(12, 8))
```

[3]:
```
<Axes: >
```



Clarification: It is a good idea to look into the spatial patterns in a dataset and to visually check if our data do not have any `NaN` values. We use the geopandas `GeoDataFrame.plot()` function with a color map that diverges regions based on the cancer incidence rates. The output choropleth map is not ideal, and (probably) it has a few unreliable results, for example:

- In counties with a small population, where the ratio number of cases to population size is high even if the number of cases is low,
- Counties that are very big and sparsely populated may draw more of our attention than densely populated counties,
- Transitions of colors (rates) between counties may be too abrupt, even if we know that neighboring counties should have closer results.

We can overcome those problems using Centroid-based Poisson Kriging for filtering and denoising. But we must be aware that this method introduces a bias: each polygon falls into its centroid, but polygons' shapes and sizes differ, so centroid representation erases spatial patterns.

2) Load semivariogram model

We load a regularized semivariogram from the tutorial about **Semivariogram Regularization**. You can always perform semivariogram regularization along with the Poisson Kriging, but it is a very long process, and it is more convenient to separate those two steps.

[4]:
```
semivariogram = TheoreticalVariogram()  # Create TheoreticalSemivariogram object
semivariogram.from_json('output/regularized_model.json')  # Load regularized␣
↪semivariogram
```

3) Prepare training and test data.

We simply remove 40% of random ID's from the areal dataset (the same for points) and create four arrays: two training arrays with areal and point geometry and values and two test arrays with the same categories of data.

[5]:
```
# Remove 40% of rows (values) to test our model

def create_test_areal_set(areal_dataset: Blocks, points_dataset: PointSupport, frac=0.4):
    """

    Parameters
    ----------
    areal_dataset : Blocks
    points_dataset : PointSupport
    frac : float

    Returns
    -------
    : List
        [[training_areas, test_areas, training_points, test_points]]
            equal to:
        [np.ndarray, np.ndarray, np.ndarray, np.ndarray]
            where:
        - areas: [[block index, centroid x, centroid y, value]]
        - point support: [[block id, x, y, value]]
    """
    block_id_col = areal_dataset.index_column_name
    block_data = areal_dataset.data.copy()
    all_ids = block_data[block_id_col].unique()
    training_set_size = int(len(all_ids) * (1 - frac))

    training_ids = np.random.choice(all_ids,
                                    size=training_set_size,
                                    replace=False)

    training_areas = block_data[block_data[block_id_col].isin(training_ids)]
    training_areas = training_areas[[block_id_col, 'centroid_x', 'centroid_y', areal_
↪dataset.value_column_name]].values
    test_areas = block_data[~block_data[block_id_col].isin(training_ids)]
    test_areas = test_areas[[block_id_col, 'centroid_x', 'centroid_y', areal_dataset.
↪value_column_name]].values
```

(continues on next page)

(continued from previous page)

```
    ps_data = points_dataset.point_support.copy()
    ps_ids = points_dataset.block_index_column

    training_points = ps_data[ps_data[ps_ids].isin(training_ids)]
    training_points = training_points[[ps_ids,
                                       points_dataset.x_col,
                                       points_dataset.y_col,
                                       points_dataset.value_column]].values
    test_points = ps_data[~ps_data[ps_ids].isin(training_ids)]
    test_points = test_points[[ps_ids,
                               points_dataset.x_col,
                               points_dataset.y_col,
                               points_dataset.value_column]].values


    output = [training_areas, test_areas, training_points, test_points]
    return output

ar_train, ar_test, pt_train, pt_test = create_test_areal_set(blocks, point_support)
```

4) Predict values at unknown locations and calculate forecast bias and root mean squared error.

You may start to work with predictions with a fitted semivariogram model. Centroid-based Poisson Kriging takes five arguments during the run:

- semivariogram model (fitted semivariogram model),
- known areas (training set),
- known points (training set),
- unknown block (without rate value),
- unknown block's point support.

Additional two parameters control the process of interpolation:

- **number of observations** (the most critical parameter - how many neighbors are affecting your area of analysis),
- **weighted** by population (bool parameter, if True, then distances are weighted by population between points - default is True for Poisson Kriging).

The algorithm in the cell below iteratively picks one area from the test set and performs prediction. Then, forecast bias, the difference between actual and predicted value, is calculated. Forecast Bias clarifies if our algorithm predictions are too low or too high (under- or overestimation). The following parameter is the Root Mean Squared Error. This measure tells us more about outliers and huge differences between predictions and actual values. We will see it in the last part of the tutorial.

Your work with Poisson Kriging (or Kriging) will usually be the same: - Prepare training and test data, - use training data to train the semivariogram model, - Test different hyperparameters with a test set to find the optimal number of neighbors that are affecting your area of analysis, - predict values at unseen locations OR perform data smoothing.

[6]:
```
number_of_obs = 4

predslist = []
for unknown_area in ar_test:
```

(continues on next page)

(continued from previous page)

```
    upts = pt_test[pt_test[:, 0] == unknown_area[0]]
    upts = upts[:, 1:]
    try:
        kriging_preds = centroid_poisson_kriging(semivariogram_model=semivariogram,
                                                 blocks=ar_train,
                                                 point_support=pt_train,
                                                 unknown_block=unknown_area[:-1],
                                                 unknown_block_point_support=upts,
                                                 number_of_neighbors=number_of_obs,
                                                 is_weighted_by_point_support=True,
                                                 raise_when_negative_prediction=True,
                                                 raise_when_negative_error=True)
    except ValueError as _:
        predslist.append([unknown_area[0], np.nan, np.nan, np.nan, np.nan])
    else:
        rmse = np.sqrt((kriging_preds[1] - unknown_area[-1])**2)
        fb = unknown_area[-1] - kriging_preds[1]
        kriging_preds.extend([rmse, fb])
        predslist.append(kriging_preds)
```

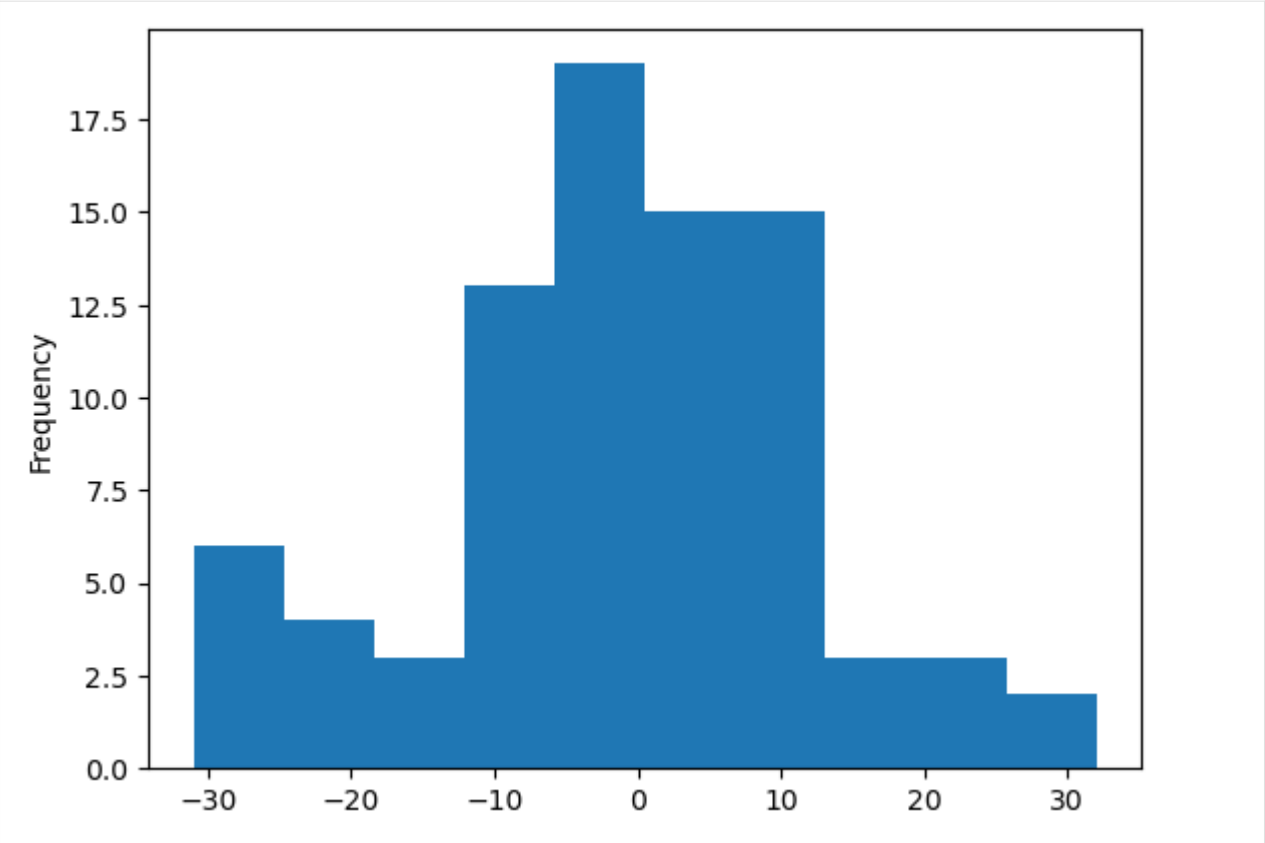
[7]:
```
# [unknown block index, prediction, error, rmse, fb]
predslist = np.array(predslist)
pred_df = pd.DataFrame(data=predslist[:, 1:],
                       index=predslist[:, 0],
                       columns=['predicted', 'err', 'rmse', 'fb'])  # Store results in␣
↪DataFrame
```

5) Analyze Forecast Bias and Root Mean Squared Error of prediction

The analysis of errors is the last analysis step. We plot two histograms: forecast bias and root mean squared error then we calculate the base statistics of a dataset.

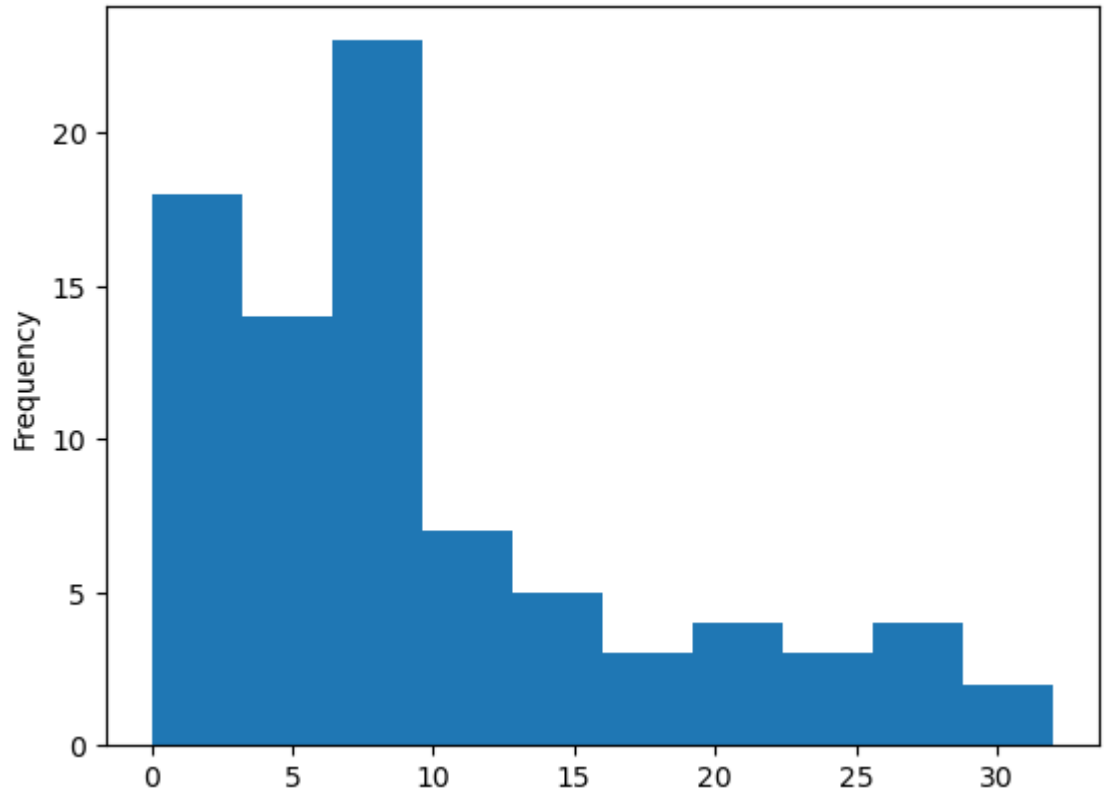
[8]:
```
# Show histograms of errors

pred_df['fb'].plot.hist(bins=10)
```

[8]:
```
<Axes: ylabel='Frequency'>
```



[9]:
```
pred_df['rmse'].plot.hist(bins=10)
```

[9]:
```
<Axes: ylabel='Frequency'>
```



[10]:
```
pred_df.describe()
```

[10]:

	predicted	err	rmse	fb
count	83.000000	83.000000	83.000000	83.000000
mean	132.867622	8.825592	9.745322	-1.017020
std	10.440986	2.751169	8.013841	12.621576
min	105.681428	1.156868	0.002419	-30.980696
25%	125.629770	7.266978	4.105316	-8.052315
50%	133.459074	8.893322	7.967837	-0.281428
75%	138.548781	10.717209	12.820111	7.643090
max	161.959004	14.974212	32.017309	32.017309

Clarification: Analysis of **Forecast Bias** and **Root Mean Squared Error** - their distribution and basic properties - could be a handy tool to analyze model performance. However, consider that the table above is a single test case (realization) and can be misleading. The good idea is to repeat the test dozens of times with a different training/test set division each time. After this, we average results from multiple tests and get insight into our model's behavior.

Note 1: Those results are not decisive. Our sample has been selected randomly, and there is a chance that it is not a spatially representative sample! (E.g., areas only from one region). The good idea is to repeat the experiment multiple times with other samples and average results to determine how well the model performs.

Note 2: If we analyze errors' statistics, we should consider not only an error's *mean* value. Let's look at different pieces of information:

- Histograms clearly show us how dispersed and grouped errors are, and most importantly, we directly see the worst predictions and how many of them are generated by our model.
- What is plotted on a histogram is described by statistics. We may check the max and the min error, but the true power comes when we analyze quartiles and standard deviation.

- The standard deviation is a good measurement of our model's variance; the less, the better.
- Sometimes, we must look into quartiles. A very high mean but relatively low median (or even the 3rd quartile) indicates that we have only a few wrong predictions, most of which are acceptable.

Where to go from here?

- C.1.4 Poisson Kriging Area to Area
- C.1.5 Poisson Kriging Area to Point

Changelog

Date	Change description	Author
2023-08-25	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-01-16	New centroid-based PK algorithm	@Simon-Molinsky
2022-10-21	Tutorial updated for the 0.3.4 version of the package	@Simon-Molinsky
2022-08-27	Tutorial updated for the 0.3.0 version of the package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-12-11	Behavior of `prepare_kriging_data()` function has benn changed	@Simon-Molinsky
2021-05-28	Updated paths for input/output data	@Simon-Molinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@Simon-Molinsky

[10]:

Poisson Kriging - Area to Area Kriging

Table of Contents:

1. Load areal and point data,
2. Load semivariogram (regularized),
3. Remove 40% of areal dataset,
4. Predict values at unknown locations,
5. Analyse Forecast Bias and Root Mean Squared Error of prediction.

Introduction

Before starting this tutorial, be sure you have understood concepts in the **Ordinary and Simple Kriging** and **Semivariogram Regularization** tutorials.

The Poisson Kriging technique is used to model spatial count data. We will analyze a particular case where values are counted over blocks, and those blocks may have irregular shapes and sizes. We will try to predict the breast cancer rates in the Northeastern counties of the U.S. We will use U.S. Census 2010 data to create point support for blocks.

> The breast cancer rates data and the point support population counts are located in the geopackage in a directory: `samples/regularization/cancer_data.gpkg`

In tutorial **Poisson Kriging - Centroid based approach**, we've assumed that all areas have the same size and shape. It is not a valid statement. We now use the Area-to-Area Kriging technique to predict and evaluate missing values.

Area-to-Area and Area-to-Point Poisson Kriging are slower than simplified Kriging with Centroids but give more reliable results because they are tuned to areal size and shape.

This tutorial covers the following steps:

1. Read and explore data,
2. Load semivariogram model,
3. Prepare training and test data,
4. Predict values of unknown locations and calculate forecast bias and root mean squared error,
5. Analyze error metrics.

1) Read and explore data

[1]:
```
import numpy as np
import pandas as pd

from pyinterpolate import TheoreticalVariogram
from pyinterpolate import Blocks, PointSupport
from pyinterpolate import area_to_area_pk
```

[2]:
```
DATASET = 'samples/regularization/cancer_data.gpkg'
OUTPUT = 'samples/regularization/regularized_variogram.json'
POLYGON_LAYER = 'areas'
POPULATION_LAYER = 'points'
POP10 = 'POP10'
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'

blocks = Blocks()
blocks.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
↪name=POLYGON_LAYER)

point_support = PointSupport()
point_support.from_files(point_support_data_file=DATASET,
                         blocks_file=DATASET,
                         point_support_geometry_col=GEOMETRY_COL,
```

(continues on next page)

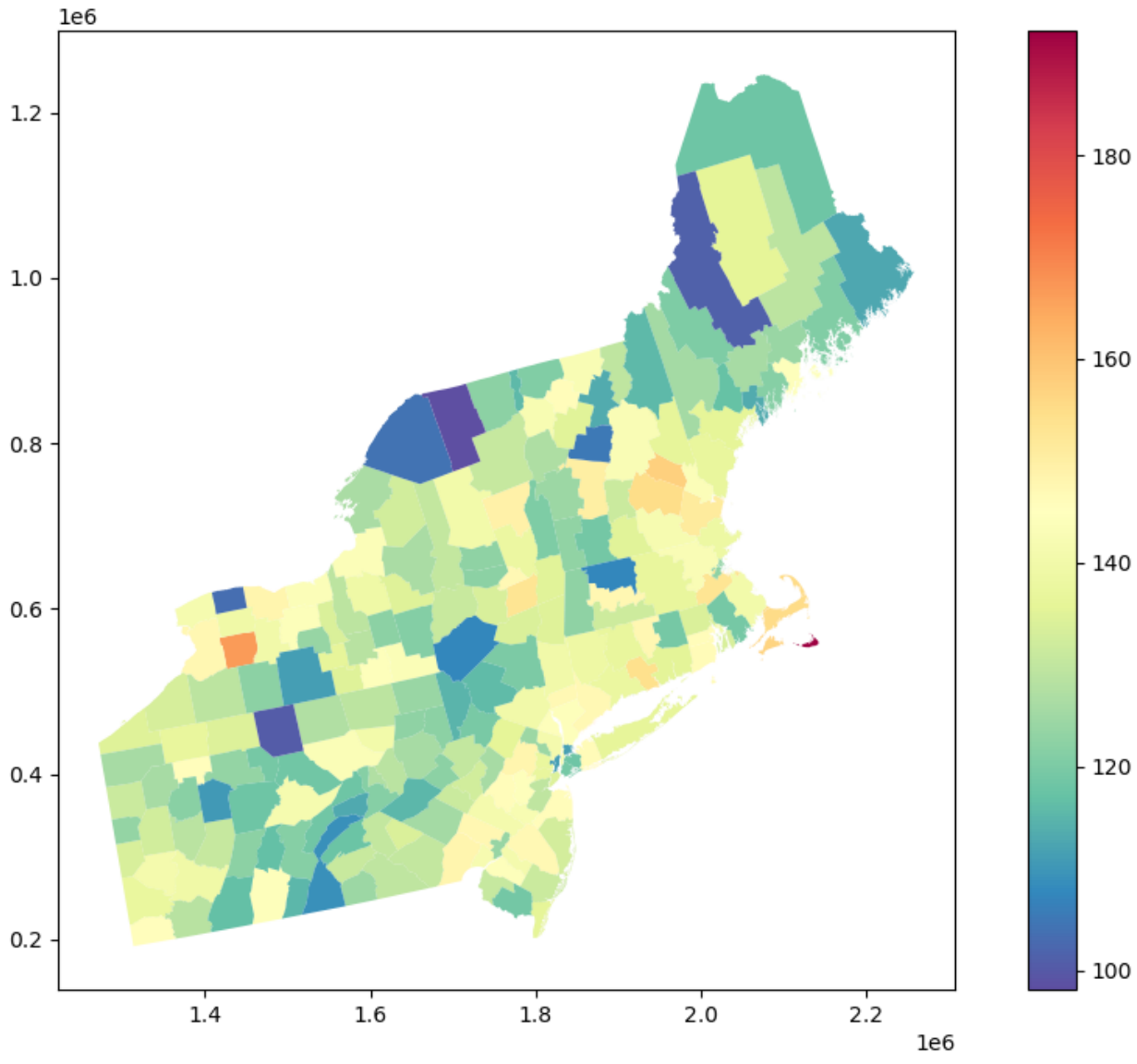
(continued from previous page)

```
                        point_support_val_col=POP10,
                        blocks_geometry_col=GEOMETRY_COL,
                        blocks_index_col=POLYGON_ID,
                        use_point_support_crs=True,
                        point_support_layer_name=POPULATION_LAYER,
                        blocks_layer_name=POLYGON_LAYER)
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

[3]:
```
# Lets take a look into a map of areal counts

blocks.data.plot(column=blocks.value_column_name, cmap='Spectral_r', legend=True,␣
↪figsize=(12, 8))
```

[3]:
```
<Axes: >
```



Clarification: It is a good idea to investigate the spatial patterns in a dataset and to visually check if our data do not have any NaN values. We use the geopandas `GeoDataFrame.plot()` function with a color map that diverges regions based on the cancer incidence rates. The output choropleth map is not ideal, and (probably) it has a few unreliable results, for example:

- In counties with a small population, where the ratio number of cases to population size is high even if the number of cases is low,
- Counties that are very big and sparsely populated may draw more of our attention than densely populated counties,
- Transitions of colors (rates) between counties may be too abrupt, even if we know that neighboring counties should have closer results.

We can overcome those problems using Area-to-Area Kriging for filtering and denoising. But, in this tutorial, we will interpolate with Area-to-Area Kriging and analyze root mean squared errors of predictions and their bias.

2) Load a semivariogram model

We load a regularized semivariogram from the tutorial about **Semivariogram Regularization**. You can always perform semivariogram regularization along with the Poisson Kriging, but it is a very long process, and it is more convenient to separate those two steps.

[4]:
```
semivariogram = TheoreticalVariogram()  # Create TheoreticalSemivariogram object
semivariogram.from_json('output/regularized_model.json')  # Load regularized␣
↪semivariogram
```

3) Prepare training and test data.

We simply remove 40% of random ID's from the areal dataset (the same for points) and create four arrays: two training arrays with areal and point geometry and values and two test arrays with the same categories of data.

[5]:
```
# Remove 40% of rows (values) to test our model

def create_test_areal_set(areal_dataset: Blocks, points_dataset: PointSupport, frac=0.4):
    """

    Parameters
    ----------
    areal_dataset : Blocks
    points_dataset : PointSupport
    frac : float

    Returns
    -------
    : List
        [[training_areas, test_areas, training_points, test_points]]
            equal to:
        [np.ndarray, np.ndarray, np.ndarray, np.ndarray]
            where:
        - areas: [[block index, centroid x, centroid y, value]]
        - point support: [[block id, x, y, value]]
    """
    block_id_col = areal_dataset.index_column_name
```

(continues on next page)

(continued from previous page)

```
    block_data = areal_dataset.data.copy()
    all_ids = block_data[block_id_col].unique()
    training_set_size = int(len(all_ids) * (1 - frac))

    training_ids = np.random.choice(all_ids,
                                    size=training_set_size,
                                    replace=False)

    training_areas = block_data[block_data[block_id_col].isin(training_ids)]
    training_areas = training_areas[[block_id_col, 'centroid_x', 'centroid_y', areal_
↪dataset.value_column_name]].values
    test_areas = block_data[~block_data[block_id_col].isin(training_ids)]
    test_areas = test_areas[[block_id_col, 'centroid_x', 'centroid_y', areal_dataset.
↪value_column_name]].values

    ps_data = points_dataset.point_support.copy()
    ps_ids = points_dataset.block_index_column

    training_points = ps_data[ps_data[ps_ids].isin(training_ids)]
    training_points = training_points[[ps_ids,
                                       points_dataset.x_col,
                                       points_dataset.y_col,
                                       points_dataset.value_column]].values
    test_points = ps_data[~ps_data[ps_ids].isin(training_ids)]
    test_points = test_points[[ps_ids,
                               points_dataset.x_col,
                               points_dataset.y_col,
                               points_dataset.value_column]].values


    output = [training_areas, test_areas, training_points, test_points]
    return output

ar_train, ar_test, pt_train, pt_test = create_test_areal_set(blocks, point_support)
```

4) Predict values at unknown locations and calculate forecast bias and root mean squared error.

You may start to work with predictions with a fitted semivariogram model. Centroid-based Poisson Kriging takes five arguments during the run:

- semivariogram model (fitted semivariogram model),
- known areas (training set),
- known points (training set),
- unknown block (without rate value),
- unknown block's point support.

Additional two parameters control the process of interpolation:

- **number of observations** (the most critical parameter - how many neighbors are affecting your area of analysis),

- **weighted** by population (bool parameter, if True, then distances are weighted by population between points - default is True for Poisson Kriging).

The algorithm in the cell below iteratively picks one area from the test set and performs prediction. Then, forecast bias, the difference between actual and predicted value, is calculated. Forecast Bias clarifies if our algorithm predictions are too low or too high (under- or overestimation). The following parameter is the Root Mean Squared Error. This measure tells us more about outliers and huge differences between predictions and actual values. We will see it in the last part of the tutorial.

Your work with Poisson Kriging (or Kriging) will usually be the same: - Prepare training and test data, - use training data to train the semivariogram model, - Test different hyperparameters with a test set to find the optimal number of neighbors that are affecting your area of analysis, - predict values at unseen locations OR perform data smoothing.

[6]:
```
number_of_obs = 6

predslist = []
for unknown_area in ar_test:
    upts = pt_test[pt_test[:, 0] == unknown_area[0]]
    upts = upts[:, 1:]
    # [unknown block index, prediction, error]
    try:
        kriging_preds = area_to_area_pk(
            semivariogram_model=semivariogram,
            blocks=ar_train,
            point_support=pt_train,
            unknown_block=unknown_area[:-1],
            unknown_block_point_support=upts,
            number_of_neighbors=number_of_obs,
            raise_when_negative_prediction=True,
            raise_when_negative_error=True
        )
    except ValueError:
        predslist.append([unknown_area[0], np.nan, np.nan, np.nan, np.nan])
    else:
        rmse = np.sqrt((kriging_preds[1] - unknown_area[-1])**2)
        fb = unknown_area[-1] - kriging_preds[1]
        kriging_preds.extend([rmse, fb, unknown_area[-1]])
        predslist.append(kriging_preds)
```

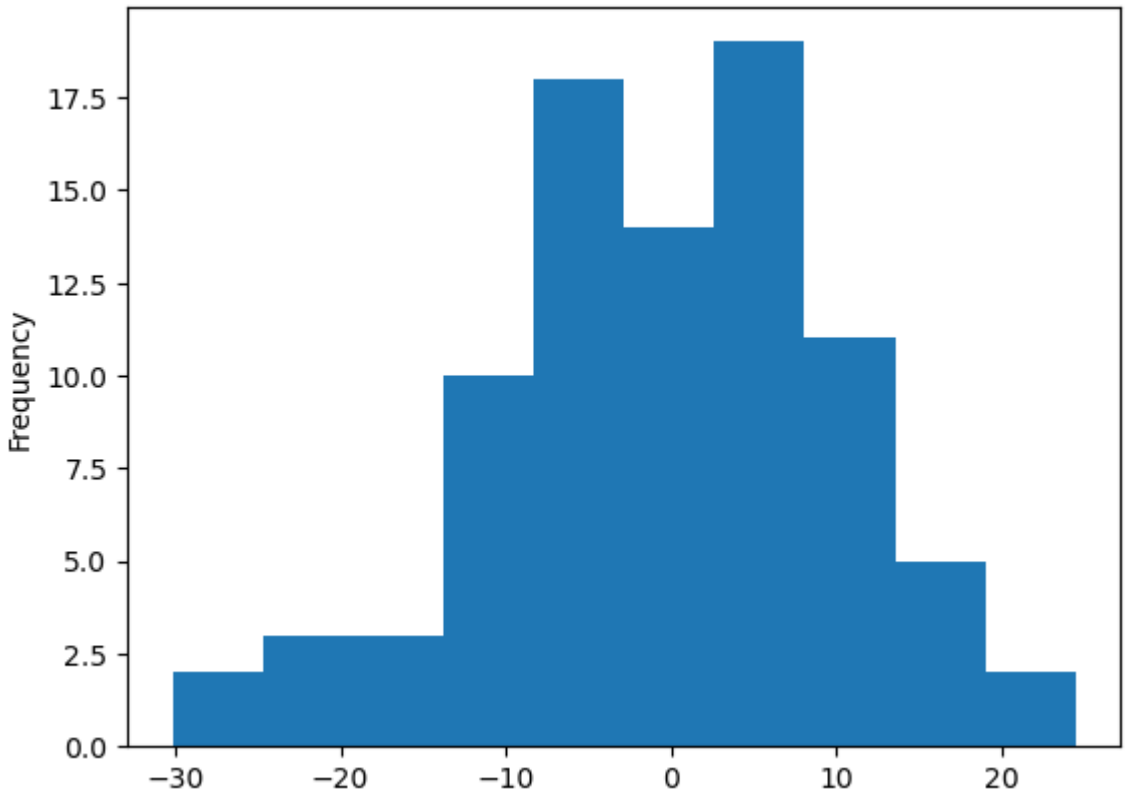
[7]:
```
kriged_predictions = np.array(predslist)
pred_df = pd.DataFrame(data=kriged_predictions[:, 1:],
                       index=kriged_predictions[:, 0],
                       columns=['predicted', 'err', 'rmse', 'fb', 'real'])  # Store␣
↪results in DataFrame
```

5) Analyze Forecast Bias and Root Mean Squared Error of prediction

The last step is the analysis of errors. We plot two histograms: forecast bias and root mean squared error then we calculate the base statistics of a dataset.

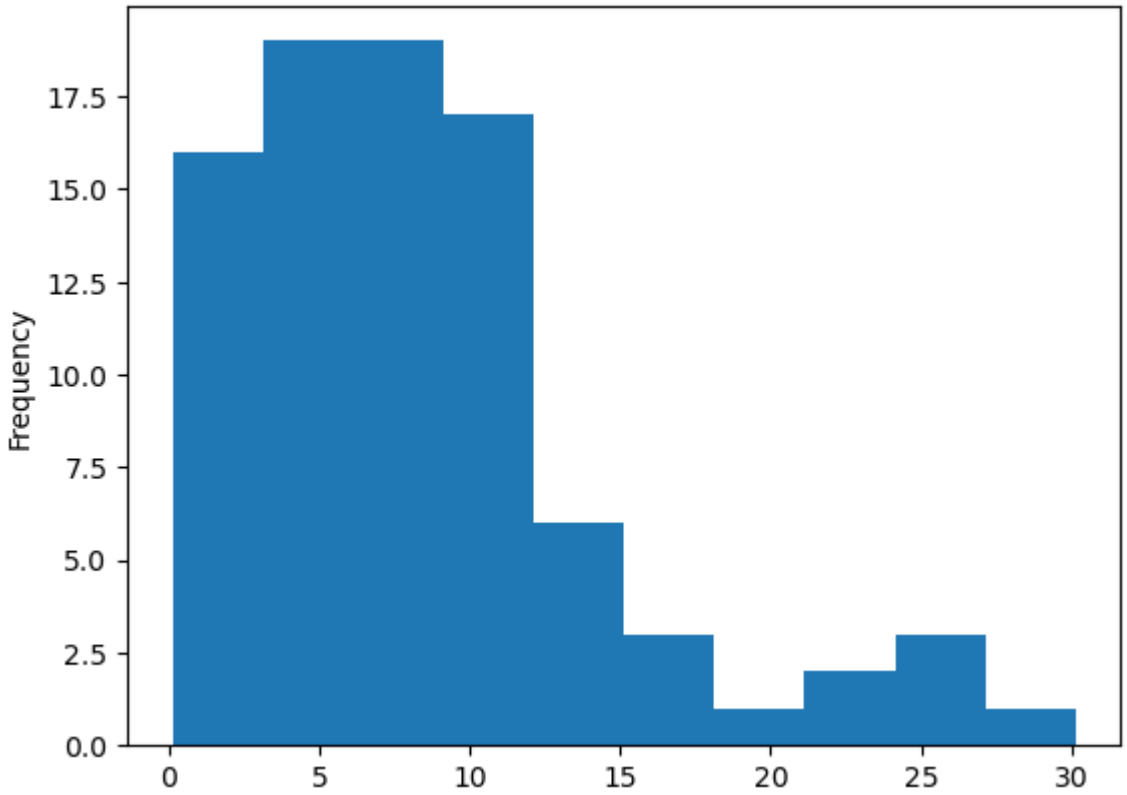
[8]:
```
# Show histograms of errors

pred_df['fb'].plot.hist(bins=10)
```

[8]:
```
<Axes: ylabel='Frequency'>
```



[9]:
```
pred_df['rmse'].plot.hist(bins=10)
```

[9]:
```
<Axes: ylabel='Frequency'>
```



[10]:
```
pred_df.describe()
```

[10]:

	predicted	err	rmse	fb	real
count	87.000000	87.000000	87.000000	87.000000	87.000000
mean	131.695080	9.173880	8.432769	-0.656000	131.039080
std	7.766279	1.247393	6.342679	10.570369	11.372287
min	118.399540	5.898166	0.117618	-30.138693	98.100000
25%	126.024124	8.471337	4.251209	-6.904058	122.100000
50%	131.509964	9.102989	6.860562	-0.349700	131.500000
75%	135.699627	9.913730	10.960136	6.406806	140.000000
max	157.267297	12.248617	30.138693	24.448021	155.300000

Clarification: Analysis of **Forecast Bias** and **Root Mean Squared Error** - their distribution and basic properties - could be a handy tool to analyze model performance. However, consider that the table above is a single test case (realization) and can be misleading. The good idea is to repeat the test dozens of times with a different training/test set division each time. After this, we average results from multiple tests and get insight into our model's behavior.

Note 1: Those results are not decisive. Our sample has been selected randomly, and there is a chance that it is not a spatially representative sample! (E.g., areas only from one region). The good idea is to repeat the experiment multiple times with other samples and average results to determine how well the model performs.

Note 2: If we analyze errors' statistics, we should consider not only an error's *mean* value. Let's look at different pieces of information:

- Histograms clearly show us how dispersed and grouped errors are, and most importantly, we directly see the worst predictions and how many of them are generated by our model.
- What is plotted on a histogram is described by statistics. We may check the max and the min error, but the true power comes when we analyze quartiles and standard deviation.

- The standard deviation is a good measurement of our model's variance; the less, the better.
- Sometimes, we must look into quartiles. A very high mean but relatively low median (or even the 3rd quartile) indicates that we have only a few wrong predictions, most of which are acceptable.

Where to go from here?

- C.1.5 Poisson Kriging Area to Point

Changelog

Date	Change description	Author
2023-08-25	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-01-14	New area-to-area PK algorithm	@Simon-Molinsky
2022-10-21	The tutorial was updated for the 0.3.4 version of the package	@Simon-Molinsky
2022-08-27	The tutorial was updated for the 0.3.0 version of the package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-12-11	Behavior of `prepare_kriging_data()` function has benn changed	@Simon-Molinsky
2021-05-28	Updated paths for input/output data	@Simon-Molinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@Simon-Molinsky

[10]:

Poisson Kriging - Area to Point Kriging

Table of Contents:

1. Load areal and point data,
2. Load semivariogram (regularized),
3. Build point-based map of better spatial resolution.

Introduction

To start this tutorial, it is required to understand concepts in the **Ordinary and Simple Kriging** and **Semivariogram Regularization** tutorials. The good idea is to end **Poisson Kriging - Centroid based** and **Poisson Kriging - Area to Area** tutorials before this one.

The Poisson Kriging technique is used to model spatial count data. We are analyzing a particular case where data is counted over areas. Those areas may have irregular shapes and sizes because they represent administrative regions.

We will transform the areal rates of Breast Cancer in the Northeastern US counties into representative population blocks (point support).

> The breast cancer rates data and the point support population counts are in the geopackage in a directory: `samples/regularization/cancer_data.gpkg`.

The tutorial covers the following steps:

1. Read and explore data,
2. Load semivariogram model,
3. Perform Area to Point smoothing of areal data.
4. Visualize points.

1) Read and explore data

[1]:
```
import numpy as np

from pyinterpolate import TheoreticalVariogram
from pyinterpolate import Blocks, PointSupport
from pyinterpolate import smooth_blocks
```

[2]:
```
DATASET = 'samples/regularization/cancer_data.gpkg'
OUTPUT = 'samples/regularization/regularized_variogram.json'
POLYGON_LAYER = 'areas'
POPULATION_LAYER = 'points'
POP10 = 'POP10'
GEOMETRY_COL = 'geometry'
POLYGON_ID = 'FIPS'
POLYGON_VALUE = 'rate'

blocks = Blocks()
blocks.from_file(DATASET, value_col=POLYGON_VALUE, index_col=POLYGON_ID, layer_
↪name=POLYGON_LAYER)

point_support = PointSupport()
point_support.from_files(point_support_data_file=DATASET,
                         blocks_file=DATASET,
                         point_support_geometry_col=GEOMETRY_COL,
                         point_support_val_col=POP10,
                         blocks_geometry_col=GEOMETRY_COL,
                         blocks_index_col=POLYGON_ID,
                         use_point_support_crs=True,
                         point_support_layer_name=POPULATION_LAYER,
```

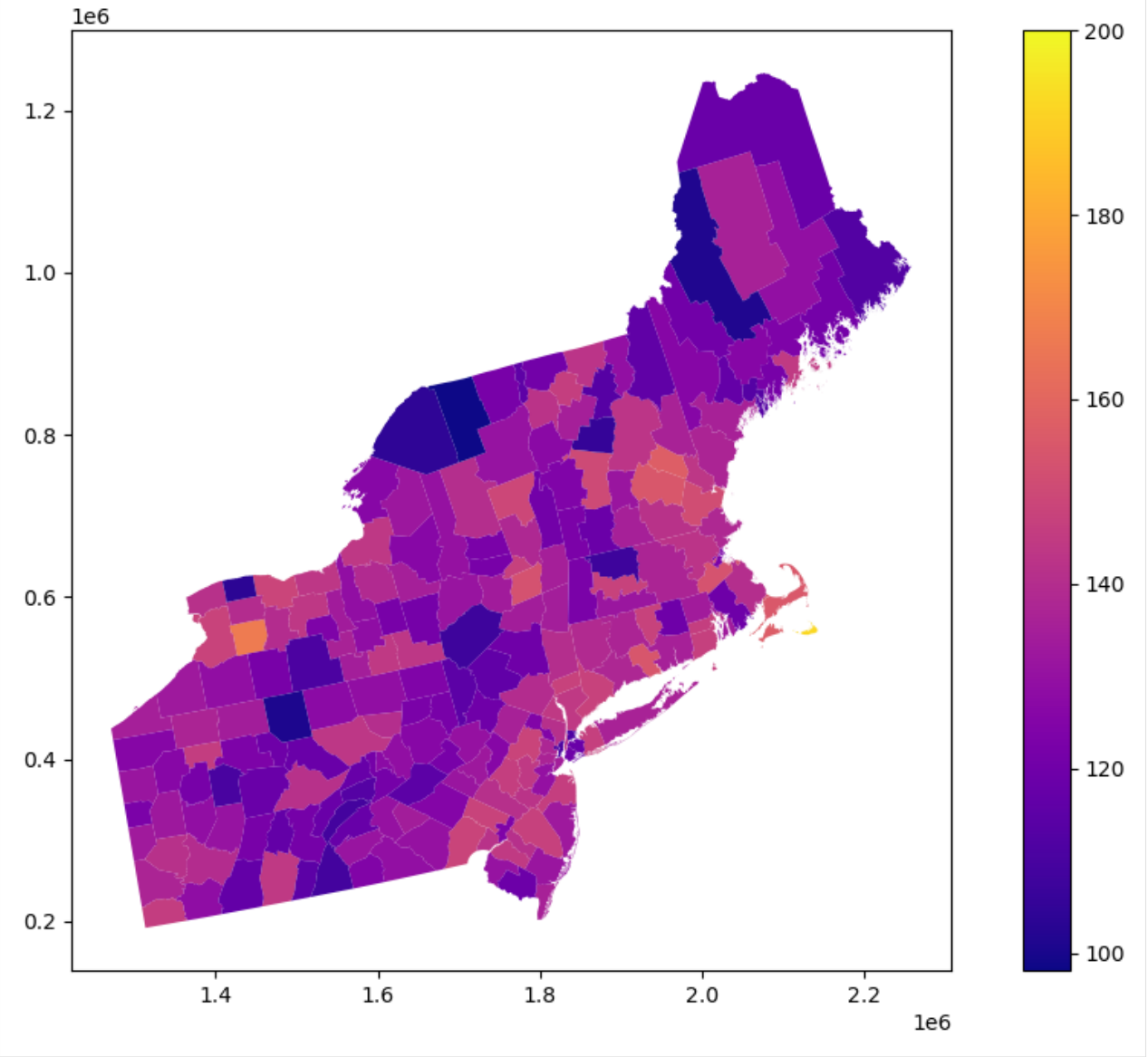
(continues on next page)

(continued from previous page)

```
                                blocks_layer_name=POLYGON_LAYER)

# Let's take a look into a map of areal counts

blocks.data.plot(column=blocks.value_column_name, cmap='plasma', vmax=200, legend=True,␣
↪figsize=(12, 8))
```

```
ERROR 1: PROJ: proj_create_from_database: Open of /home/szymon/miniconda3/envs/
↪pyinterpolate38/share/proj failed
```

[2]:
```
<Axes: >
```



Clarification: It is a good idea to look into the spatial patterns in a dataset and to visually check if our data do not have any `NaN` values. We use the geopandas `GeoDataFrame.plot()` function with a color map that diverges regions based on the cancer incidence rates. The output choropleth map is not ideal, and (probably) it has a few unreliable results, for example:

- In counties with a small population, where the ratio number of cases to population size is high even if the number of cases is low,
- Counties that are very big and sparsely populated may draw more of our attention than densely populated counties,
- Transitions of colors (rates) between counties may be too abrupt, even if we know that neighboring counties should have closer results.

We can overcome those problems using Area-to-Point Kriging. A deconvoluted choropleth map is smoother than the colored polygon representation, and the mapping does not include places where the population is equal to zero.

2) Load semivariogram model

In this step, we load regularized semivariogram from the tutorial **Semivariogram Regularization**. You can always perform semivariogram regularization along with the Poisson Kriging, but it is a very long process, and it is more convenient to separate those two steps.

[3]:
```
semivariogram = TheoreticalVariogram()  # Create TheoreticalSemivariogram object
semivariogram.from_json('output/regularized_model.json')  # Load regularized␣
↪semivariogram
```

3) Perform Area to Point smoothing of areal data.

The process of map smoothing is straightforward. We use the `smooth_blocks()` function and pass a block and its point support into it. We pass into the function two interesting parameters:

- The `number_of_neighbors` (the most crucial parameter - how many neighbors are affecting your analysis area).
- The CRS (Coordinate Reference System). We may leave it as `None`, but then returned `GeoDataFrame` won't have **CRS**. Be careful! The function doesn't check if passed **CRS** is valid for given blocks and point support!

The method returns `GeoDataFrame` with points and predicted values. It iteratively re-calculates each area risk and produces predictions per point. In Area-to-Area Kriging, those predictions are aggregated. We leave them and use them as a smooth map of areal risk.

[4]:
```
smoothed = smooth_blocks(semivariogram_model=semivariogram,
                         blocks=blocks,
                         point_support=point_support,
                         number_of_neighbors=16,
                         max_range=10000,
                         crs=blocks.data.crs,
                         raise_when_negative_error=False,
                         raise_when_negative_prediction=True)
```

```
100%|| 217/217 [00:43<00:00,  5.03it/s]
```

[5]:
```
smoothed.head()
```

[5]:

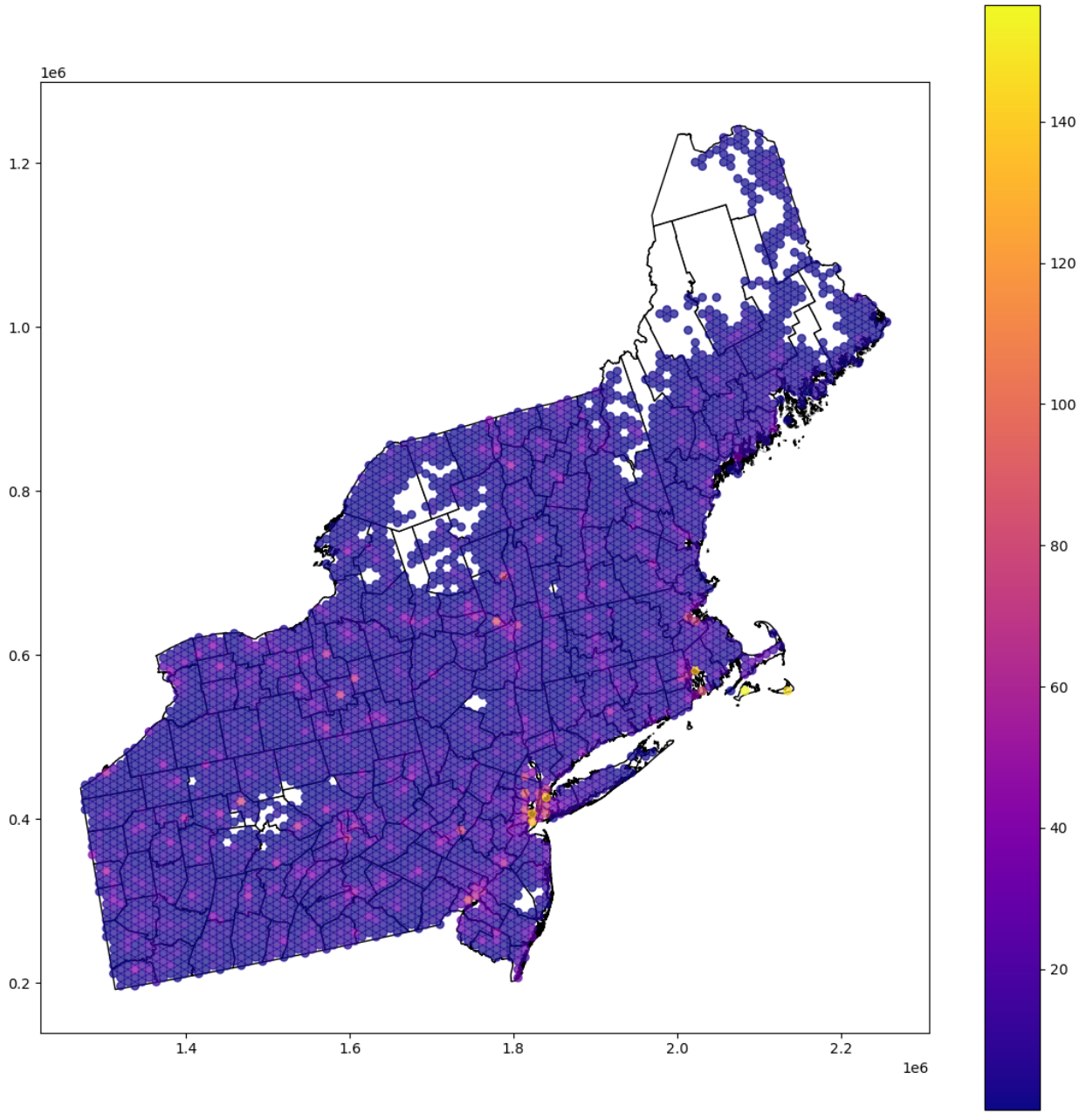
	area id	geometry	pred	err
0	25019.0	POINT (2117322.312 556124.507)	0.085944	1.048220
1	25019.0	POINT (2134642.820 556124.507)	146.811399	4.084356
2	36121.0	POINT (1424501.989 556124.507)	5.246010	8.495757
3	36121.0	POINT (1424501.989 546124.507)	6.088192	8.518457
4	36121.0	POINT (1433162.243 561124.507)	18.819513	8.828136

4) Visualize data

The last step is data visualization. We use a choropleth map from the GeoPandas package, but you can store a smoothed map to a spatial file of points and process it in a different place or with specific software (in our idea, the best for it is **QGIS**).

[6]:
```
base = blocks.data.plot(figsize=(14, 14), color='white', edgecolor='black')
smooth_plot_data = smoothed.copy()
smooth_plot_data[smooth_plot_data['pred'] < 0.1] = np.nan
smooth_plot_data.plot(ax=base, column='pred', cmap='plasma', legend=True, markersize=30,␣
↪alpha=0.7)
```

[6]:
```
<Axes: >
```



Changelog

Date	Change description	Author
2023-08-25	The tutorial was refreshed and set along with the 0.5.0 version of the package	@Simon-Molinsky
2023-01-16	New area-to-point PK algorithm	@Simon-Molinsky
2022-10-21	The tutorial was updated for the 0.3.4 version of the package	@Simon-Molinsky
2022-08-27	The tutorial was updated for the 0.3.0 version of the package	@Simon-Molinsky
2021-12-14	Sill selection was upgraded: now optimal sill is derived from the grid search within `TheoreticalSemivariogram` class	@Simon-Molinsky
2021-12-13	Changed behavior of `select_values_in_range()` function	@Simon-Molinsky
2021-12-11	Behavior of `prepare_kriging_data()` function has benn changed	@Simon-Molinsky
2021-05-28	Updated paths for input/output data	@Simon-Molinsky
2021-05-11	Refactored TheoreticalSemivariogram class	@Simon-Molinsky
2021-03-31	Update related to the change of semivariogram weighting. Updated cancer rates data.	@Simon-Molinsky

[6]:

2.4 How to cite

Moliński, S., (2022). Pyinterpolate: Spatial interpolation in Python for point measurements and aggregated datasets. Journal of Open Source Software, 7(70), 2869, https://doi.org/10.21105/joss.02869

2.5 API

2.5.1 Core data structures

class Blocks

Class stores and prepares aggregated data.

Examples

```
>>> geocls = Blocks()
>>> geocls.from_file('testfile.shp', value_col='val', geometry_col='geometry',␣
↪index_col='idx')
>>> parsed_columns = geocls.data.columns
>>> print(list(parsed_columns))
(idx, val, geometry, centroid.x, centroid.y)
```

Attributes

data
[gpd.GeoDataFrame] Dataset with block values.

value_column_name
[Any] Name of the column with block rates.

geometry_column_name
[Any] Name of the column with a block geometry.

index_column_name
[Any] Name of the column with the index.

Methods

from_file()	Reads and parses data from spatial file supported by GeoPandas.
from_geodataframe()	Reads and parses data from GeoPandas `GeoDataFrame`.

`from_file`(*fpath*, *value_col*, *geometry_col='geometry'*, *index_col=None*, *layer_name=None*, ***kwargs*)

Loads areal dataset from a file supported by GeoPandas.

Parameters

fpath
[str] Path to the spatial file with appropriate extension such as `.shp`,

``.gpkg``, ``.feather``, or ``.parquet``.
value_col
[Any] The name of a column with values.

geometry_col
[default='geometry'] The name of a column with blocks.

index_col
[default = None] Index column name. It could be any unique value from a dataset. If not given then index is taken from the index array of `GeoDataFrame`, and it is named `'index'`.

layer_name
[Any, default = None] The name of a layer with data if provided input is a *gpkg* file.

****kwargs**
[Any] Additional kwargs parameters passed to the `geopandas.read_file()`, `geopandas.read_feather()` or `geopandas.read_parquet()` functions.

Raises

IndexColNotUniqueError
Raised when given index column has not unique values.

`from_geodataframe`(*gdf*, *value_col*, *geometry_col='geometry'*, *index_col=None*)

Loads areal dataset from a GeoDataFrame supported by GeoPandas.

Parameters

gdf
[gpd.GeoDataFrame]

value_col
[Any] The name of a column with values.

geometry_col
[Any, default = 'geometry'] The name of a column with blocks.

index_col
[Any, default = None] If set then a specific column is treated as an index.

Raises

IndexColNotUniqueError
Given index column values are not unique.

`class PointSupport`(*log_not_used_points=False*)

Class prepares the point support data in relation to block dataset.

Parameters

log_not_used_points
[bool, default=False] Should dropped points be logged?

Notes

The PointSupport class structure is designed to store the information about the points within polygons. During the regularization process, the inblock variograms are estimated from the polygon's point support, and semivariances are calculated between point supports of neighbouring blocks.

The class takes population grid (support) and blocks data (polygons). Then, spatial join is performed and points are assigned to areas within they are placed. The core attribute is `point_support` - GeoDataFrame with columns:

- `x_col` - a floating representation of longitude,
- `y_col` - a floating representation of latitude,
- `value_column` - the attribute which describes the name of a column with the point-support's value,
- `geometry_column` - the attribute which describes the name of a geometry column with `Point()` representation of the point support coordinates,
- `block_index_column` - the name of a column which directs to the block index values.

Examples

```
>>> import geopandas as gpd
>>> from pyinterpolate import PointSupport
>>>
>>>
>>> POPULATION_DATA = "path to the point support file"
>>> POLYGON_DATA = "path to the polygon data"
>>> GEOMETRY_COL = "geometry"
>>> POP10 = "POP10"
>>> POLYGON_ID = "FIPS"
>>>
>>> gdf_points = gpd.read_file(POPULATION_DATA)
>>> gdf_polygons = gpd.read_file(POLYGON_DATA)
>>> point_support = PointSupport()
>>> out = point_support.from_geodataframes(gdf_points,
...                                        gdf_polygons,
...                                        point_support_geometry_col=GEOMETRY_COL,
...                                        point_support_val_col=POP10,
...                                        blocks_geometry_col=GEOMETRY_COL,
...                                        blocks_index_col=POLYGON_ID)
```

Attributes

point_support
: [gpd.GeoDataFrame] Dataset with point support values and indexes of blocks (where points fall into).

value_column
: [str] The value column name

geometry_column
: [str] The geometry column name.

block_index_column
: [str] The area index.

x_col
: [str, default = "x_col"] Longitude column name.

y_col
: [str, default = "y_col"] Latitude column name.

log_dropped
: [bool] See log_not_used_points parameter.

Methods

from_files()	Loads point support and polygon data from files.
from_geodataframes()	Loads point support and polygon data from dataframe.

from_files(*point_support_data_file*, *blocks_file*, *point_support_geometry_col*, *point_support_val_col*, *blocks_geometry_col*, *blocks_index_col*, *use_point_support_crs=True*, *point_support_layer_name=None*, *blocks_layer_name=None*, ***kwargs*)

Methods prepares the point support data from files.

Parameters

point_support_data_file
[str] Path to the file with point support data. Reads all files processed by the GeoPandas with appropriate extension such as `.shp`, `.gpkg`, `.feather`, or `.parquet`.

blocks_file
[str] Path to the file with polygon data. Reads all files processed by GeoPandas with appropriate extension such as `.shp`,

``.gpkg``, ``.feather``, or ``.parquet``.
point_support_geometry_col
[Any] The name of the point support geometry column.

point_support_val_col
[Any] The name of the point support column with values.

blocks_geometry_col
[Any] The name of the polygon geometry column.

blocks_index_col
[Any] The name of polygon's index column (must be unique!).

use_point_support_crs
[bool, default = True] If set to `False` then the point support crs is transformed to the same crs as polygon dataset.

point_support_layer_name
[Any, default = None] If provided file is *.gpkg* then this parameter must be provided.

blocks_layer_name
[Any, default = None] If provided file is *.gpkg* then this parameter must be provided.

****kwargs**
[Any] Additional kwargs parameters passed to the `geopandas.read_file()`, `geopandas.read_feather()` or `geopandas.read_parquet()` functions.

from_geodataframes(*point_support_dataframe*, *blocks_dataframe*, *point_support_geometry_col*, *point_support_val_col*, *blocks_geometry_col*, *blocks_index_col*, *use_point_support_crs=True*)

Methods prepares the point support data from files.

Parameters

point_support_dataframe
[GeoDataFrame or GeoSeries]

blocks_dataframe
[GeoDataFrame] Block data with block indexes and geometries.

point_support_geometry_col
[Any] The name of the point support geometry column.

point_support_val_col
[Any] The name of the point support column with values.

point_support_val_col
[Any] The name of the point support column with values.

blocks_geometry_col
[Any] The name of the polygon geometry column.

blocks_index_col
[Any] The name of polygon's index column (must be unique!).

use_point_support_crs
[bool, default = True] If set to `False` then the point support crs is transformed to the same crs as polygon dataset.

2.5.2 Input / Output

`read_block`(*path*, *val_col_name*, *geometry_col_name='geometry'*, *id_col_name=None*, *centroid_col_name=None*, *epsg=None*, *crs=None*, ***kwargs*)

Function reads block data from files supported by geopandas.

Value column name must be provided. If geometry column has different name than *'geometry'* then it must be provided too. ID column name is optional, if not given then `GeoDataFrame` *index* is treated as an id column. Optional parameters are *epsg* and *crs*. If any is set then data is reprojected into a specific *crs/epsg*.` Function returns `GeoDataFrame` with columns: `[id, value, geometry, centroid]`.

Parameters

path
[str] Path to the file with appropriate extension such as `.shp`, `.gpkg`, `.feather`, or `.parquet`.

val_col_name
[str] Name of the value column (header title).

geometry_col_name
[str, default='geometry'] Name of the column with polygons.

id_col_name: str or None, default=None, optional
Name of the colum with unique indexes.

centroid_col_name: str or None, default=None
Name of the column with block centroid. Centroids are calculated from `MultiPolygon` or `Polygon` later on but their accuracy may be limited. For most applications it does not matter.

epsg
[str or None, default=None] If provided then `GeoDataFrame` projection is set to it. You should choose if you provide *EPSG* or *CRS*.

crs
[str or None, default=None] If provided then `GeoDataFrame` projection is set to it. You should choose if you provide *CRS* or *EPSG*.

****kwargs**
[Any] Additional kwargs parameters passed to the `geopandas.read_file()`, `geopandas.read_feather()` or `geopandas.read_parquet()` functions.

Returns

gpd
[GeoDataFrame] Returned output has columns: `['id', 'geometry', 'value', 'centroid']`.

Raises

TypeError
EPSG and *CRS* are provided both (should be only one).

TypeError
Provided column name does not exist in a dataset.

Examples

```
>>> bblock = 'path_to_the_shapefile.shp'
>>> bdf = read_block(bblock, val_col_name='rate', id_col_name='id')
>>> print(bdf.columns)
Index(['id', 'geometry', 'rate', 'centroid'], dtype='object')
```

---

read_csv(*path*, *val_col_name*, *lat_col_name*, *lon_col_name*, *delim=','*)

Function reads data from a csv file.

Provided data should include: **latitude**, **longitude**, **value**.

Parameters

path
[str] Path to the file.

val_col_name
[str] Name of the value column (header title).

lat_col_name
[str] Name of the latitude column (header title).

lon_col_name
[str] Name of the longitude column (header title).

delim
[str, default=',’] Delimiter that separates columns.

Returns

data_arr
[numpy array]

Examples

```
>>> path_to_the_data = 'path_to_the_data.csv'
>>> data = read_csv(path_to_the_data, val_col_name='value', lat_col_name='y', lon_
↪col_name='x')
>>> print(data[:2, :])
[
    [15.11524 52.76515 91.275597]
    [15.11524 52.74279 96.548294]
]
```

---

read_txt(*path*, *delim=','*, *skip_header=True*)

Function reads data from a text file.

Provided data format should include: **longitude (x)**, **latitude (y)**, **value**. Function converts data into numpy array.

Parameters

path
[str] Path to the file.

delim
[str, default=',] Delimiter that separates columns.

skip_header
[bool, default=True] Skips the first row of a file if set to `True`.

Returns

data_arr
[numpy array]

Examples

```
>>> path_to_the_data = 'path_to_the_data.txt'
>>> data = read_txt(path_to_the_data, skip_header=False)
>>> print(data[:2, :])
[
    [15.11524 52.76515 91.275597]
    [15.11524 52.74279 96.548294]
]
```

2.5.3 Distance calculations

point_distance(*points*, *other*, *metrics='euclidean'*)

Calculates the euclidean distance from one group of points to another group of points.

Parameters

points
[array] Spatial coordinates.

other
[array] Other array with spatial coordinates.

metrics
[str, default = 'euclidean'] Metrics used to calculate distance. See `scipy.spatial.distance.cdist` for more details.

Returns

distances
[array] Distances matrix. Row index = `points` point index, and column index = `other` point index.

Notes

The function creates array of size MxN, where M = number of `points` and N = number of `other`. Very big array with coordinates may cause a memory error.

Examples

```
>>> points = [(0, 0), (0, 1), (0, 2)]
>>> other = [(2, 2), (3, 3)]
>>> distances = point_distance(points=points, other=other)
>>> print(distances)
[[2.82842712 4.24264069]
 [2.23606798 3.60555128]
 [2.         3.16227766]]
```

`calc_block_to_block_distance`(*blocks*)

Function calculates distances between blocks.

Parameters

blocks
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,
- `DataFrame` and `GeoDataFrame`: `columns={x, y, ds, index}`,
- `PointSupport`.

Returns

block_distances
[Dict] Ordered block ids (the order from the list of distances): {block id : [distances to other]}.

Raises

TypeError
Wrong input's data type.

deprecated

`calc_point_to_point_distance`(*points_a*, *points_b=None*)

Function calculates distances between two group of points of a single group to itself.

Parameters

points_a
[numpy array] The point coordinates.

points_b
[numpy array, default=None] Other point coordinates. If provided then algorithm calculates distances between `points_a` against `points_b`.

Returns

distances
[numpy array] The distances from each point from the `points_a` to other point (from the same `points_a` or from the other set of points `points_b`).

Deprecated since version 0.5.1: This will be removed in 1.0. Use *point_distance()* instead

2.5.4 Gridding

`create_grid`(*ds*, *min_number_of_cells*, *grid_type='box'*)

Function creates grid based on a set of points.

Parameters

ds
[Union[np.ndarray, List, gpd.GeoSeries]] Data to be transformed, point coordinates [x, y] <-> [lon, lat] or GeoSeries with Point geometry.

min_number_of_cells
[int] Expected number of cells in the smaller dimension.

grid_type
[str, default = "square"] Available types: `box`, `hex`.

Returns

grid
[gpd.GeoSeries] Empty grid that can be used to aggregate points. It is GeoSeries of Polygons (squares or hexes).

`points_to_grid`(*points*, *grid*, *fillna=None*)

Function aggregates points over a specified grid.

Parameters

points
[geopandas GeoDataFrame]

grid
[geopandas GeoSeries]

fillna
[Any, optional] The value to fill NaN's, if None given then this step is skipped.

Returns

aggregated
[geopandas GeoDataFrame]

2.5.5 Variogram

Experimental

`build_experimental_variogram`(*input_array*, *step_size*, *max_range*, *weights=None*, *direction=None*, *tolerance=1*, *method='t'*)

Function prepares:

- experimental semivariogram,
- experimental covariogram,
- variance.

Parameters

input_array
[numpy array] Spatial coordinates and their values: `[pt x, pt y, value]` or `[shapely.Point(), value]`.

step_size
[float] The distance between lags within each points are included in the calculations.

max_range
[float] The maximum range of analysis.

weights
[numpy array or None, optional, default=None] Weights assigned to points, index of weight must be the same as index of point.

direction
[float (in range [0, 360]), optional] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), default = 1] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance:

- the major axis size == `step_size`,
- the minor axis size is `tolerance * step_size`,
- the baseline point is at a center of the ellipse,
- the `tolerance == 1` creates an omnidirectional semivariogram.

method
[str, default = triangular] The method used for neighbors selection. Available methods:

- "triangle" or "t", default method where a point neighbors are selected from a triangular area,
- "ellipse" or "e", the most accurate method but also the slowest one.

Returns

semivariogram_stats
[EmpiricalSemivariogram] The class with empirical semivariogram, empirical covariogram and a variance.

See also:

`calculate_covariance`
function to calculate experimental covariance and variance of a given set of points.

`calculate_semivariance`
function to calculate experimental semivariance from a given set of points.

`EmpiricalSemivariogram`
class that calculates and stores experimental semivariance, covariance and variance.

Notes

Function is an alias for `EmpiricalSemivariogram` class and it forces calculations of all spatial statistics from a given dataset.

Examples

```
>>> import numpy as np
>>> REFERENCE_INPUT = np.array([
...     [0, 0, 8],
...     [1, 0, 6],
...     [2, 0, 4],
...     [3, 0, 3],
...     [4, 0, 6],
...     [5, 0, 5],
...     [6, 0, 7],
...     [7, 0, 2],
...     [8, 0, 8],
...     [9, 0, 9],
...     [10, 0, 5],
...     [11, 0, 6],
...     [12, 0, 3]
...     ])
>>> STEP_SIZE = 1
>>> MAX_RANGE = 4
>>> empirical_smv = build_experimental_variogram(REFERENCE_INPUT, step_size=STEP_
↪SIZE, max_range=MAX_RANGE)
>>> print(empirical_smv)
+-----+--------------------+--------------------+--------------------+
| lag |    semivariance    |      covariance    |    var_cov_diff    |
+-----+--------------------+--------------------+--------------------+
| 1.0 |       4.625        |       -0.543       |       4.792        |
```

(continues on next page)

(continued from previous page)

```
| 2.0 |       5.227        |       -0.795        |       5.043        |
| 3.0 |        6.0         |       -1.26         |       5.509        |
+-----+--------------------+---------------------+--------------------+
```

---

build_variogram_point_cloud(*input_array*, *step_size*, *max_range*, *direction=None*, *tolerance=1.0*)

Function calculates lagged variogram point cloud. Variogram is calculated as a squared difference of each point
against other point within range specified by step_size parameter.

Parameters

input_array
[numpy array] Spatial coordinates and their values: `[pt x, pt y, value]` or `[shapely.Point(), value]`.

step_size
[float] The distance between lags within each points are included in the calculations.

max_range
[float] The maximum range of analysis.

direction
[float (in range [0, 360]), default=None] Direction of semivariogram, values from 0 to 360 degrees: - 0 or 180: is E-W, - 90 or 270 is N-S, - 45 or 225 is NE-SW, - 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), optional, default=1] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is > `0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `step_size`.
- The minor axis size is `tolerance * step_size`
- The baseline point is at a center of the ellipse.
- The `tolerance == 1` creates an omnidirectional semivariogram.

Returns

variogram_cloud
[Dict] `{Lag: array of semivariances within a given lag}`

---

class ExperimentalVariogram(*input_array*, *step_size*, *max_range*, *weights=None*, *direction=None*, *tolerance=1.0*, *method='t'*, *is_semivariance=True*, *is_covariance=True*)

Class calculates Experimental Semivariogram and Experimental Covariogram of a given dataset.

Parameters

input_array
[numpy array, list, tuple]

- As a `list` and `numpy array`: coordinates and their values: `(pt x, pt y, value)`,

- as a `dict`: `polyset = {'points': numpy array with coordinates and their values}`,
- as a `Blocks`: `Blocks.polyset['points']`.

step_size
[float] The distance between lags within each points are included in the calculations.

max_range
[float] The maximum range of analysis.

weights
[numpy array, default=None] Weights assigned to points, index of weight must be the same as index of point.

direction
[float (in range [0, 360]), optional] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), default = 1] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `step_size`.
- The minor axis size is `tolerance * step_size`
- The baseline point is at a center of the ellipse.
- The `tolerance == 1` creates an omnidirectional semivariogram.

method
[str, default = triangular] The method used for neighbors selection. Available methods:

- "triangle" or "t", default method where a point neighbors are selected from a triangular area,
- "ellipse" or "e", the most accurate method but also the slowest one.

is_semivariance
[bool, optional, default=True] Should semivariance be calculated?

is_covariance
[bool, optional, default=True] Should covariance be calculated?

See also:

`calculate_covariance`
function to calculate experimental covariance and variance of a given set of points.

`calculate_semivariance`
function to calculate experimental semivariance from a given set of points.

Examples

```
>>> import numpy as np
>>> REFERENCE_INPUT = np.array([
...     [0, 0, 8],
...     [1, 0, 6],
...     [2, 0, 4],
...     [3, 0, 3],
...     [4, 0, 6],
...     [5, 0, 5],
...     [6, 0, 7],
...     [7, 0, 2],
...     [8, 0, 8],
...     [9, 0, 9],
...     [10, 0, 5],
...     [11, 0, 6],
...     [12, 0, 3]
...     ])
>>> STEP_SIZE = 1
>>> MAX_RANGE = 4
>>> empirical_smv = ExperimentalVariogram(REFERENCE_INPUT, step_size=STEP_SIZE, max_
↪range=MAX_RANGE)
>>> print(empirical_smv)
+-----+--------------------+---------------------+--------------------+
| lag |    semivariance    |      covariance     |    var_cov_diff    |
+-----+--------------------+---------------------+--------------------+
| 1.0 |       4.625        | -0.5434027777777798 | 4.791923487836951  |
| 2.0 | 5.2272727272727275 | -0.7954545454545454 | 5.0439752555137165 |
| 3.0 |        6.0         | -1.2599999999999958 | 5.508520710059168  |
+-----+--------------------+---------------------+--------------------+
```

Attributes

input_array
: [numpy array] The array with coordinates and observed values.

experimental_semivariance_array
: [numpy array or None, optional, default=None] The array of semivariance per lag in the form: `(lag, semivariance, number of points within lag)`.

experimental_covariance_array
: [numpy array or None, optional, default=None] The array of covariance per lag in the form: `(lag, covariance, number of points within lag)`.

experimental_semivariances
: [numpy array or None, optional, default=None] The array of semivariances.

experimental_covariances
: [numpy array or None, optional, default=None] The array of covariances.

variance_covariances_diff
: [numpy array or None, optional, default=None] The array of differences $c(0) - c(h)$.

lags
: [numpy array or None, default=None] The array of lags (upper bound for each lag).

points_per_lag
[numpy array or None, default=None] A number of points in each lag-bin.

variance
[float or None, optional, default=None] The variance of a dataset, if data is second-order stationary then we are able to retrieve a semivariance s a difference between the variance and the experimental covariance:

$$g(h) = c(0) - c(h)$$

where:

- $g(h)$: semivariance at a given lag h,
- $c(0)$: variance of a dataset,
- $c(h)$: covariance of a dataset.

Important! Have in mind that it works only if process is second-order stationary (variance is the same for each distance bin) and if the semivariogram has the upper bound.

step
[float] Derived from the `step_size` parameter.

mx_rng
[float] Derived from the `max_range` parameter.

weights
[numpy array or None] Derived from the `weights` parameter.

direct: float
Derived from the `direction` parameter.

tol
[float] Derived from the `tolerance` parameter.

method
[str] See the `method` parameter.

Methods

plot()	Shows experimental variances.

`plot`(*plot_semivariance=True*, *plot_covariance=True*, *plot_variance=True*)

Parameters

plot_semivariance
[bool, default=True] Show semivariance on a plot. If class attribute `is_semivariance` is set to `False` then semivariance is not plotted and warning is printed.

plot_covariance
[bool, default=True] Show covariance on a plot. If class attribute `is_covariance` is set to `False` then covariance is not plotted and warning is printed.

plot_variance
[bool, default=True] Show variance level on a plot.

Warns

AttributeSetToFalseWarning
Warning invoked when plotting parameter for semivariance, covariance or variance is set to `True` but class attributes to calculate those indices are set to `False`.

`class VariogramCloud`(*input_array*, *step_size*, *max_range*, *direction=0*, *tolerance=1*, *calculate_on_creation=True*)

Class calculates Variogram Point Cloud and presents it in a scatterplot, boxplot and violinplot.

Parameters

input_array
[numpy array] Spatial coordinates and their values: `[pt x, pt y, value]` or `[shapely.Point(), value]`.

step_size
[float] The distance between lags within each points are included in the calculations.

max_range
[float] The maximum range of analysis.

direction
[float (in range [0, 360]), optional, default=None] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), optional, default=1] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is > `0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `step_size`.
- The minor axis size is `tolerance * step_size`
- The baseline point is at a center of the ellipse.
- The `tolerance == 1` creates an omnidirectional semivariogram.

See also:

`get_variogram_point_cloud`
function to calculate variogram point cloud, class VariogramCloud is a wrapper around it.

`ExperimentalVariogram`
class that calculates experimental semivariogram, experimental covariogram and data variance.

Examples

```
>>> import numpy as np
>>> REFERENCE_INPUT = np.array([
...     [0, 0, 8],
...     [1, 0, 6],
...     [2, 0, 4],
...     [3, 0, 3],
...     [4, 0, 6],
...     [5, 0, 5],
...     [6, 0, 7],
...     [7, 0, 2],
...     [8, 0, 8],
...     [9, 0, 9],
...     [10, 0, 5],
...     [11, 0, 6],
...     [12, 0, 3]
...     ])
>>> STEP_SIZE = 1
>>> MAX_RANGE = 4
>>> point_cloud = VariogramCloud(REFERENCE_INPUT, step_size=STEP_SIZE, max_
↪range=MAX_RANGE)
# >>> print(point_cloud)
# +-----+--------------------+--------------------+--------------------+
# | lag |    count    |      avg semivariance    |    std  | min | 25% | median | 75
↪% | max | skewness | kurtosis |
# +-----+--------------------+--------------------+--------------------+
# | 1.0 |       4.625        | -0.5434027777777798 | 4.791923487836951  |
# | 2.0 | 5.22727272727272 | -0.7954545454545454 | 5.0439752555137165 |
# | 3.0 |        6.0         | -1.2599999999999958 | 5.508520710059168  |
# +-----+--------------------+--------------------+--------------------+
```

Attributes

input_array
: [numpy array] The array with coordinates and observed values.

experimental_point_cloud
: [dict or None, default=None] Dict `{lag: [variances]}`.

lags
: [numpy array or None, default=None] The array of lags (upper bound for each lag).

points_per_lag
: [int or None, default=None] A number of points in each lag-bin.

step
: [float] Derived from the step_size parameter.

mx_rng
: [float] Derived from the max_range parameter.

direct: float
: Derived from the direction parameter.

tol
: [float] Derived from the tolerance parameter.

calculate_on_creation
[bool, default=True] Perform calculations of semivariogram point cloud when object is initialized.

Methods

calculate_experimental_variogram()	Method calculates experimental variogram from a point cloud.
describe()	calculates statistics for point clouds. It is invoked by default by class __str__() method.
plot(kind='scatter')	plots scatterplot, boxplot or violinplot of a point cloud.
remove_outliers()	Removes outliers from a semivariance scatterplots.

`calculate_experimental_variogram()`

Method transforms the experimental point cloud into the experimental variogram.

Raises

RunetimeError
The attribute experimental_point_cloud is not calculated.

`describe()`

Method calculates basic statistcs of a data: count (point pairs number), average semivariance, standard deviation, minimum, 1st quartile, median, 3rd quartile, maximum, skewness, kurtosis.

Returns

statistics
[Dict]

```
>>> statistics = {
...      lag_number:
...      {
...          'count': point pairs count,
...          'avg semivariance': mean semivariance,
...          'std': standard deviation,
...          'min': minimal variance,
...          '25%': first quartile of variances,
...          'median': second quartile of variances,
...          '75%': third quartile of variances,
...          'max': max variance,
...          'skewness': skewness,
...          'kurtosis': kurtosis
...      }
...  }
```

`plot(kind='scatter')`

Method plots variogram point cloud.

Parameters

kind
[string, default='scatter'] available plot types: 'scatter', 'box', 'violin'

Returns

: bool
`True` if Figure was plotted.

`remove_outliers`(*method='zscore'*, *z_lower_limit=-3*, *z_upper_limit=3*, *iqr_lower_limit=1.5*, *iqr_upper_limit=1.5*, *inplace=False*)

Parameters

method
[str, default='zscore'] Method used to detect outliers. Can be 'zscore' or 'iqr'.

z_lower_limit
[float] Number of standard deviations from the mean to the left side of a distribution. Must be lower than 0.

z_upper_limit
[float] Number of standard deviations from the mean to the right side of a distribution. Must be greater than 0.

iqr_lower_limit
[float] Number of standard deviations from the 1st quartile into the lowest values. Must be greater or equal to zero.

iqr_upper_limit
[float] Number of standard deviations from the 3rd quartile into the largest values. Must be greater or equal to zero.

inplace
[bool, default=False] If set to True then method updates experimental_point_cloud attribute of the existing object and returns nothing. Else new VariogramCloud object is returned.

Returns

cleaned
[VariogramCloud] VariogramCloud object with removed outliers from the experimental_point_cloud attribute.

Raises

RunetimeError
The attribute experimental_point_cloud is not calculated.

Theoretical

`build_theoretical_variogram`(*experimental_variogram*, *model_name*, *sill*, *rang*, *nugget=0.0*, *direction=None*)

Function is a wrapper into `TheoreticalVariogram` class and its `fit()` method.

Parameters

experimental_variogram
[ExperimentalVariogram]

model_name
[str] Available types:

- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',

- 'linear',
- 'power',
- 'spherical'.

sill
[float] The value at which dissimilarity is close to its maximum if model is bounded. Otherwise, it is usually close to observations variance.

rang
[float] The semivariogram range is a distance at which spatial correlation exists. It shouldn't be set at a distance larger than a half of a study extent.

nugget
[float, default=0.] The nugget parameter (bias at a zero distance).

direction
[float, in range [0, 360], default=None] The direction of a semivariogram. If `None` given then semivariogram is isotropic.

Returns

theo
[TheoreticalVariogram] Fitted theoretical semivariogram model.

`class TheoreticalVariogram(`*model_params=None*`)`

Theoretical model of a spatial dissimilarity.

Parameters

model_params
[Dict, default=None] Dictionary with `'nugget'`, `'sill'`, `'range'` and `'name'` of the model.

See also:

`ExperimentalVariogram`
class to calculate experimental variogram and more.

Examples

```
>>> import numpy
>>> REFERENCE_INPUT = numpy.array([
...     [0, 0, 1],
...     [1, 0, 2],
...     [2, 0, 3],
...     [3, 0, 4],
...     [4, 0, 5],
...     [5, 0, 6],
...     [6, 0, 7],
...     [7, 0, 5],
...     [8, 0, 3],
...     [9, 0, 1],
...     [10, 0, 4],
...     [11, 0, 6],
```

(continues on next page)

(continued from previous page)

```
...     [12, 0, 8]
...     ])
>>> STEP_SIZE = 1
>>> MAX_RANGE = 4
>>> empirical_smv = ExperimentalVariogram(REFERENCE_INPUT, step_size=STEP_SIZE, max_
↪range=MAX_RANGE)
>>> theoretical_smv = TheoreticalVariogram()
>>> _ = theoretical_smv.autofit(experimental_variogram=empirical_smv, model_name=
↪'gaussian')
>>> print(theoretical_smv.rmse)
1.5275214898546217
```

Attributes

experimental_variogram
[EmpiricalVariogram, default=None] Empirical Variogram class and its attributes.

experimental_array
[numpy array, default=None] Empirical variogram in a form of numpy array.

variogram_models
[Dict] A dictionary with keys representing theoretical variogram models and values that are pointing into a modeling functions. Available models:

- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',
- 'spherical'.

fitted_model
[numpy array or None] Trained theoretical model values. Array of `[lag, variances]`.

name
[str or None, default=None] The name of the chosen model. Available names are the same as keys in `variogram_models` attribute.

nugget
[float, default=0] The nugget parameter (bias at a zero distance).

sill
[float, default=0] A value at which dissimilarity is close to its maximum if model is bounded. Otherwise, it is usually close to observations variance.

rang
[float, default=0] The semivariogram range is a distance at which spatial correlation exists. It shouldn't be set at a distance larger than a half of a study extent.

direction
[float, in range [0, 360], default=None] The direction of a semivariogram. If `None` given then semivariogram is isotropic.

rmse
[float, default=0] Root mean squared error of the difference between the empirical observations and the modeled curve.

mae
[bool, default=True] Mean Absolute Error of a model.

bias
[float, default=0] Forecast Bias of the modeled variogram vs experimental points. Large positive value means that the estimated model underestimates values. A large negative value means that model overestimates predictions.

smape
[float, default=0] Symmetric Mean Absolute Percentage Error of the prediction - values from 0 to 100%.

spatial_dependency_ratio
[float, default = None] The ratio of nugget vs sill multiplied by 100. Levels from 0 to 25 indicate strong spatial dependency, from 25 to 75 moderate spatial dependency, from 75 to 95 weak spatial dependency, and above process is considered to be not spatially-depended.

spatial_dependency_strength
[str, default = "Unknown"]

Descriptive indicator of spatial dependency strength based on the `spatial_dependency_level`. It could be:

- `unknown` if ratio is `None`,
- `strong` if ratio is below 25,
- `moderate` if ratio is between 25 and 75,
- `weak` if ratio is between 75 and 95,
- `no spatial dependency` if ratio is greater than 95.

are_params
[bool] Check if model parameters were given during initialization.

Methods

fit()	Fits experimental variogram data into theoretical model.
autofit()	The same as fit but tests multiple nuggets, ranges, sills and models.
calculate_model_error()	Evaluates the model performance against experimental values.
to_dict()	Store model parameters in a dict.
from_dict()	Read model parameters from a dict.
to_json()	Save model parameteres in a JSON file.
from_json()	Read model parameters from a JSON file.
plot()	Shows theoretical model.

`autofit`(*experimental_variogram*, *model_name='safe'*, *model_types=None*, *nugget=None*, *min_nugget=0*, *max_nugget=0.5*, *number_of_nuggets=16*, *rang=None*, *min_range=0.1*, *max_range=0.5*, *number_of_ranges=16*, *sill=None*, *min_sill=0.0*, *max_sill=1*, *number_of_sills=16*, *direction=None*, *error_estimator='rmse'*, *deviation_weighting='equal'*, *auto_update_attributes=True*, *warn_about_set_params=True*, *verbose=False*, *return_params=True*)

Method tries to find the optimal range, sill and model (function) of the theoretical semivariogram.

Parameters

experimental_variogram
[ExperimentalVariogram] Prepared Empirical Variogram or array.

model_name
[str, default='safe'] The name of the model to check or special command for multiple models. Available models:

- 'all' - the same as list with all models,
- 'safe' - ['linear', 'power', 'spherical'],
- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',
- 'spherical'.

model_types
[List, default = None] The list with model names to check excluding 'all' and 'safe' names.

nugget
[float, optional] Nugget (bias) of a variogram. If given then it is fixed to this value.

min_nugget
[float, default = 0] The minimum nugget as the ratio of the parameter to the first lag variance.

max_nugget
[float, default = 0.5] The maximum nugget as the ratio of the parameter to the first lag variance.

number_of_nuggets
[int, default = 16] How many equally spaced nuggets tested between `min_nugget` and `max_nugget`.

rang
[float, optional] If given, then range is fixed to this value.

min_range
[float, default = 0.1] The minimal fraction of a variogram range, `0 < min_range <= max_range`.

max_range
[float, default = 0.5] The maximum fraction of a variogram range, `min_range <= max_range <= 1`. Parameter `max_range` greater than **0.5** raises warning.

number_of_ranges
[int, default = 16] How many equally spaced ranges are tested between `min_range` and `max_range`.

sill
[float, default = None] If given, then sill is fixed to this value.

min_sill
[float, default = 0] The minimal fraction of the variogram variance at lag 0 to find a sill, `0 <= min_sill <= max_sill`.

max_sill
[float, default = 1] The maximum fraction of the variogram variance at lag 0 to find a sill. It *should be* lower or equal to 1. It is possible to set it above 1, but then warning is printed.

number_of_sills
[int, default = 16] How many equally spaced sill values are tested between `min_sill` and `max_sill`.

direction
[float, in range [0, 360], default=None] The direction of a semivariogram. If `None` given then semivariogram is isotropic. This parameter is required if passed experimental variogram is stored in a numpy array.

error_estimator
[str, default = 'rmse'] A model error estimation method. Available options are:

- 'rmse': Root Mean Squared Error,
- 'mae': Mean Absolute Error,
- 'bias': Forecast Bias,
- 'smape': Symmetric Mean Absolute Percentage Error.

deviation_weighting
[str, default = "equal"] The name of a method used to weight error at a given lags. Works only with RMSE. Available methods:

- equal: no weighting,
- closest: lags at a close range have bigger weights,
- distant: lags that are further away have bigger weights,
- dense: error is weighted by the number of point pairs within a lag.

auto_update_attributes
[bool, default = True] Update sill, range, model type and nugget based on the best model.

warn_about_set_params: bool, default=True
Should class invoke warning if model parameters has been set during initialization?

verbose
[bool, default = False] Show iteration results.

return_params
[bool, default = True] Return model parameters.

Returns

model_attributes
[Dict] Attributes dict:

```
>>> {
...     'model_name': model_name,
...     'sill': model_sill,
...     'range': model_range,
...     'nugget': model_nugget,
...     'error_type': type_of_error_metrics,
...     'error value': error_value
... }
```

Raises

ValueError
Raised when `sill < 0` or `range < 0` or `range > 1`.

KeyError
Raised when wrong model name(s) are provided by the users.

KeyError
Raised when wrong error type is provided by the users.

Warns

SillOutsideSafeRangeWarning
Warning printed when `max_sill > 1`.

RangeOutsideSafeDistanceWarning
Warning printed when `max_range > 0.5`.

Warning
Model parameters were given during initilization but program is forced to fit the new set of parameters.

Warning
Passed `experimental_variogram` is a numpy array and `direction` parameter is `None`.

`calculate_model_error`(*fitted_values*, *rmse=True*, *bias=True*, *mae=True*, *smape=True*, *deviation_weighting='equal'*)

Method calculates error associated with a difference between the theoretical model and the experimental semivariogram.

Parameters

fitted_values
[numpy array] `[lag, fitted value]`

rmse
[bool, default=True] Root Mean Squared Error of a model.

bias
[bool, default=True] Forecast Bias of a model.

mae
[bool, default=True] Mean Absolute Error of a model.

smape
[bool, default=True] Symmetric Mean Absolute Percentage Error of a model.

deviation_weighting
[str, default = "equal"] The name of a method used to **weight errors at a given lags**. Works only with RMSE. Available methods:

- equal: no weighting,
- closest: lags at a close range have bigger weights,
- distant: lags that are further away have bigger weights,
- dense: error is weighted by the number of point pairs within a lag.

Returns

model_errors
[Dict] Dict with error values per model: rmse, bias, mae, smape.

Raises

MetricsTypeSelectionError
User has set all error types to `False`.

`fit`(*experimental_variogram*, *model_name*, *sill*, *rang*, *nugget=0.0*, *direction=None*, *update_attrs=True*, *warn_about_set_params=True*)

Parameters

experimental_variogram
[ExperimentalVariogram] Prepared Empirical Variogram.

model_name
[str] The name of the model to check. Available models:

- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',
- 'spherical'.

sill
[float, default=0] A value at which dissimilarity is close to its maximum if model is bounded. Otherwise, it is usually close to observations variance.

rang
[float, default=0] The semivariogram range is a distance at which spatial correlation exists. It shouldn't be set at a distance larger than a half of a study extent.

nugget
[float, default=0.] Nugget parameter (bias at a zero distance).

direction
[float, in range [0, 360], default=None] The direction of a semivariogram. If `None` given then semivariogram is isotropic. This parameter is required if passed experimental variogram is stored in a numpy array.

update_attrs
[bool, default=True] Should class attributes be updated?

warn_about_set_params: bool, default=True
Should class invoke warning if model parameters has been set during initialization?

Returns

_theoretical_values, _error: Tuple[numpy array, dict]
`[ theoretical semivariances, {'rmse bias smape mae'}]`

Raises

KeyError
Model type (function) not implemented.

Warns

Warning
Model parameters were given during initialization but program is forced to fit the new set of parameters.

Warning
Passed `experimental_variogram` is a numpy array and `direction` parameter is `None`.

`from_dict`(*parameters*)

Method updates model with a given set of parameters.

Parameters

parameters
[Dict] Dictionary with model's: `'name'`, `'nugget'`, `'sill'`, `'range'`, `'direction'`.

`from_json`(*fname*)

Method reads data from a JSON file.

Parameters

fname
[str]

`plot`(*experimental=True*)

Method plots theoretical curve and (optionally) experimental scatter plot.

Parameters

experimental
[bool] Plot experimental observations.

Raises

AttributeError
Model is not fitted yet, nothing to plot.

`predict`(*distances*)

Method returns a semivariance per distance.

Parameters

distances
[np.ndarray]

Returns

predicted
[np.ndarray]

`to_dict`()

Method exports the theoretical variogram parameters to a dictionary.

Returns

model_parameters
[Dict] Dictionary with model's `'name'`, `'nugget'`, `'sill'`, `'range'` and `'direction'`.

Raises

AttributeError
The model parameters have not been derived yet.

`to_json`(*fname*)

Method stores variogram parameters into a JSON file.

Parameters

fname
[str]

Block

class AggregatedVariogram(*aggregated_data*, *agg_step_size*, *agg_max_range*, *point_support*, *agg_direction=0*, *agg_tolerance=1*, *agg_nugget=0*, *variogram_weighting_method='closest'*, *verbose=False*, *log_process=False*)

Class calculates semivariance of aggregated counts.

Parameters

aggregated_data
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] Blocks with aggregated data.

- Blocks: `Blocks()` class object.
- GeoDataFrame and DataFrame must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- numpy array: `[[block index, centroid x, centroid y, value]]`.

agg_step_size
[float] Step size between lags.

agg_max_range
[float] Maximal distance of analysis.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

- Dict: `{block id: [[point x, point y, value]]}`
- numpy array: `[[block id, x, y, value]]`
- DataFrame and GeoDataFrame: columns = `{x, y, ds, index}`
- PointSupport

agg_direction
[float (in range [0, 360]), default=0] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

agg_tolerance
[float (in range [0, 1]), default=1] If `agg_tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `agg_tolerance` is > `0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `agg_step_size`.
- The minor axis size is `agg_tolerance * agg_step_size`
- The baseline point is at a center of the ellipse.

- `agg_tolerance == 1` creates an omnidirectional semivariogram.

agg_nugget
[float, default = 0] The nugget of blocks data.

variogram_weighting_method
[str, default = "closest"] Method used to weight error at a given lags. Available methods:

- **equal**: no weighting,
- **closest**: lags at a close range have bigger weights,
- **distant**: lags that are further away have bigger weights,
- **dense**: error is weighted by the number of point pairs within a lag - more pairs, lesser weight.

verbose
[bool, default = False] Print steps performed by the algorithm.

log_process
[bool, default = False] Log process info (Level `DEBUG`).

References

[1] Goovaerts P., Kriging and Semivariogram Deconvolution in the Presence of Irregular Geographical Units, Mathematical Geology 40(1), 101-128, 2008

Attributes

aggregated_data
[Union[Blocks, Dict, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] See `aggregared_data` parameter.

agg_step_size
[float] See `agg_step_size` parameter.

agg_max_range
[float] See `agg_max_range` parameter.

agg_lags
[numpy array] Lags calculated as a set of equidistant points from `agg_step_size` to `agg_max_range` with a step of size `agg_step_size`.

agg_tolerance
[float, default = 1] See `agg_tolerance` parameter.

agg_direction
[float, default = 0] See `agg_direction` parameter.

agg_nugget
[float, default = 0] See `agg_nugget` parameter.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]] See the `point_support` parameter.

weighting_method
[str, default = 'closest'] See the `variogram_weighting_method` parameter.

verbose
[bool, default = False] See `verbose` parameter.

log_process
[bool, default = False] See the `log_process` parameter.

experimental_variogram
[ExperimentalVariogram] The experimental variogram calculated from blocks (their centroids).

theoretical_model
[TheoreticalVariogram] The theoretical model derived from blocks' centroids.

inblock_semivariance
[Dict] `{area id: the average inblock semivariance}`

avg_inblock_semivariance
[numpy array] `[lag, average inblocks semivariances, number of blocks within a lag]`

block_to_block_semivariance
[Dict] `{(block i, block j): [distance, semivariance, number of point pairs between blocks]}`

avg_block_to_block_semivariance
[numpy array] `[lag, semivariance, number of point pairs between blocks]`.

regularized_variogram
[numpy array] `[lag, semivariance]`

distances_between_blocks
[Dict] Weighted distances between all blocks: `{block id : [distances to other blocks]}`.

Methods

regularize()	Method performs semivariogram regularization.
show_semivario	Shows experimental variogram, theoretical model, average inblock semivariance, average block to block semivariance and regularized variogram.

`calculate_avg_inblock_semivariance()`

Method calculates the average semivariance within blocks $\gamma_h(v, v)$. The average inblock semivariance is calculated as:

$$\gamma_h(v, v) = \frac{1}{(2 * N(h))} \sum_{a=1}^{N(h)} [\gamma(v_a, v_a) + \gamma(v_{a+h}, v_{a+h})]$$

where:

- $\gamma(v_a, v_a)$ is a semivariance within a block a,
- $\gamma(v_{a+h}, v_{a+h})$ is samivariance within a block at a distance h from the block a.

Returns

avg_inblock_semivariance
[numpy array] `[lag, semivariance, number of block pairs]`

`calculate_avg_semivariance_between_blocks()`

Function calculates semivariance between areas based on their division into smaller blocks. It is $\gamma(v, v_h)$ - semivariogram value between any two blocks separated by the distance h.

Returns

avg_block_to_block_semivariance
[numpy array] The average semivariance between neighboring blocks point-supports: `[lag, semivariance, number of block pairs within a range]`.

Notes

Block-to-block semivariance is calculated as:

$$\gamma(v_a, v_{a+h}) = \frac{1}{P_a P_{a+h}} \sum_{s=1}^{P_a} \sum_{s'=1}^{P_{a+h}} \gamma(u_s, u_{s'})$$

where:

- $\gamma(v_a, v_{a+h})$ - block-to-block semivariance of block a and paired block $a + h$.
- P_a and P_{a+h} - number of support points within block a and block $a + h$.
- $\gamma(u_s, u_{s'})$ - semivariance of point supports between blocks.

Then average block-to-block semivariance is calculated as:

$$\gamma_h(v, v_h) = \frac{1}{N(h)} \sum_{a=1}^{N(h)} \gamma(v_a, v_{a+h})$$

where:

- $\gamma_h(v, v_h)$ - averaged block-to-block semivariances for a lag h,
- $\gamma(v_a, v_{a+h})$ - semivariance of block a and paired block at a distance h.

regularize(*average_inblock_semivariances=None*, *semivariance_between_point_supports=None*, *experimental_block_variogram=None*, *theoretical_block_model=None*)

Method regularizes point support model. Procedure is described in [1].

Parameters

average_inblock_semivariances
[np.ndarray, default = None] The mean semivariance between the blocks. See Notes to learn more.

semivariance_between_point_supports
[np.ndarray, default = None] Semivariance between all blocks calculated from the theoretical model.

experimental_block_variogram
[np.ndarray, default = None] The experimental semivariance between area centroids.

theoretical_block_model
[TheoreticalVariogram, default = None] A modeled variogram.

Returns

regularized_model
[numpy array] `[lag, semivariance, number of point pairs]`

Notes

Function has the form:

$$\gamma_v(h) = \gamma(v, v_h) - \gamma_h(v, v)$$

where:

- $\gamma_v(h)$ - the regularized variogram,
- $\gamma(v, v_h)$ - a variogram value between any two blocks separated by the distance h (calculated from their point support),
- $\gamma_h(v, v)$ - average inblock semivariance between blocks.

Average inblock semivariance between blocks:

$$\gamma_h(v, v) = \frac{1}{(2 * N(h))} \sum_{a=1}^{N(h)} [\gamma(v_a, v_a) + \gamma(v_{a+h}, v_{a+h})]$$

where $\gamma(v_a, v_a)$ and $\gamma(v_{a+h}, v_{a+h})$ are inblock semivariances of block a and block $a + h$ separated by the distance h.

References

[1] Goovaerts P., Kriging and Semivariogram Deconvolution in the Presence of Irregular Geographical Units,
Mathematical Geology 40(1), 101-128, 2008

`regularize_variogram()`

Function regularizes semivariograms.

Returns

reg_variogram
[numpy array] `[lag, semivariance]`

Raises

ValueError
Semivariance at a given lag is below zero.

Notes

Regularized semivariogram is a difference between the average block to block semivariance $\gamma(v, v_h)$ and the average inblock semivariances $\gamma h(v, v)$ at a given lag h.

$$\gamma_v(h) = \gamma(v, v_h) - \gamma h(v, v)$$

`show_semivariograms()`

Method plots:

- experimental variogram,
- theoretical model,
- average inblock semivariance,
- average block-to-block semivariance,

- regularized variogram.

Raises

AttributeError
The semivariogram regularization process has not been performed.

Deconvolution

class Deconvolution(*verbose=True*, *store_models=False*)

Class performs deconvolution of semivariogram of areal data. Whole procedure is based on the iterative process described in: [1].

Steps to regularize semivariogram:

- initialize your object (no parameters),
- use `fit()` method to build initial point support model,
- use `transform()` method to perform semivariogram regularization,
- save deconvoluted semivariogram model with `export_model()` method.

References

[1] Goovaerts P., Kriging and Semivariogram Deconvolution in the Presence of Irregular Geographical Units, Mathematical Geology 40(1), 101-128, 2008

Examples

```
>>> dcv = Deconvolution(verbose=True)
>>> dcv.fit(agg_dataset=...,
...         point_support_dataset=...,
...         agg_step_size=...,
...         agg_max_range=...,
...         variogram_weighting_method='closest')
>>> dcv.transform(max_iters=5)
>>> dcv.plot_variograms()
>>> dcv.plot_deviations()
>>> dcv.plot_weights()
>>> dcv.export_model('results.csv')
```

Attributes

ps
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

- Dict: `{block id: [[point x, point y, value]]}`
- numpy array: `[[block id, x, y, value]]`
- DataFrame and GeoDataFrame: columns = `{x, y, ds, index}`
- PointSupport

agg
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] Blocks with aggregated data.

- Blocks: `Blocks()` class object.
- GeoDataFrame and DataFrame must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used and optional.
- numpy array: `[[block index, centroid x, centroid y, value]]`.

initial_regularized_variogram
[numpy array] `[lag, semivariance]`

initial_theoretical_agg_model
[TheoreticalVariogram]

initial_theoretical_model_prediction
[numpy array] `[lag, semivariance]`

initial_experimental_variogram
[numpy array] `[lag, semivariance, number of pairs]`

final_theoretical_model
[TheoreticalVariogram]

final_optimal_variogram
[numpy array] `[lag, semivariance]`

agg_step
[float] Step size between lags.

agg_rng
[float] Maximal distance of analysis.

ranges
[numpy array] `np.arange(agg_step, agg_rng, agg_step)`

direction
[float (in range [0, 360])] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1])] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is > `0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `step_size`.
- The minor axis size is `tolerance * step_size`
- The baseline point is at a center of the ellipse.
- The `tolerance == 1` creates an omnidirectional semivariogram.

weighting_method
[str] Method used to weight error at a given lags. Available methods:

- **equal**: no weighting,
- **closest**: lags at a close range have bigger weights,
- **distant**: lags that are further away have bigger weights,
- **dense**: error is weighted by the number of point pairs within a lag - more pairs, lesser weight.

deviations
[List] List of deviations per iteration. The first element is the initial deviation.

weights
[List] List of weights applied to lags in each iteration.

verbose
[bool] Should algorithm `print()` process steps into a terminal.

store_models
[bool] Should theoretical and regularized models be stored after each iteration.

theoretical_models
[List] List with theoretical models parameters.

regularized_models
[List] List with numpy arrays with regularized models.

Methods

fit()	Fits areal data and the point support data into a model, initializes the experimental semivariogram, the theoretical semivariogram model, regularized point support model, and deviation.
transform()	Performs semivariogram regularization.
fit_transfo	Performs fit() and transform() at one time.
export_mode	Exports regularized (or fitted) model.
plot_variog	Plots semivariances before and after regularization.
plot_deviat	Plots each deviation divided by the initial deviation.
plot_weigh	Plots the mean weight value per lag.

`export_model`(*fname*)

Function exports final theoretical model.

Parameters

fname
[str] File name for model parameters (nugget, sill, range, model type)

Raises

RunetimeError
A model hasn't been transformed yet.

`fit`(*agg_dataset*, *point_support_dataset*, *agg_step_size*, *agg_max_range*, *agg_nugget=None*, *agg_direction=None*, *agg_tolerance=1*, *variogram_weighting_method='closest'*, *model_name='safe'*, *model_types=None*)

Function fits given areal data variogram into point support variogram - it is the first step of regularization process.

Parameters

agg_dataset
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] Blocks with aggregated data.

- Blocks: `Blocks()` class object.
- GeoDataFrame and DataFrame must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- numpy array: `[[block index, centroid x, centroid y, value]]`.

point_support_dataset
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

- Dict: `{block id: [[point x, point y, value]]}`
- numpy array: `[[block id, x, y, value]]`
- DataFrame and GeoDataFrame: columns = `{x, y, ds, index}`
- PointSupport

agg_step_size
[float] Step size between lags.

agg_max_range
[float] Maximal distance of analysis.

agg_nugget
[float, default = 0] The nugget of a data.

agg_direction
[float (in range [0, 360]), optional, default=0] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

agg_tolerance
[float (in range [0, 1]), optional, default=1] If `agg_tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `agg_tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `agg_step_size`.
- The minor axis size is `agg_tolerance * agg_step_size`
- The baseline point is at a center of the ellipse.
- `agg_tolerance == 1` creates an omnidirectional semivariogram.

variogram_weighting_method
[str, default = "closest"] Method used to weight error at a given lags. Available methods:

- **equal**: no weighting,
- **closest**: lags at a close range have bigger weights,

- **distant**: lags that are further away have bigger weights,
- **dense**: error is weighted by the number of point pairs within a lag - more pairs, lesser weight.

model_name
[str, default='safe'] The name of the model to check or special command for multiple models. Available models:

- 'all' - the same as list with all models,
- 'safe' - ['linear', 'power', 'spherical'],
- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',
- 'spherical'.

model_types
[List, default = None] The list with model names to check excluding 'all' and 'safe' names.

`fit_transform`(*agg_dataset*, *point_support_dataset*, *agg_step_size*, *agg_max_range*, *agg_nugget=None*, *agg_direction=None*, *agg_tolerance=1*, *variogram_weighting_method='closest'*, *model_name='safe'*, *model_types=None*, *max_iters=25*, *limit_deviation_ratio=0.1*, *minimum_deviation_decrease=0.01*, *reps_deviation_decrease=3*)

Method performs `fit()` and `transform()` operations at once.

Parameters

agg_dataset
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] Blocks with aggregated data.

- Blocks: `Blocks()` class object.
- GeoDataFrame and DataFrame must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- numpy array: `[[block index, centroid x, centroid y, value]]`.

point_support_dataset
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

- Dict: `{block id: [[point x, point y, value]]}`
- numpy array: `[[block id, x, y, value]]`
- DataFrame and GeoDataFrame: columns = `{x, y, ds, index}`
- PointSupport

agg_step_size
[float] Step size between lags.

agg_max_range
[float] Maximal distance of analysis.

agg_nugget
[float, default = None] The nugget of a dataset.

agg_direction
[float (in range [0, 360]), default=0] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

agg_tolerance
[float (in range [0, 1]), optional, default=1] If `agg_tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `agg_tolerance` is > `0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `agg_step_size`.
- The minor axis size is `agg_tolerance * agg_step_size`
- The baseline point is at a center of the ellipse.
- `agg_tolerance == 1` creates an omnidirectional semivariogram.

variogram_weighting_method
[str, default = "closest"] Method used to weight error at a given lags. Available methods:

- **equal**: no weighting,
- **closest**: lags at a close range have bigger weights,
- **distant**: lags that are further away have bigger weights,
- **dense**: error is weighted by the number of point pairs within a lag - more pairs, lesser weight.

model_name
[str, default='safe'] The name of the model to check or special command for multiple models. Available models:

- 'all' - the same as list with all models,
- 'safe' - ['linear', 'power', 'spherical'],
- 'circular',
- 'cubic',
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',
- 'spherical'.

model_types
[List, default = None] The list with model names to check excluding 'all' and 'safe' names.

max_iters
[int, default = 25] Maximum number of iterations.

limit_deviation_ratio
[float, default = 0.01] Minimal ratio of model deviation to initial deviation when algorithm is stopped. Parameter must be set within the limits `(0, 1)`.

minimum_deviation_decrease
[float, default = 0.001] The minimum ratio of the difference between model deviation and optimal deviation to the optimal deviation: `|dev - opt_dev| / opt_dev`. The parameter must be set within the limits `(0, 1)`.

reps_deviation_decrease
[int, default = 3] How many repetitions of small deviation decrease before termination of the algorithm.

`plot_variograms()`

Function shows experimental semivariogram, theoretical semivariogram and regularized semivariogram after semivariogram regularization with `transform()` method.

`transform`(*max_iters=25*, *limit_deviation_ratio=0.1*, *minimum_deviation_decrease=0.01*, *reps_deviation_decrease=3*)

Method performs semivariogram regularization after model fitting.

Parameters

max_iters
[int, default = 25] Maximum number of iterations.

limit_deviation_ratio
[float, default = 0.1] Minimal ratio of model deviation to initial deviation when algorithm is stopped. Parameter must be set within the limits `(0, 1)`.

minimum_deviation_decrease
[float, default = 0.01] The minimum ratio of the difference between model deviation and optimal deviation to the optimal deviation: `|dev - opt_dev| / opt_dev`. Parameter must be set within the limits `(0, 1)`.

reps_deviation_decrease
[int, default = 3] How many repetitions of small deviation decrease before termination of the algorithm.

Raises

AttributeError
`initial_regularized_model` is undefined (user didn't perform `fit()` method).

ValueError
`limit_deviation_ratio` or `minimum_deviation_decrease` parameters <= `0` or >= `1`.

Indicator

class IndicatorVariogramData(*input_array*, *number_of_thresholds*)

Class describes indicator variogram data.

Parameters

input_array
[numpy array, list, tuple] Coordinates and their values: `(pt x, pt y, value)`

number_of_thresholds: int
The number of thresholds to model data.

See also:

`ExperimentalIndicatorVariogram`
Class that calculates experimental variograms for each indicator.

Attributes

input_array
[numpy array, list, tuple] Coordinates and their values: `(pt x, pt y, value)`

n_thresholds: int
The number of thresholds to model data.

thresholds
[numpy array] The 1D numpy array with thresholds.

ids
[numpy array] The numpy array with `[coordinate_x, coordinate_y, threshold_0, ..., threshold_n]`.

---

class ExperimentalIndicatorVariogram(*input_array*, *number_of_thresholds*, *step_size*, *max_range*, *weights=None*, *direction=None*, *tolerance=1.0*, *method='t'*, *fit=True*)

Class describes Experimental Indicator Variogram models.

Parameters

input_array
[numpy array, list, tuple] Coordinates and their values: `(pt x, pt y, value)`

number_of_thresholds: int
The number of thresholds to model data.

step_size
[float] The distance between lags within each points are included in the calculations.

max_range
[float] The maximum range of analysis.

weights
[numpy array, default=None] Weights assigned to points, index of weight must be the same as index of point.

direction
[float (in range [0, 360]), default=None] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), default=1] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y axis and direction parameter. If `tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance.

- The major axis size == `step_size`.
- The minor axis size is `tolerance * step_size`
- The baseline point is at a center of the ellipse.
- The `tolerance == 1` creates an omnidirectional semivariogram.

method
[str, default = triangular] The method used for neighbors selection. Available methods:

- "triangle" or "t", default method where a point neighbors are selected from a triangular area,
- "ellipse" or "e", the most accurate method but also the slowest one.

fit
[bool, default = True] Should models be fitted in the class initialization?

References

Goovaerts P. AUTO-IK: a 2D indicator kriging program for automated non-parametric modeling of local uncertainty in earth sciences. DOI: TODO

Attributes

ds
[IndicatorVariogramData] Prepared indicator data.

step_size
[float] Derived from the `step_size` parameter.

max_range
[float] Derived from the `max_range` parameter.

weights
[numpy array] Derived from the `weights` parameter.

direction
[float] Derived from the `direction` parameter.

tolerance
[float] Derived from the `tolerance` parameter.

method
[str] Derived from the `method` parameter.

experimental_models
[List] The `[threshold, experimental_variogram]` pairs.

Methods

fit()	Fits indicators to experimental variograms.
show()	Show experimental variograms for each indicator.

`fit()`

Function fits indicators to models and updates class models.

`show()`

Function shows generated experimental variograms for each indicator.

`class IndicatorVariograms(`*experimental_indicator_variogram*`)`

Class models indicator variograms for all indices.

Parameters

experimental_indicator_variogram
[ExperimentalIndicatorVariogram] Fitted experimanetal variograms with indicators for each threshold.

Attributes

experimental_indicator_variogram
[ExperimentalIndicatorVariogram] See `experimental_indicator_variogram` parameter.

theoretical_indicator_variograms
[Dict] Dictionary with fitted theoretical models for each threshold.

Methods

fit()	Fits theoretical models to experimental variograms.
show()	Shows experimental and theoretical variograms for each threshold.

`fit(`*model_name='linear'*, *nugget=0*, *rang=None*, *min_range=0.1*, *max_range=0.5*, *number_of_ranges=64*, *sill=None*, *min_sill=0.5*, *max_sill=1.5*, *number_of_sills=64*, *direction=None*, *error_estimator='rmse'*, *deviation_weighting='equal'*, *auto_update_attributes=True*, *warn_about_set_params=True*, *verbose=False*`)`

Method tries to find the optimal range, sill and model (function) of the theoretical semivariogram.

Parameters

model_name
[str, default = "linear"] The name of a modeling function. Available models:

- 'all',
- 'safe' : linear, power and spherical models,
- 'exponential',
- 'gaussian',
- 'linear',
- 'power',

- 'spherical'.

nugget
[float, default = 0] Nugget (bias) of a variogram. Default value is 0.

rang
[float, optional] If given, then range is fixed to this value.

min_range
[float, default = 0.1] The minimal fraction of a variogram range, `0 < min_range <= max_range`.

max_range
[float, default = 0.5] The maximum fraction of a variogram range, `min_range <= max_range <= 1`. Parameter `max_range` greater than **0.5** raises warning.

number_of_ranges
[int, default = 64] How many equally spaced ranges are tested between `min_range` and `max_range`.

sill
[float, default = None] If given, then sill is fixed to this value.

min_sill
[float, default = 0] The minimal fraction of the variogram variance at lag 0 to find a sill, `0 <= min_sill <= max_sill`.

max_sill
[float, default = 1] The maximum fraction of the variogram variance at lag 0 to find a sill. It *should be* lower or equal to 1. It is possible to set it above 1, but then warning is printed.

number_of_sills
[int, default = 64] How many equally spaced sill values are tested between `min_sill` and `max_sill`.

direction
[float, in range [0, 360], default=None] The direction of a semivariogram. If `None` given then semivariogram is isotropic. This parameter is required if passed experimental variogram is stored in a numpy array.

error_estimator
[str, default = 'rmse'] A model error estimation method. Available options are:

- 'rmse': Root Mean Squared Error,
- 'mae': Mean Absolute Error,
- 'bias': Forecast Bias,
- 'smape': Symmetric Mean Absolute Percentage Error.

deviation_weighting
[str, default = "equal"] The name of a method used to weight error at a given lags. Works only with RMSE. Available methods:

- equal: no weighting,
- closest: lags at a close range have bigger weights,
- distant: lags that are further away have bigger weights,
- dense: error is weighted by the number of point pairs within a lag.

auto_update_attributes
[bool, default = True] Update sill, range, model type and nugget based on the best model.

warn_about_set_params: bool, default=True
Should class invoke warning if model parameters has been set during initialization?

verbose
[bool, default = False] Show iteration results.

Raises

ValueError
Raised when `sill < 0` or `range < 0` or `range > 1`.

KeyError
Raised when wrong model name(s) are provided by the users.

KeyError
Raised when wrong error type is provided by the users.

Warns

SillOutsideSafeRangeWarning
Warning printed when `max_sill > 1`.

RangeOutsideSafeDistanceWarning
Warning printed when `max_range > 0.5`.

Warning
Model parameters were given during initilization but program is forced to fit the new set of parameters.

Warning
Passed `experimental_variogram` is a numpy array and `direction` parameter is `None`.

`show`(*subplots=False*)

Method plots experimental and theoretical variograms.

Parameters

subplots
[bool, default = False] If `True` then each indicator variogram is plotted on a separate plot. Otherwise, all variograms are plotted on a scatter single plot.

2.5.6 Kriging

Point Kriging

`kriging`(*observations*, *theoretical_model*, *points*, *how='ok'*, *neighbors_range=None*, *no_neighbors=4*, *use_all_neighbors_in_range=False*, *sk_mean=None*, *allow_approx_solutions=False*, *number_of_workers=1*, *show_progress_bar=True*)

Function manages Ordinary Kriging and Simple Kriging predictions.

Parameters

observations
[numpy array] Known points and their values.

theoretical_model
[TheoreticalVariogram] Fitted variogram model.

points
[numpy array] Coordinates with missing values (to estimate results).

how
[str, default='ok']

Select what kind of kriging you want to perform:

- 'ok': ordinary kriging,
- 'sk': simple kriging - if it is set then `sk_mean` parameter must be provided.

neighbors_range
[float, default=None] The maximum distance where we search for neighbors. If `None` is given then range is selected from the `theoretical_model` `rang` attribute.

no_neighbors
[int, default = 4] The number of the **n-closest neighbors** used for interpolation.

use_all_neighbors_in_range
[bool, default = False] `True`: if the real number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors` parameter then take all of them anyway.

sk_mean
[float, default=None] The mean value of a process over a study area. Should be know before processing. That's why Simple Kriging has a limited number of applications. You must have multiple samples and well-known area to know this parameter.

allow_approx_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

number_of_workers
[int, default=1] How many processing units can be used for predictions. Increase it only for a very large number of interpolated points (~10k+).

show_progress_bar
[bool, default=True] Show progress bar of predictions.

Returns

: numpy array
Predictions `[predicted value, variance error, longitude (x), latitude (y)]`

`ordinary_kriging`(*theoretical_model*, *known_locations*, *unknown_location*, *neighbors_range=None*, *no_neighbors=4*, *use_all_neighbors_in_range=False*, *allow_approximate_solutions=False*)

Function predicts value at unknown location with Ordinary Kriging technique.

Parameters

theoretical_model
[TheoreticalVariogram] A trained theoretical variogram model.

known_locations
[numpy array] The known locations.

unknown_location
[Union[List, Tuple, numpy array]] Point where you want to estimate value `(x, y) <-> (lon, lat)`.

neighbors_range
[float, default=None] The maximum distance where we search for neighbors. If `None` is given then range is selected from the `theoretical_model` `rang` attribute.

no_neighbors
[int, default = 4] The number of the **n-closest neighbors** used for interpolation.

use_all_neighbors_in_range
[bool, default = False] `True`: if the real number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors` parameter then take all of them anyway.

allow_approximate_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

: numpy array
`[predicted value, variance error, longitude (x), latitude (y)]`

Raises

RunetimeError
Singularity matrix in a Kriging system.

`simple_kriging`(*theoretical_model*, *known_locations*, *unknown_location*, *process_mean*, *neighbors_range=None*, *no_neighbors=1*, *use_all_neighbors_in_range=False*, *allow_approximate_solutions=False*)

Function predicts value at unknown location with Ordinary Kriging technique.

Parameters

theoretical_model
[TheoreticalVariogram] A trained theoretical variogram model.

known_locations
[numpy array] The known locations.

unknown_location
[Union[List, Tuple, numpy array]] Point where you want to estimate value `(x, y) <-> (lon, lat)`.

process_mean
[float] The mean value of a process over a study area. Should be know before processing. That's why Simple Kriging has a limited number of applications. You must have multiple samples and well-known area to know this parameter.

neighbors_range
[float, default=None] The maximum distance where we search for neighbors. If `None` is given then range is selected from the `theoretical_model` `rang` attribute.

no_neighbors
[int, default = 4] The number of the **n-closest neighbors** used for interpolation.

use_all_neighbors_in_range
[bool, default = False] `True`: if the real number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors` parameter then take all of them anyway.

allow_approximate_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

: numpy array
`[predicted value, variance error, longitude (x), latitude (y)]`

Raises

RunetimeError
Singularity matrix in a Kriging system.

Block - Poisson Kriging

`centroid_poisson_kriging`(*semivariogram_model*, *blocks*, *point_support*, *unknown_block*, *unknown_block_point_support*, *number_of_neighbors*, *is_weighted_by_point_support=True*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*, *allow_approximate_solutions=False*)

Function performs centroid-based Poisson Kriging of blocks (areal) data.

Parameters

semivariogram_model
[TheoreticalVariogram] A fitted variogram.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid_x, centroid_y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,
- `DataFrame` and `GeoDataFrame`: `columns={x_col, y_col, ds, index}`,
- `PointSupport`.

unknown_block
[numpy array] `[index, centroid x, centroid y]`

unknown_block_point_support
[numpy array] Points within block `[[x, y, point support value]]`

number_of_neighbors
[int] The minimum number of neighbours that can potentially affect block.

is_weighted_by_point_support
[bool, default = True] Are distances between blocks weighted by the point support?

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

allow_approximate_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

results
[List] `[unknown block index, prediction, error]`

Raises

ValueError
Prediction or prediction error are negative.

Warns

ExperimentalFeatureWarning
Directional Kriging is in early-phase and may contain bugs.

`area_to_area_pk`(*semivariogram_model*, *blocks*, *point_support*, *unknown_block*, *unknown_block_point_support*, *number_of_neighbors*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*, *log_process=True*)

Function predicts areal value in an unknown location based on the area-to-area Poisson Kriging

Parameters

semivariogram_model
[TheoreticalVariogram] A fitted variogram.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid_x, centroid_y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,

- `DataFrame` and `GeoDataFrame`: `columns={x_col, y_col, ds, index}`,
- `PointSupport`.

unknown_block
[numpy array] `[index, centroid x, centroid y]`

unknown_block_point_support
[numpy array] Points within block `[[x, y, point support value]]`

number_of_neighbors
[int] The minimum number of neighbours that can potentially affect block.

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

log_process
[bool, default=True] Log process info and debug info.

Returns

results
[List] `[unknown block index, prediction, error]`

Raises

ValueError
Prediction or prediction error are negative.

Warns

ExperimentalFeatureWarning
Directional Kriging is in early-phase and may contain bugs.

`area_to_point_pk`(*semivariogram_model*, *blocks*, *point_support*, *unknown_block*, *unknown_block_point_support*, *number_of_neighbors*, *max_range=None*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*, *err_to_nan=True*)

Function predicts areal value in the unknown location based on the area-to-area Poisson Kriging

Parameters

semivariogram_model
[TheoreticalVariogram] A fitted variogram.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid_x, centroid_y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- Dict: {block id: [[point x, point y, value]]},
- numpy array: [[block id, x, y, value]],
- DataFrame and GeoDataFrame: columns={x_col, y_col, ds, index},
- PointSupport.

unknown_block
[numpy array] [index, centroid x, centroid y]

unknown_block_point_support
[numpy array] Points within block [[x, y, point support value]]

number_of_neighbors
[int] The minimum number of neighbours that can potentially affect block.

max_range
[float , default=None] The maximum distance to search for a neighbors, if None given then algorithm uses the theoretical variogram's range.

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

err_to_nan
[bool, default=True] ValueError to NaN.

Returns

results
[List] [(unknown point coordinates), prediction, error]

Raises

ValueError
Prediction or prediction error are negative.

Warns

ExperimentalFeatureWarning
Directional Kriging is in early-phase and may contain bugs.

Indicator Kriging

class IndicatorKriging(*datapoints*, *indicator_variograms*, *unknown_locations*, *kriging_type='ok'*, *process_mean=None*, *neighbors_range=None*, *no_neighbors=4*, *use_all_neighbors_in_range=False*, *allow_approximate_solutions=False*, *get_expected_values=True*)

Class performs indicator kriging.

Parameters

datapoints
[numpy ndarray] The known locations [x, y, value].

indicator_variograms
[IndicatorVariograms] Modeled variograms for each threshold.

unknown_locations
[numpy ndarray] Points where we want to estimate value (x, y) <-or-> (lon, lat).

kriging_type
[str, default = 'ok'] Type of kriging to perform. Possible values: 'ok' - ordinary kriging, 'sk' - simple kriging.

process_mean
[float] The mean value of a process over a study area. Should be know before processing. That's why Simple Kriging has a limited number of applications. You must have multiple samples and well-known area to know this parameter.

neighbors_range
[float, default=None] The maximum distance where we search for neighbors. If `None` is given then range is selected from the `theoretical_model` `rang` attribute.

no_neighbors
[int, default = 4] The number of the **n-closest neighbors** used for interpolation.

use_all_neighbors_in_range
[bool, default = False] `True`: if the real number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors` parameter then take all of them anyway.

allow_approximate_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

get_expected_values
[bool, default=True] If `True` then expected values and variances are calculated.

Attributes

thresholds
[numpy ndarray] Thresholds used for indicator kriging.

coordinates
[numpy ndarray] Coordinates of unknown locations.

indicator_predictions
[numpy ndarray] Indicator kriging predictions for each threshold and each unknown location.

expected_values
[numpy ndarray] Expected values derived from `indicator_predictions` for each unknown location.

variances
[numpy ndarray] Variances derived from `indicator_predictions` for each unknown location.

Methods

get_indicator_ma	Returns dictionary with thresholds and indicator maps for each of them.
get_expected_valu	Returns two arrays: one array with coordinates and expected values, and the second with coordinates and variances.

`get_expected_values_maps()`

Method returns expected values and variances for each threshold.

Returns

expected_values, variances
[numpy ndarray, numpy ndarray] Expected values and variances.

`get_indicator_maps()`

Method returns indicator map for each threshold.

Returns

indicator_maps
[Dict] Indicator map for each threshold.

2.5.7 Inverse Distance Weighting

`inverse_distance_weighting(known_points, unknown_location, number_of_neighbours=-1, power=2.0)`

Inverse Distance Weighting with a given set of points and an unknown location.

Parameters

known_points
[numpy array] The MxN array, where **M** is a number of rows (points) and **N** is the number of columns, where the last column represents a value of a known point. (It could be **(N-1)**-dimensional data).

unknown_location
[Iterable] Array or list with coordinates of the unknown point. It's length is N-1 (number of dimensions). The unknown location *shape* should be the same as the `known_points` parameter *shape*, if not, then new dimension is added once - vector of points `[x, y]` becomes `[[x, y]]` for 2-dimensional data.

number_of_neighbours
[int, default = -1] If default value **(-1)** then all known points will be used to estimate value at the unknown location. Can be any number within the limits `[2, len(known_points)]`,

power
[float, default = 2.] Power value must be larger or equal to 0. It controls weight assigned to each known point. Larger power means stronger influence of the closest neighbors, but it decreases quickly.

Returns

result
[float] The estimated value.

Raises

ValueError
Power parameter set to be smaller than 0.

ValueError
Less than 2 neighbours or more than the number of `known_points` neighbours are given in the `number_of_neighbours` parameter.

2.5.8 Pipelines

Data download

Available datasets are stored in the **pyinterpolate-datasets** package: https://pypi.org/project/pyinterpolate-datasets/2023.0.0/

Kriging-based processes

BlockFilter
alias of BlockPK

---

class BlockPK(*semivariogram_model*, *blocks*, *point_support*, *kriging_type='ata'*)

Class is an object that can be used for Area-to-Area, Area-to-Point, Centroid-based Poisson Kriging regularization.

Parameters

semivariogram_model
[TheoreticalVariogram] The fitted variogram model.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,
- `DataFrame` and `GeoDataFrame`: `columns={x, y, ds, index}`,
- `PointSupport`.

kriging_type
[str, default='ata']

A type of Poisson Kriging operation. Available methods:

- `'ata'`: Area-to-Area Poisson Kriging.
- `'atp'`: Area-to-Point Poisson Kriging.
- `'cb'`: Centroid-based Poisson Kriging.

Attributes

semivariogram_model
[TheoreticalVariogram] See the `semivariogram_model` parameter.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] See the `blocks` parameter.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]] See the `point_support` parameter.

kriging_type
[str, default='ata'] See the `kriging_type` parameter.

geo_ds
[geopandas GeoDataFrame] A regularized set of blocks: `['id', 'geometry', 'reg.est', 'reg.err', 'rmse']`

statistics
[Dict]

A dictionary with two keys:

- `'RMSE'`: root mean squared error of regularization,
- `'time'`: time (in seconds) of the regularization process.

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

Methods

regularize()	Regularize blocks (you should use it for a data deconvolution - with ATP PK, or for a data filtering - with ATA, C-B PK).

regularize(*number_of_neighbors*, *data_crs=None*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*)

Function regularizes whole dataset and creates new values and error maps based on the kriging type. Function does not predict unknown and missing values, areas with `NaN` values are skipped.

Parameters

number_of_neighbors
[int] The minimum number of neighbours that potentially affect block.

data_crs
[str, default=None] Data crs, look into: https://geopandas.org/projections.html. If None given then returned GeoDataFrame doesn't have a crs.

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

Returns

regularized
[gpd.GeoDataFrame] Regularized set of blocks: `['id', 'geometry', 'reg.est', 'reg.err', 'rmse']`

class BlockToBlockKrigingComparison(*variogram*, *blocks*, *point_support*, *no_of_neighbors=16*, *neighbors_range=None*, *simple_kriging_mean=None*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*, *training_set_frac=0.8*, *allow_approx_solutions=False*, *iters=20*)

Class compares different block kriging models and techniques.

Parameters

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,
- `DataFrame` and `GeoDataFrame`: `columns={x, y, ds, index}`,
- `PointSupport`.

no_of_neighbors
[int, default = 16] The maximum number of n-closest neighbors used for interpolation.

neighbors_range
[float, default = None] Maximum distance where we search for point neighbors. If None given then range is selected from the `theoretical_model rang` attribute. If algorithms picks less neighbors than `no_of_neighbors` within the range then additional points are selected outside the `neighbors_range`.

simple_kriging_mean
[float, default = None] The mean value of a process over a study area. Should be known before processing. If not provided then Simple Kriging estimator is skipped.

raise_when_negative_prediction
[bool, default = True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

training_set_frac
[float, default = 0.8] How many values sampled as a known points set in each iteration. Could be any fraction within `(0:1)` range.

allow_approx_solutions
[bool, default = False] Allows the approximation of kriging weights based on the OLS algorithm. Not recommended to set to `True` if you don't know what you are doing!

iters
[int, default = 20] How many tests to perform over random samples of a data.

Attributes

variogram
[TheoreticalVariogram] See `variogram` parameter.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]] See `blocks` parameter.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]] See `point_support` parameter.

no_of_neighbors
[int] See `no_of_neighbors` parameter.

neighbors_range
[float] See `neighbors_range` parameter.

simple_kriging_mean
[float] See `simple_kriging_mean` parameter.

raise_when_negative_prediction
[bool, default = True] See `raise_when_negative_prediction` parameter.

raise_when_negative_error
[bool, default=True] See `raise_when_negative_error` parameter.

training_set_frac
[float] See `training_set_frac` parameter.

iters
[int] See `iters` parameter.

common_indexes
[Set] Indexes that are common for blocks and point support.

training_set_indexes
[List[List]] List of lists with indexes used in a random sampling for a training set.

results
[Dict] Results for each type of Block Kriging method.

Methods

run_tests()	Compares different types of Kriging, returns Dict with the mean root mean squared error of each iteration.

`run_tests()`

Method compares ordinary, simple, area-to-area and centroid-based block Poisson Kriging.

`smooth_blocks`(*semivariogram_model*, *blocks*, *point_support*, *number_of_neighbors*, *max_range=None*, *crs=None*, *raise_when_negative_prediction=True*, *raise_when_negative_error=True*, *err_to_nan=True*)

Function smooths blocks data into their point support values.

Parameters

semivariogram_model
[TheoreticalVariogram] The regularized variogram.

blocks
[Union[Blocks, gpd.GeoDataFrame, pd.DataFrame, np.ndarray]]

Blocks with aggregated data.

- `Blocks`: `Blocks()` class object.
- `GeoDataFrame` and `DataFrame` must have columns: `centroid.x, centroid.y, ds, index`. Geometry column with polygons is not used.
- `numpy array`: `[[block index, centroid x, centroid y, value]]`.

point_support
[Union[Dict, np.ndarray, gpd.GeoDataFrame, pd.DataFrame, PointSupport]]

The point support of polygons.

- `Dict`: `{block id: [[point x, point y, value]]}`,
- `numpy array`: `[[block id, x, y, value]]`,
- `DataFrame` and `GeoDataFrame`: `columns={x, y, ds, index}`,
- `PointSupport`.

number_of_neighbors
[int] The minimum number of neighbours that potentially affect block.

max_range
[float, default=None] The maximum distance to search for neighbors.

crs
[Any, default=None] CRS of data.

raise_when_negative_prediction
[bool, default=True] Raise error when prediction is negative.

raise_when_negative_error
[bool, default=True] Raise error when prediction error is negative.

err_to_nan
[bool, default=True] `ValueError` to `NaN`.

Returns

results
[gpd.GeoDataFrame] Columns = `[area_id, geometry (Point), prediction, error]`.

2.5.9 Visualization

interpolate_raster(*data*, *dim=1000*, *number_of_neighbors=4*, *semivariogram_model=None*, *direction=None*, *tolerance=None*, *allow_approx_solutions=True*)

Function interpolates raster from data points using ordinary kriging.

Parameters

data
[numpy array] `[coordinate x, coordinate y, value]`.

dim
[int] Number of pixels (points) of a larger dimension (it could be width or height). Ratio is preserved.

number_of_neighbors
[int, default=16] Number of points used to interpolate data.

semivariogram_model
[TheoreticalVariogram, default=None] Variogram model, if not provided then it is estimated from a given dataset.

direction
[float (in range [0, 360]), optional] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), optional] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y-axis and direction parameter. If `tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance:

- the major axis size == `step_size`,
- the minor axis size is `tolerance * step_size`,
- the baseline point is at a center of the ellipse,
- the `tolerance == 1` creates an omnidirectional semivariogram.

allow_approx_solutions
[bool, default=True] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

raster_dict
[Dict] A dictionary with keys:

- **'result'**: numpy array of interpolated values,
- **'error'**: numpy array of interpolation errors,
- **'params':**

- 'pixel size',
- 'min x',
- 'max x',
- 'min y',
- 'max y'

`to_tiff`(*data*, *dir_path*, *fname=''*, *dim=1000*, *number_of_neighbors=4*, *semivariogram_model=None*, *direction=None*, *tolerance=None*, *allow_approx_solutions=True*)

Function interpolates raster from data points using ordinary kriging and stores output results in tiff and tfw files.

Parameters

data
[numpy array] `[coordinate x, coordinate y, value]`.

dir_path
[str] Path to directory where output files will be stored.

fname
[str, default=''] Suffix of the output `*results.tiff` and `*error.tiff` files.

dim
[int] Number of pixels (points) of a larger dimension (it could be width or height). Ratio is preserved.

number_of_neighbors
[int, default=16] Number of points used to interpolate data.

semivariogram_model
[TheoreticalVariogram, default=None] Variogram model, if not provided then it is estimated from a given dataset.

direction
[float (in range [0, 360]), optional] Direction of semivariogram, values from 0 to 360 degrees:

- 0 or 180: is E-W,
- 90 or 270 is N-S,
- 45 or 225 is NE-SW,
- 135 or 315 is NW-SE.

tolerance
[float (in range [0, 1]), optional] If `tolerance` is 0 then points must be placed at a single line with the beginning in the origin of the coordinate system and the direction given by y-axis and direction parameter. If `tolerance` is `> 0` then the bin is selected as an elliptical area with major axis pointed in the same direction as the line for 0 tolerance:

- the major axis size == `step_size`,
- the minor axis size is `tolerance * step_size`,
- the baseline point is at a center of the ellipse,
- the `tolerance == 1` creates an omnidirectional semivariogram.

allow_approx_solutions
[bool, default=True] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This

parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

files: Tuple[str, str]
Tuple of two strings: path to tiff file with interpolated data and path to tiff file with interpolation errors.

2.5.10 Validation

Cross-validation

`validate_kriging`(*points*, *theoretical_model*, *how='ok'*, *neighbors_range=None*, *no_neighbors=4*, *use_all_neighbors_in_range=False*, *sk_mean=None*, *allow_approx_solutions=False*)

Function performs cross-validation of kriging models.

Parameters

points
[numpy array] Known points and their values.

theoretical_model
[TheoreticalVariogram] Fitted variogram model.

how
[str, default='ok']

Select what kind of kriging you want to perform:

- 'ok': ordinary kriging,
- 'sk': simple kriging - if it is set then `sk_mean` parameter must be provided.

neighbors_range
[float, default=None] The maximum distance where we search for neighbors. If `None` is given then range is selected from the `theoretical_model` `rang` attribute.

no_neighbors
[int, default = 4] The number of the **n-closest neighbors** used for interpolation.

use_all_neighbors_in_range
[bool, default = False] `True`: if the real number of neighbors within the `neighbors_range` is greater than the `number_of_neighbors` parameter then take all of them anyway.

sk_mean
[float, default=None] The mean value of a process over a study area. Should be know before processing. That's why Simple Kriging has a limited number of applications. You must have multiple samples and well-known area to know this parameter.

allow_approx_solutions
[bool, default=False] Allows the approximation of kriging weights based on the OLS algorithm. We don't recommend set it to `True` if you don't know what are you doing. This parameter can be useful when you have clusters in your dataset, that can lead to singular or near-singular matrix creation.

Returns

: Tuple

Function returns tuple with:

- Mean Prediction Error,
- Mean Kriging Error: ratio of variance of prediction errors to the average variance error of kriging,
- array with: `[coordinate x, coordinate y, prediction error, kriging estimate error]`

References

1. Clark, I., (2004) "The Art of Cross Validation in Geostatistical Applications"
2. Clark I., (1979) "Does Geostatistics Work", Proc. 16th APCOM, pp.213.-225.

2.6 Development

2.6.1 Package structure

High level overview:

- **[x] `pyinterpolate`**
  - [x] `distance` - distance calculation,
  - [x] `idw` - inverse distance weighting interpolation,
  - [x] `io` - reads and prepares input spatial datasets,
  - [x] `kriging` - Ordinary Kriging, Simple Kriging, Poisson Kriging: centroid based, area-to-area, area-to-point,
  - [x] `pipelines` - a complex functions to smooth a block data, download sample data, compare different kriging techniques, and filter blocks,
  - [x] `processing` - core data structures of the package: `Blocks` and `PointSupport`, and additional functions used for internal processes,
  - [x] `variogram` - experimental variogram, theoretical variogram, variogram point cloud, semivariogram regularization & deconvolution,
  - [x] `viz` - interpolation of smooth surfaces from points into rasters.
- [x] `tutorials` - tutorials (Basic, Intermediate and Advanced).

2.6.2 Requirements and dependencies (v 0.3.0)

Core requirements and dependencies are:

- Python >= 3.7
- descartes
- geopandas
- matplotlib
- numpy

- tqdm
- pyproj
- scipy
- shapely
- fiona
- rtree
- prettytable
- pandas
- dask
- requests

You may check a specific version of requirements in the `setup.cfg` file.

2.6.3 Tests and contribution

All tests are grouped in the `test` directory. If you would like to contribute, then you won't avoid testing, but it is described step-by-step here: CONTRIBUTION.md

2.6.4 Development

- API documentation,
- Dedicated webpage,
- Check Issues and TODOs :)

2.6.5 Known Bugs

- Huge datasets (more than 10k points) lead to a memory error

2.7 Community

2.7.1 Contributors

Author(s)

1. **Szymon Moliński**, `@SimonMolinsky`

Maintainer(s)

1. **Szymon Moliński**, `@SimonMolinsky`

Contributors

1. **Lakshaya Inani**, `@Lakshayainani`
2. **Sean Lim**, `@seanjunheng2`
3. **Scott Gallacher**, `@scottgallacher-3`
4. **Ethem Turgut**, `@ethmtrgt`
5. **Tobiasz Wojnar**, `@TobiaszWojnar`

Reviewers (JOSS)

1. `@sdesabbata` (reviewer)
2. `@kenohori` (reviewer)
3. `@hugoledoux` (editor)

2.7.2 Network

Join our community in Discord: Discord Server Pyinterpolate

2.7.3 Use Cases

- Tick-Borne Disease Detector (Data Lions company) for the European Space Agency (2019-2020).
- B2C project related to the prediction of demand for specific flu medications (2020).
- B2G project related to the large-scale infrastructure maintenance (2020-2021).
- E-commerce service for reporting and analysis, building spatial / temporal profiles of customers (2022+).
- The external data augmentation for e-commerce services (2022+).

2.8 Learning Materials

2.8.1 Publications

- - (2022) URL (JOSS) Pyinterpolate: Spatial interpolation in Python for point measurements and aggregated datasets

2.8.2 Blog posts

- - (2022) URL (Ordinary Kriging) Interpolate Air Quality Measurements with Python,
- - (2022) URL (Area-to-Point Poisson Kriging) Get More from Crime Rate Data and Other Socio-Economic Indicators with Pyinterpolate,
- - (2022) URL (Theoretical Variograms) Geostatistics: Theoretical Variogram Models
- - (2022) URL (Kriging - General) Interpolate Air Pollution with Pyinterpolate

2.8.3 Presentations & Workshops

- - (2022) URL (Poisson Kriging) Wzbogacanie agregowanych danych publicznych z Pythonem i Geostatystyką

2.9 Bibliography

Pyinterpolate was created thanks to many resources and some of them are pointed here:

- Armstrong M., Basic Linear Geostatistics, Springer 1998,
- GIS Algorithms by Ningchuan Xiao: https://uk.sagepub.com/en-gb/eur/gis-algorithms/book241284
- Pardo-Iguzquiza E., VARFIT: a fortran-77 program for fitting variogram models by weighted least squares, Computers & Geosciences 25, 251-261, 1999,
- Goovaerts P., Kriging and Semivariogram Deconvolution in the Presence of Irregular Geographical Units, Mathematical Geology 40(1), 101-128, 2008
- Deutsch C.V., Correcting for Negative Weights in Ordinary Kriging, Computers & Geosciences Vol.22, No.7, pp. 765-773, 1996

CHAPTER

THREE

HOW TO CITE

Moliński, S., (2022). Pyinterpolate: Spatial interpolation in Python for point measurements and aggregated datasets. Journal of Open Source Software, 7(70), 2869, https://doi.org/10.21105/joss.02869