Spatial Data Analysis Using Python

Student Name: Tanay Singhal
Roll Number: MT21177

Project report submitted for the completion of the course
titled Spatial Statistics and Spatial Econometrics
on December 20, 2022

Project Advisor
Dr. Gaurav Arora

Indraprastha Institute of Information Technology
New Delhi

Report

December 20, 2022

0.1 Objective

Designing a **code book** for **analyzing spatial data** using python programming language, which can be used to analyze shapefiles that are most commonly done/analyzed using **ArcGis** software, which is a very heavy software and is **not open source**.

0.2 Major Contents

1. Extracting the Data and Density Plot (LAB 1 and LAB 2)
2. Overlay, Intersection and Projection (LAB 3)
3. Working with CSV data (LAB 4)
4. Working with Rastar Data (LAB 5)
5. Variogram and Krigging (LAB 6 and LAB 7)

0.3 Installing all dependencies for the project

[4]:
```
%matplotlib inline
import rtree
import pygeos
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import geopandas

pd.options.display.max_rows = 10
```

0.4 Data Used

[ ]:
```
import zipfile

zip_ref = zipfile.ZipFile("IND_adm.zip", 'r')
zip_ref.extractall()
zip_ref.close()
```

```
[ ]: zip_ref = zipfile.ZipFile("IND_rds.zip", 'r')
     zip_ref.extractall()
     zip_ref.close()
```

```
[ ]: from google.colab import drive
     drive.mount('/content/drive')
```

```
[ ]: zip_ref = zipfile.ZipFile("/content/drive/My Drive/RasterData.zip", 'r')
     zip_ref.extractall()
     zip_ref.close()
```

0.4.1 Importing geospatial data

```
[ ]: IND_districts = geopandas.read_file(r"IND_adm/IND_districts_from_DIVA-GIS.shp")
```

```
[ ]: IND_states = geopandas.read_file(r"IND_adm/IND_states_from_DIVA-GIS.shp")
     IND_national = geopandas.read_file(r"IND_adm/IND_national_from_DIVA-GIS.shp")
     IND_taluks = geopandas.read_file(r"IND_adm/IND_taluks_from_DIVA-GIS.shp")
```

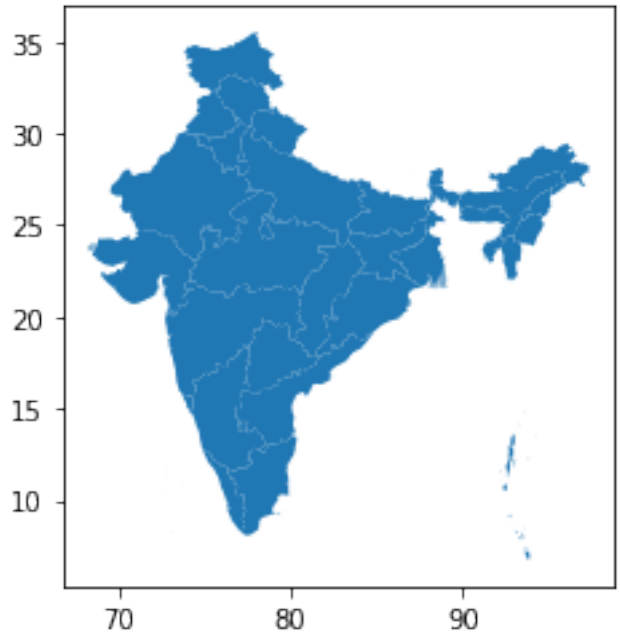
```
[ ]: IND_states.head()
```

```
[ ]:    ID_0  ISO NAME_0  ID_1              NAME_1 HASC_1  CCN_1 CCA_1  \
     0   105  IND  India     1  Andaman and Nicobar  IN.AN      0  None
     1   105  IND  India     2       Andhra Pradesh  IN.AP      0  None
     2   105  IND  India     3    Arunachal Pradesh  IN.AR      0  None
     3   105  IND  India     4                Assam  IN.AS      0  None
     4   105  IND  India     5                Bihar  IN.BR      0  None

              TYPE_1        ENGTYPE_1 NL_NAME_1  \
     0  Union Territor  Union Territory      None
     1           State            State      None
     2           State            State      None
     3           State            State      None
     4           State            State      None

                                              VARNAME_1  ID_NEW  LONGITUDE  \
     0  Andaman & Nicobar Islands|Andaman et Nicobar|I…     1.0  92.968178
     1                                             None     2.0  79.927139
     2  Agence de la Frontière du Nord-Est(French-obso…     3.0  94.676695
     3                                             None     4.0  92.829542
     4                                             None     5.0  85.604840

        LATITUDE                                           geometry
     0  11.225999  MULTIPOLYGON (((93.78773 6.85264, 93.78849 6.8…
     1  15.712608  MULTIPOLYGON (((80.19264 13.52070, 80.19264 13…
     2  28.039006  POLYGON ((96.15778 29.38310, 96.16380 29.37668…
```

```
3  26.357354  MULTIPOLYGON (((89.87145 25.53730, 89.87118 25…
4  25.679658  MULTIPOLYGON (((88.10548 26.53904, 88.10505 26…
```

[ ]:
```
IND_states.plot()
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7236de79a0>
```



What can we observe:

- Using `.head()` we can see the first rows of the dataset, just like we can do with Pandas.
- There is a 'geometry' column and the different statess are represented as polygons.
- We can use the `.plot()` method to quickly get a *basic* visualization of the data.

0.4.2 What's a GeoDataFrame?

We used the GeoPandas library to read in the geospatial data, and this returned us a `GeoDataFrame`:

[ ]:
```
type(IND_states)
```

[ ]:
```
geopandas.geodataframe.GeoDataFrame
```

A GeoDataFrame contains a tabular, geospatial dataset:

- It has a **'geometry' column** that holds the geometry information (or features in GeoJSON).
- The other columns are the **attributes** (or properties in GeoJSON) that describe each of the geometries

Such a `GeoDataFrame` is just like a pandas `DataFrame`, but with some additional functionality for working with geospatial data:

- A `.geometry` attribute that always returns the column with the geometry information (returning a GeoSeries). The column name itself does not necessarily need to be 'geometry', but it will always be accessible as the `.geometry` attribute.
- It has some extra methods for working with spatial data (area, distance, buffer, intersection, ...), which we will see in later notebooks

[ ]:
```
type(IND_states.geometry)
```

[ ]:
```
geopandas.geoseries.GeoSeries
```

0.4.3 Geometries: Points, Linestrings and Polygons

Spatial **vector** data can consist of different types, and the 3 fundamental types are:

- **Point** data: represents a single point in space.
- **Line** data ("LineString"): represents a sequence of points that form a line.
- **Polygon** data: represents a filled area.

Note: In the given Dataset we can see **MULTI** word occuring before the type of data. It rsimply refers to **group** i.e., **MULTIPOLYGON** refers to **group of POLYGONS**.

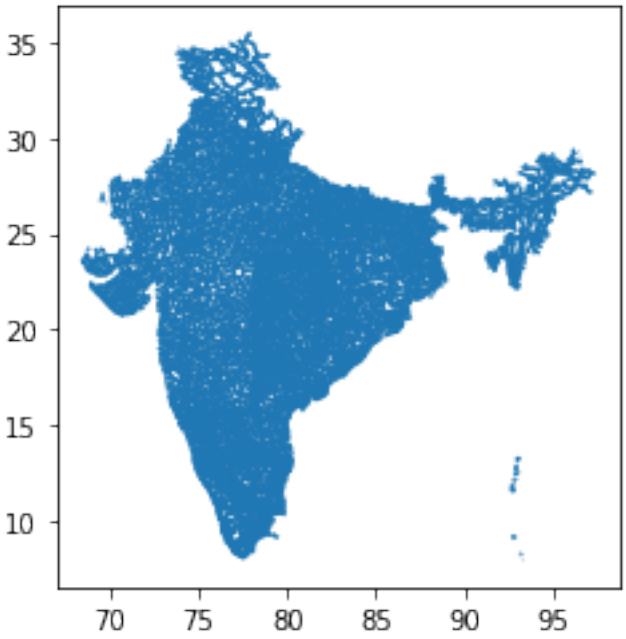
For the example we have seen up to now, the individual geometry objects are Polygons:

Let's import some other datasets with different types of geometry objects.

[ ]:
```
roads = geopandas.read_file(r"IND_rds/IND_roads.shp")
```

[ ]:
```
roads.plot()
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f723a11bfa0>
```



[ ]:
```
type(IND_UP.geometry[559])
```

[ ]:
```
shapely.geometry.polygon.Polygon
```

[ ]:
```
type(roads.geometry[0])
```

[ ]:
```
shapely.geometry.linestring.LineString
```

0.4.4 The shapely library

The individual geometry objects are provided by the `shapely` library

To construct one ourselves:

[ ]:
```
from shapely.geometry import Point, Polygon, LineString
```

[ ]:
```
p = Point(1, 1)
```

[ ]:
```
print(p)
```

```
POINT (1 1)
```

[ ]:
```
polygon = Polygon([(1, 1), (2,2), (2, 1)])
```

REMEMBER:

Single geometries are represented by `shapely` objects:

If you access a single geometry of a GeoDataFrame, you get a shapely geometry object

Those objects have similar functionality as geopandas objects (GeoDataFrame/GeoSeries). For example:

```
<li>`single_shapely_object.distance(other_point)` -> distance between two points</li>
<li>`geodataframe.distance(other_point)` ->  distance for each point in the geodataframe to the
```

0.4.5 Coordinate reference systems

A **coordinate reference system (CRS)** determines how the two-dimensional (planar) coordinates of the geometry objects should be related to actual places on the (non-planar) earth.

A GeoDataFrame or GeoSeries has a `.crs` attribute which holds (optionally) a description of the coordinate reference system of the geometries:

[ ]:
```
IND_UP.crs
```

[ ]:
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

For the `IND_UP` dataframe, it used WGS84 lon/lat reference system, which is one of the most used. It uses coordinates as latitude and longitude in degrees:

[ ]:
```
roads.crs
```

[ ]:
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

The `.crs` attribute is given as a dictionary.

Under the hood, GeoPandas uses the `pyproj` / `proj4` libraries to deal with the re-projections.

There are sometimes good reasons you want to change the coordinate references system of your dataset, for example:

- different sources with different crs -> need to convert to the same crs
- distance-based operations -> if you a crs that has meter units (not degrees)
- plotting in a certain crs (eg to preserve area)

We can convert a GeoDataFrame to another reference system using the `to_crs` function.

Note the different scale of x and y.

0.4.6 Constructing a GeoDataFrame manually

[ ]:
```
geopandas.GeoDataFrame({
    'geometry': [Point(1, 1), Point(2, 2)],
    'attribute1': [1, 2],
    'attribute2': [0.1, 0.2]})
```

[ ]:
```
                  geometry  attribute1  attribute2
0  POINT (1.00000 1.00000)           1         0.1
1  POINT (2.00000 2.00000)           2         0.2
```

0.4.7 Creating a GeoDataFrame from an existing dataframe

For example, if you have lat/lon coordinates in two columns:

[ ]:
```
df = pd.DataFrame(
    {'City': ['Buenos Aires', 'Brasilia', 'Santiago', 'Bogota', 'Caracas'],
     'Country': ['Argentina', 'Brazil', 'Chile', 'Colombia', 'Venezuela'],
     'Latitude': [-34.58, -15.78, -33.45, 4.60, 10.48],
     'Longitude': [-58.66, -47.91, -70.66, -74.08, -66.86]})
```

[ ]:
```
df['Coordinates']  = list(zip(df.Longitude, df.Latitude))
```

[ ]:
```
df['Coordinates'] = df['Coordinates'].apply(Point)
```

[ ]:
```
gdf = geopandas.GeoDataFrame(df, geometry='Coordinates')
```

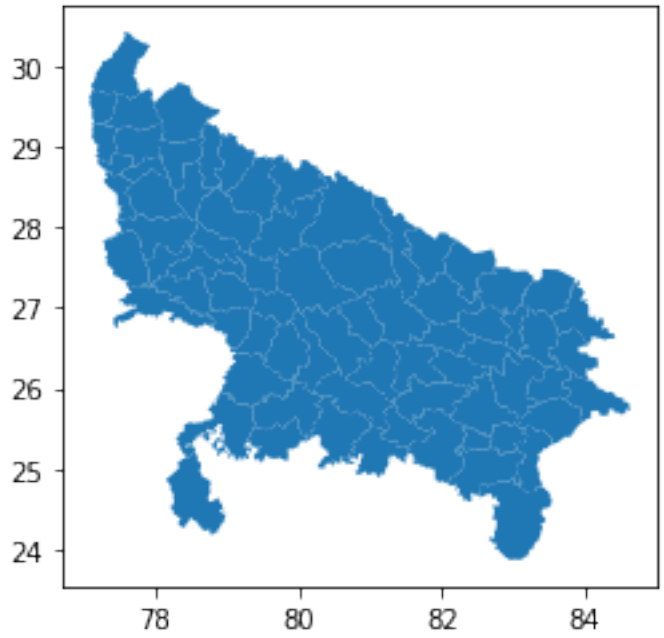
0.5 Extracting Data (Part of LAB 1 and LAB 2)

We can use boolean filtering to select a subset of the dataframe based on a condition:

[ ]:
```
IND_UP = IND_districts[IND_districts["NAME_1"] == "Uttar Pradesh"]
```

[ ]:
```
IND_UP.plot()
```

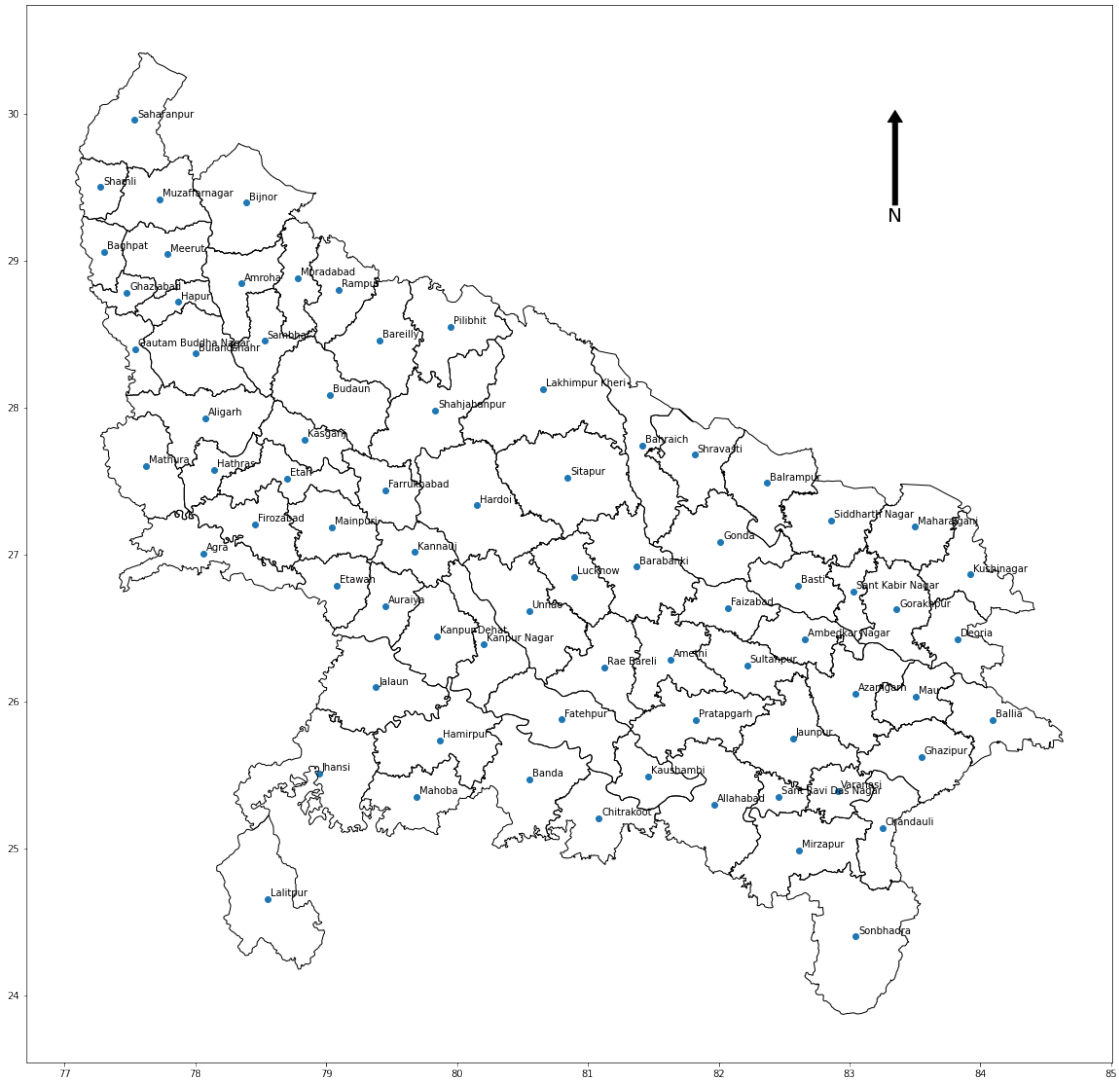
[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f72399c2e20>



[ ]:
```
UP_2 = IND_UP.copy()
UP_2["geometry"] = UP_2["geometry"].centroid
ax = UP_2.plot(figsize=(20, 20))
# IND_UP.plot(edgecolor='k', facecolor='none', figsize=(15, 10))
# gdf_inter.plot(ax=ax)

for x, y, label in zip(UP_2.geometry.x, UP_2.geometry.y, UP_2.NAME_2):
    ax.annotate(label, xy=(x, y), xytext=(3, 3), textcoords="offset points")

IND_UP.plot(ax = ax,edgecolor='k', facecolor='none')
x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)
```

```
<ipython-input-26-46f177a41e8c>:2: UserWarning: Geometry is in a geographic CRS.
Results from 'centroid' are likely incorrect. Use 'GeoSeries.to_crs()' to re-
project geometries to a projected CRS before this operation.

  UP_2["geometry"] = UP_2["geometry"].centroid
```

[ ]: Text(0.8, 0.8, 'N')



0.5.1 Density Plot (Part of LAB 1 and 2)

[ ]:
```
IND_UP
```

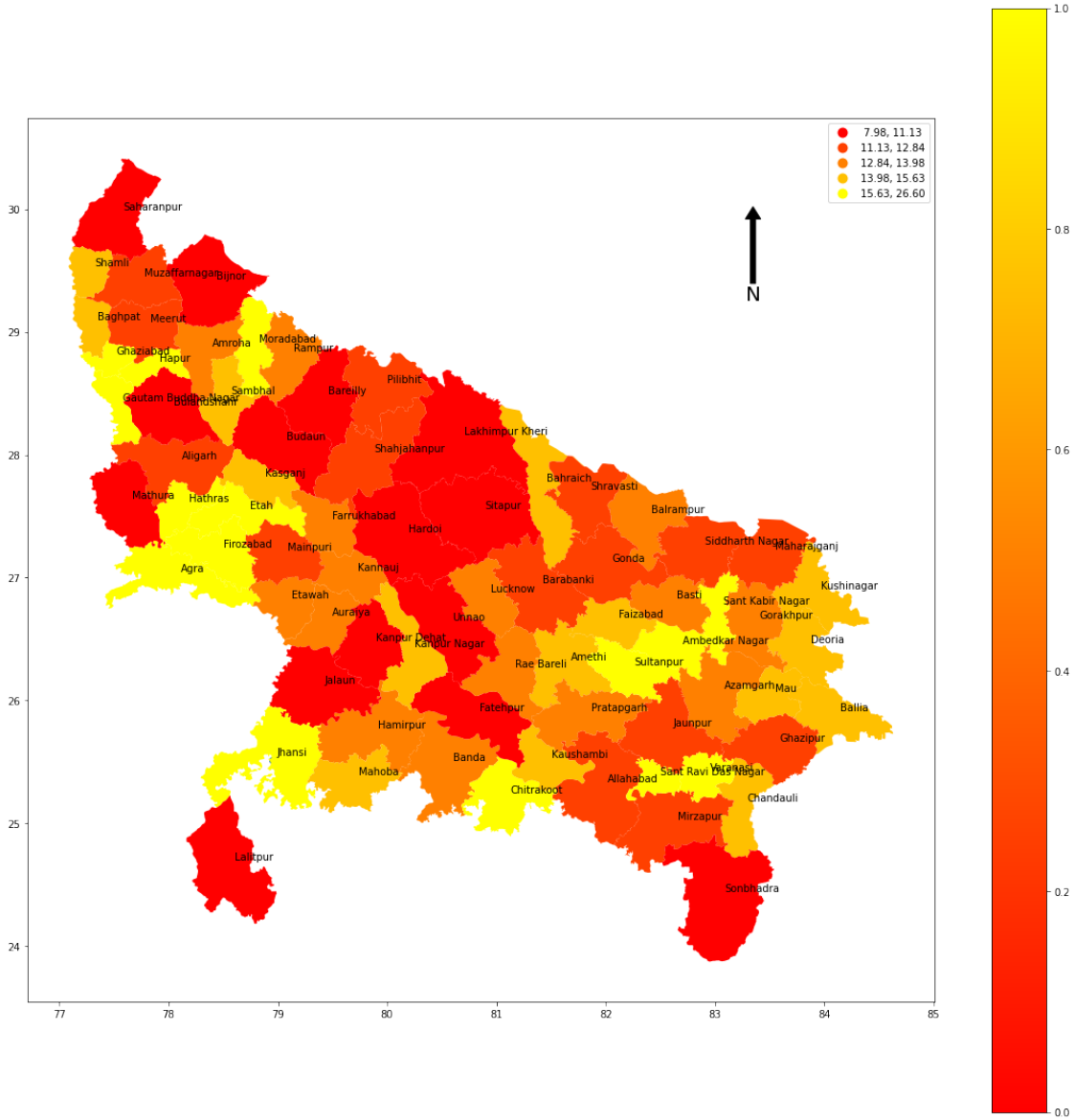
[ ]:
```
IND_UP['Perimeter'] = IND_UP.length
```

[ ]:
```
IND_UP['Area'] = IND_UP.area
```

[ ]:
```
IND_UP['Density'] = IND_UP['Perimeter'] / IND_UP['Area']
```

[ ]:
```
import matplotlib.pyplot as plt
```

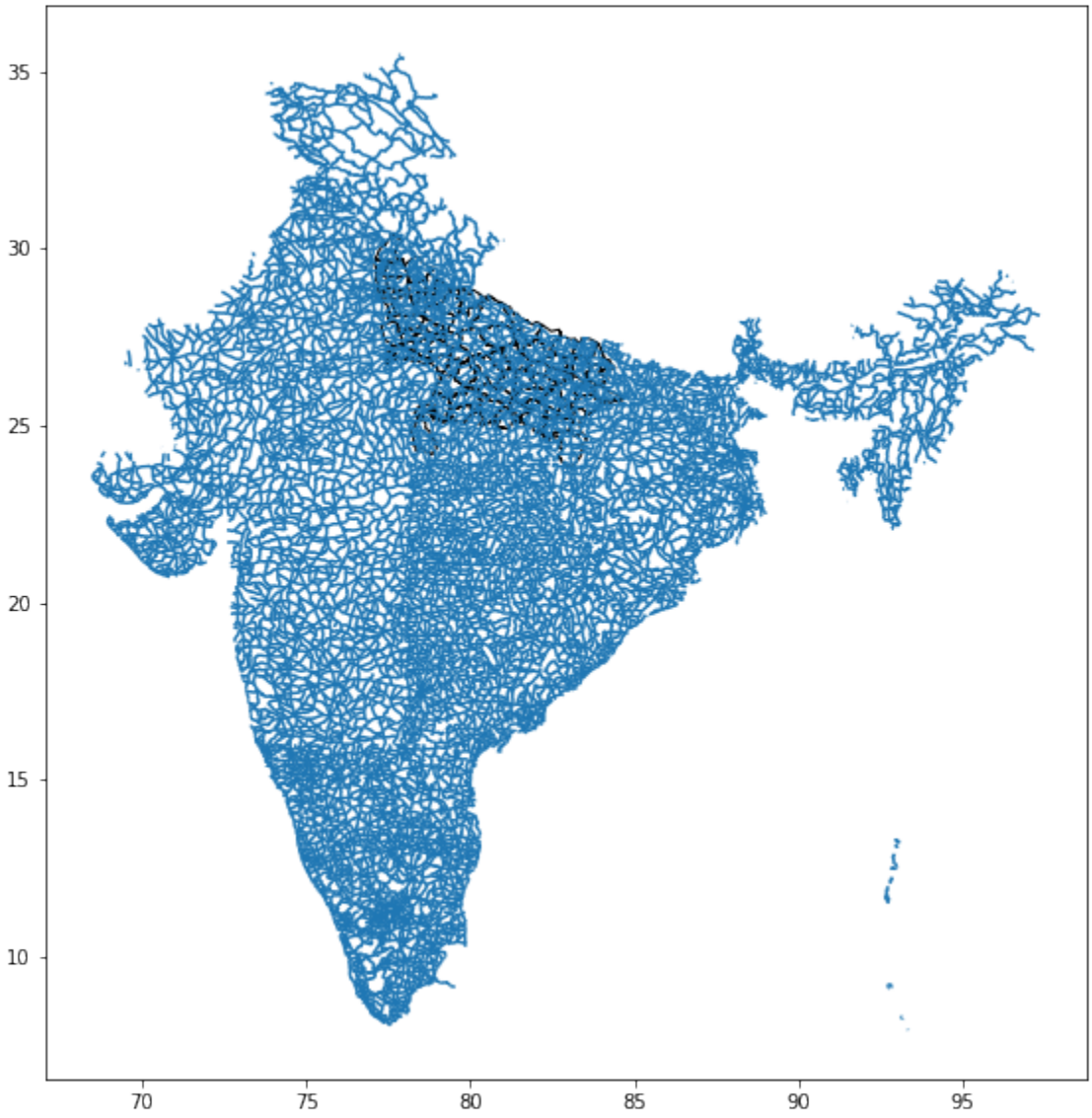
[ ]:
```
# fig, ax = plt.subplots(figsize=(12,10), subplot_kw=dict(aspect='equal'))
ax = UP_2.plot(figsize=(20, 20))

for x, y, label in zip(UP_2.geometry.x, UP_2.geometry.y, UP_2.NAME_2):
    ax.annotate(label, xy=(x, y), xytext=(5, 5), textcoords="offset points")
x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)
IND_UP.plot(column='Density', scheme='Quantiles', label = IND_UP['NAME_2'],
        k=5, cmap='autumn', legend=True, ax=ax)

sm = plt.cm.ScalarMappable(cmap='autumn')
plt.colorbar(sm, ax = ax)
plt.show()
```



0.6 Plotting our different layers together (Part of LAB 3)

[ ]:
```
ax = IND_UP.plot(edgecolor='k', facecolor='none', figsize=(15, 10))
roads.plot(ax=ax)
# cities.plot(ax=ax, color='red')
# # ax.set(xlim=(-20, 60), ylim=(-40, 40))
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f723424a5b0>
```



[ ]:
```
geopandas.__version__
```

[ ]:
```
pygeos.__version__
```

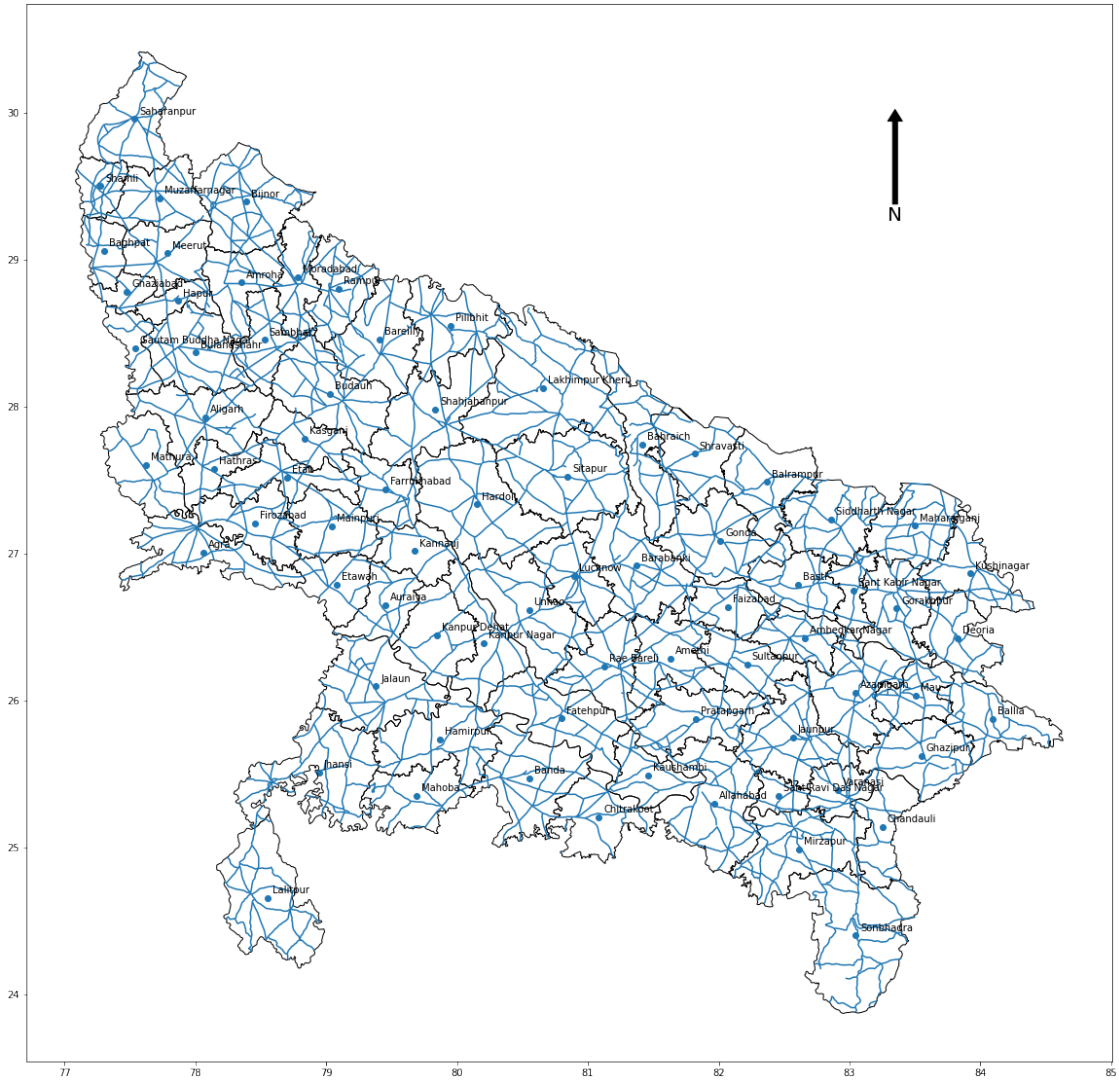
[ ]:
```
geopandas.options.use_pygeos = True
```

[ ]:
```
gdf2 = geopandas.read_file(r"IND_rds/IND_roads.shp").to_crs(IND_UP.crs)
gdf_inter = geopandas.overlay(IND_UP, gdf2,keep_geom_type=False,␣
↪how='intersection')
```

[ ]:
```
# ax = IND_UP.plot(edgecolor='k', facecolor='none', figsize=(20, 20))
ax = UP_2.plot(figsize=(20, 20))
```

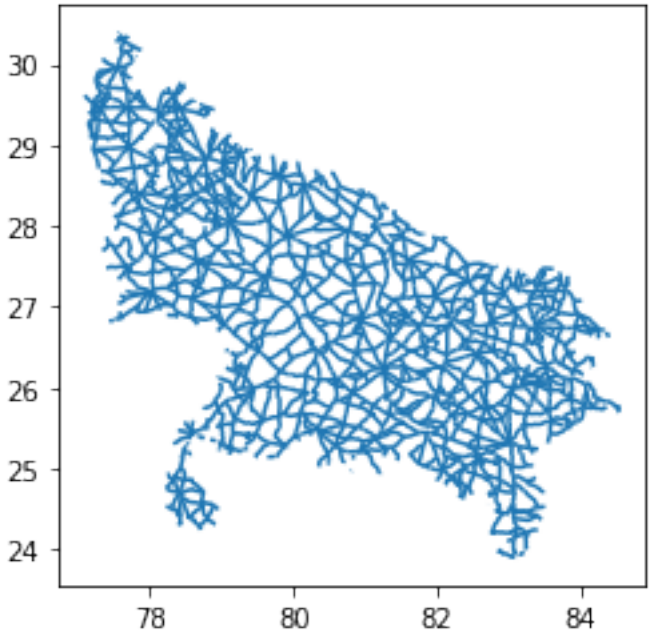
```
for x, y, label in zip(UP_2.geometry.x, UP_2.geometry.y, UP_2.NAME_2):
    ax.annotate(label, xy=(x, y), xytext=(5, 5), textcoords="offset points")
IND_UP.plot(ax=ax, edgecolor='k', facecolor='none')
gdf_inter.plot(ax=ax)
x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)
```

[ ]: Text(0.8, 0.8, 'N')



[ ]:
```
gdf_inter.plot()
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f72334b2d30>



1 Working With CSV data (LAB 4)

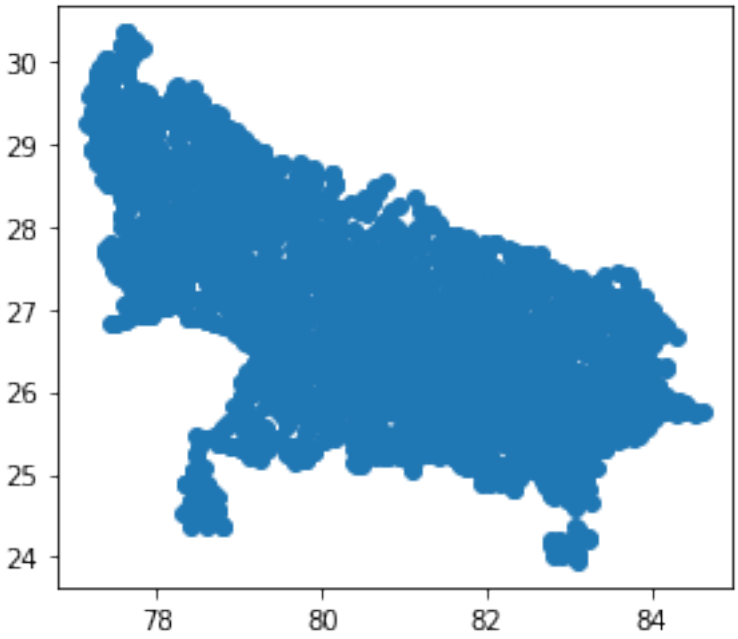
[ ]:
```
import pandas as pd
```

[ ]:
```
df = pd.read_csv('Final_Data_excel.csv')
```

[ ]:
```
gdf = geopandas.GeoDataFrame(
    df, geometry=geopandas.points_from_xy(df.Longtitude, df.Latitude_1)
)
```

[ ]:
```
gdf.plot()
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f72342329a0>



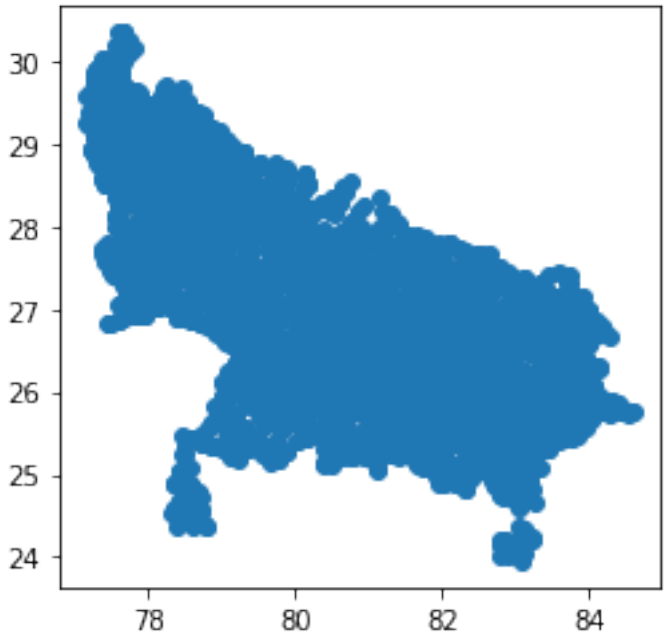
[ ]:
```
gdf_2 = gdf.set_crs("EPSG:4326")
```

[ ]:
```
gdf_2.crs
```

[ ]:
```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

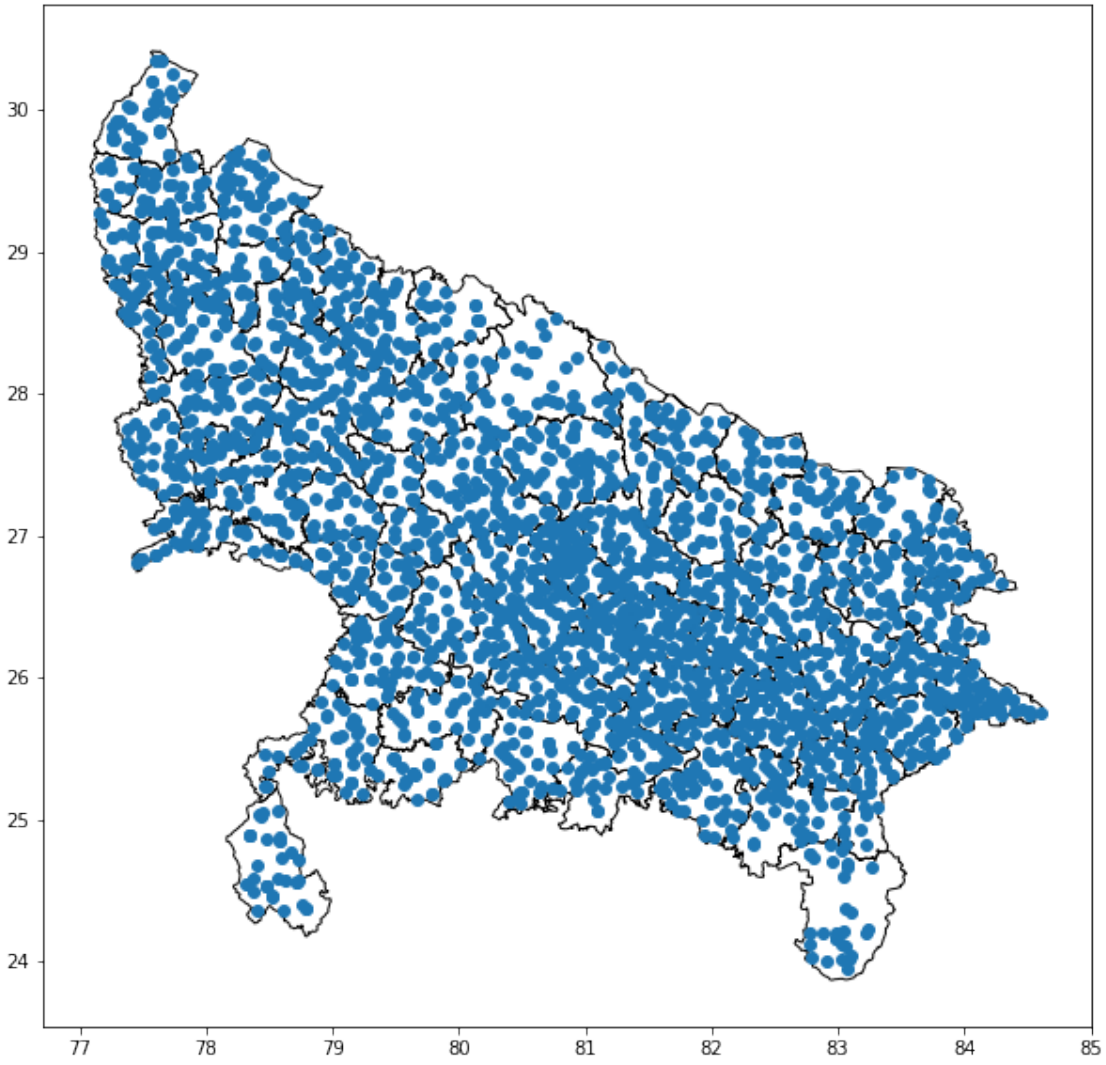
[ ]:
```
gdf_2.plot()
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7224798790>
```



[ ]:
```
ax = IND_UP.plot(edgecolor='k', facecolor='none', figsize=(15, 10))
gdf_2.plot(ax=ax)
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f72247c5100>

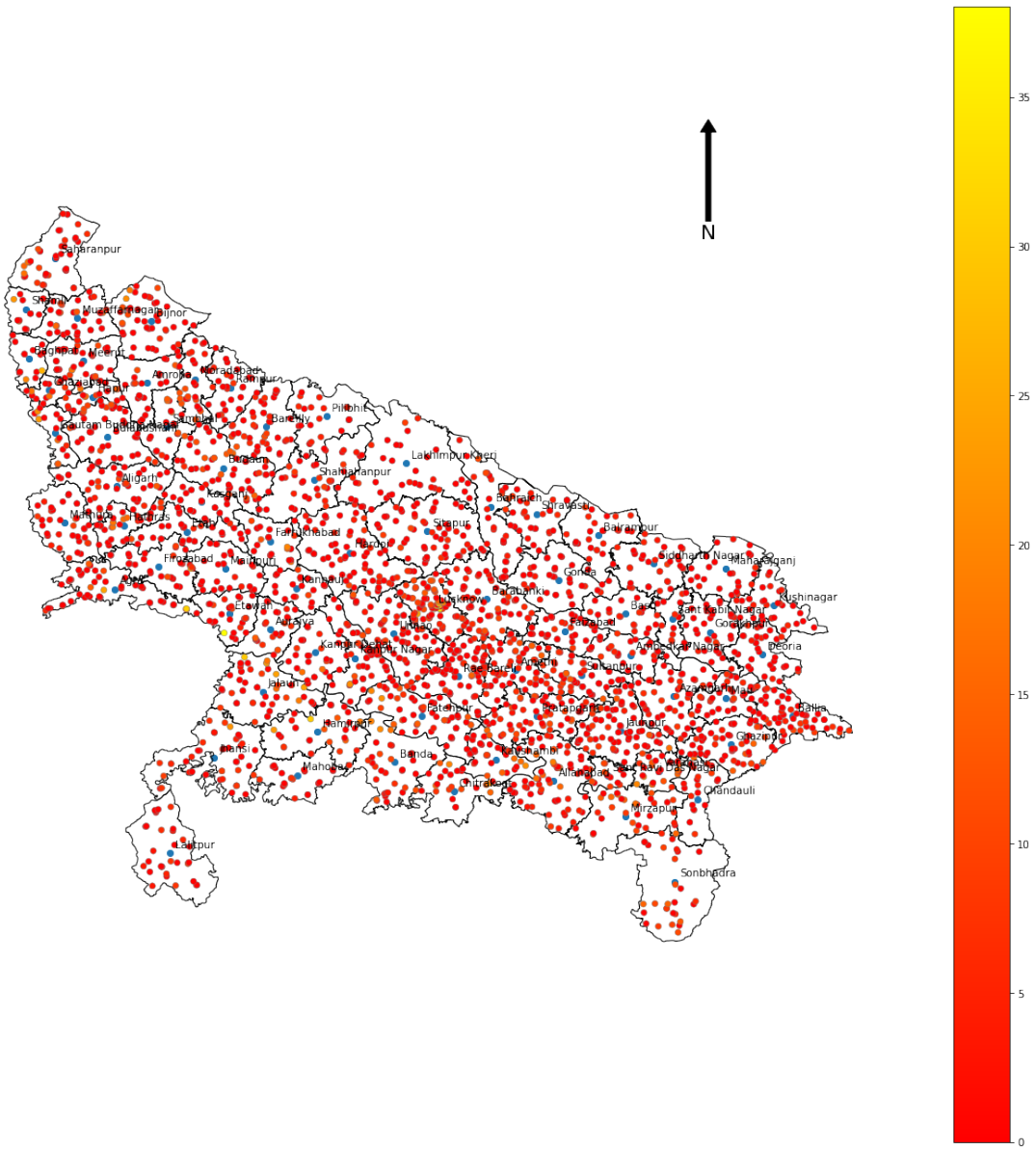


[ ]:
```
gdf_point_inter = geopandas.overlay(IND_UP, gdf_2,keep_geom_type=False,␣
↪how='intersection')
```

[ ]:
```
gdf_point_union = geopandas.overlay(IND_UP, gdf_2,keep_geom_type=False,␣
↪how='union')
```

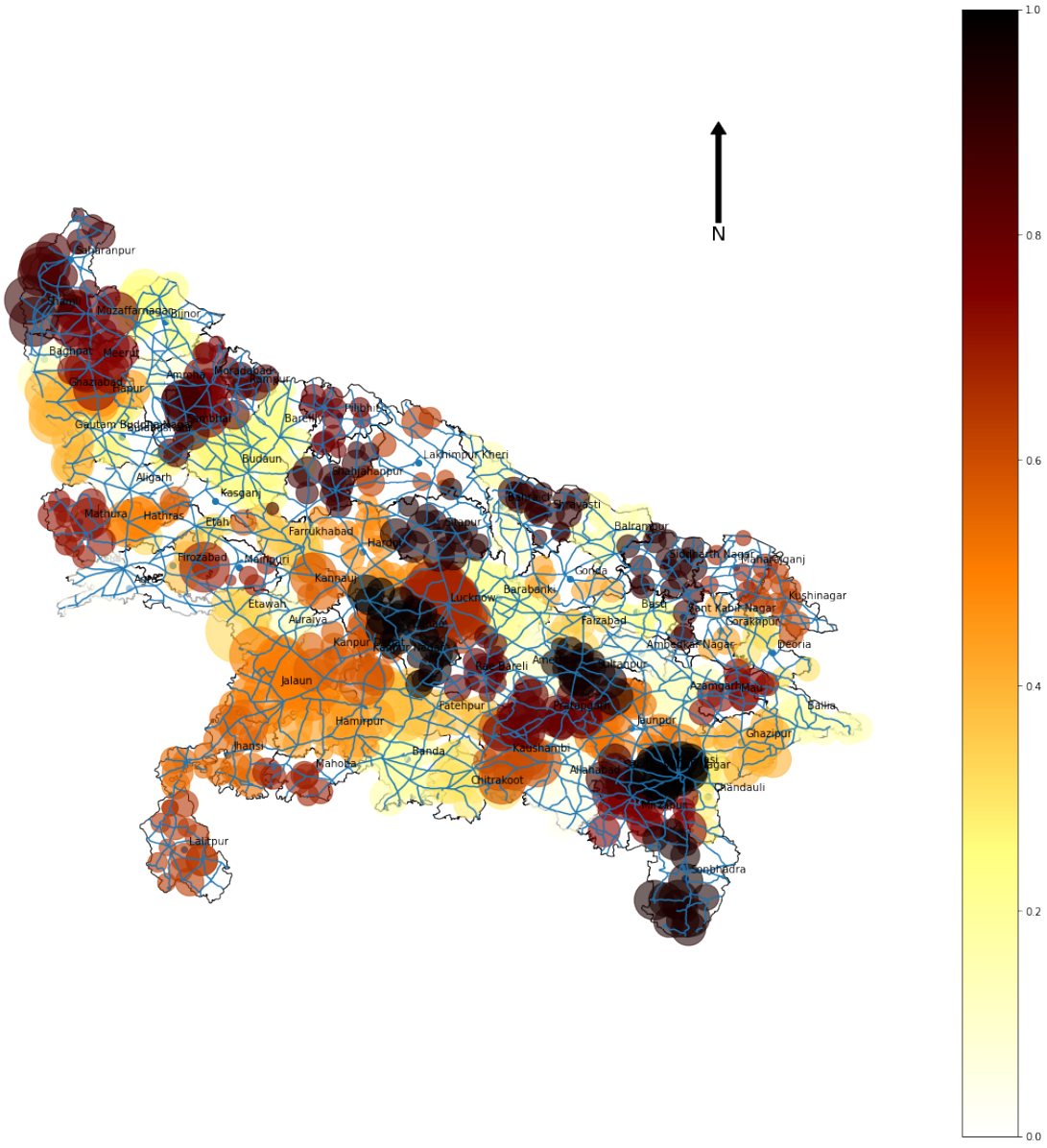
[ ]:
```
ax = UP_2.plot(figsize=(20, 20))

for x, y, label in zip(UP_2.geometry.x, UP_2.geometry.y, UP_2.NAME_2):
    ax.annotate(label, xy=(x, y), xytext=(5, 5), textcoords="offset points")
IND_UP.plot(ax=ax, edgecolor='k', facecolor='none')

x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
```

```
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)

gdf_point_inter.plot(ax=ax, column="Grnd_lvl_2",␣
 ↪cmap="autumn",edgecolor="grey", linewidth=0.4, legend=True)
ax.axis("off")
plt.axis('equal')
plt.show()
```



1.0.1 Symbology Part

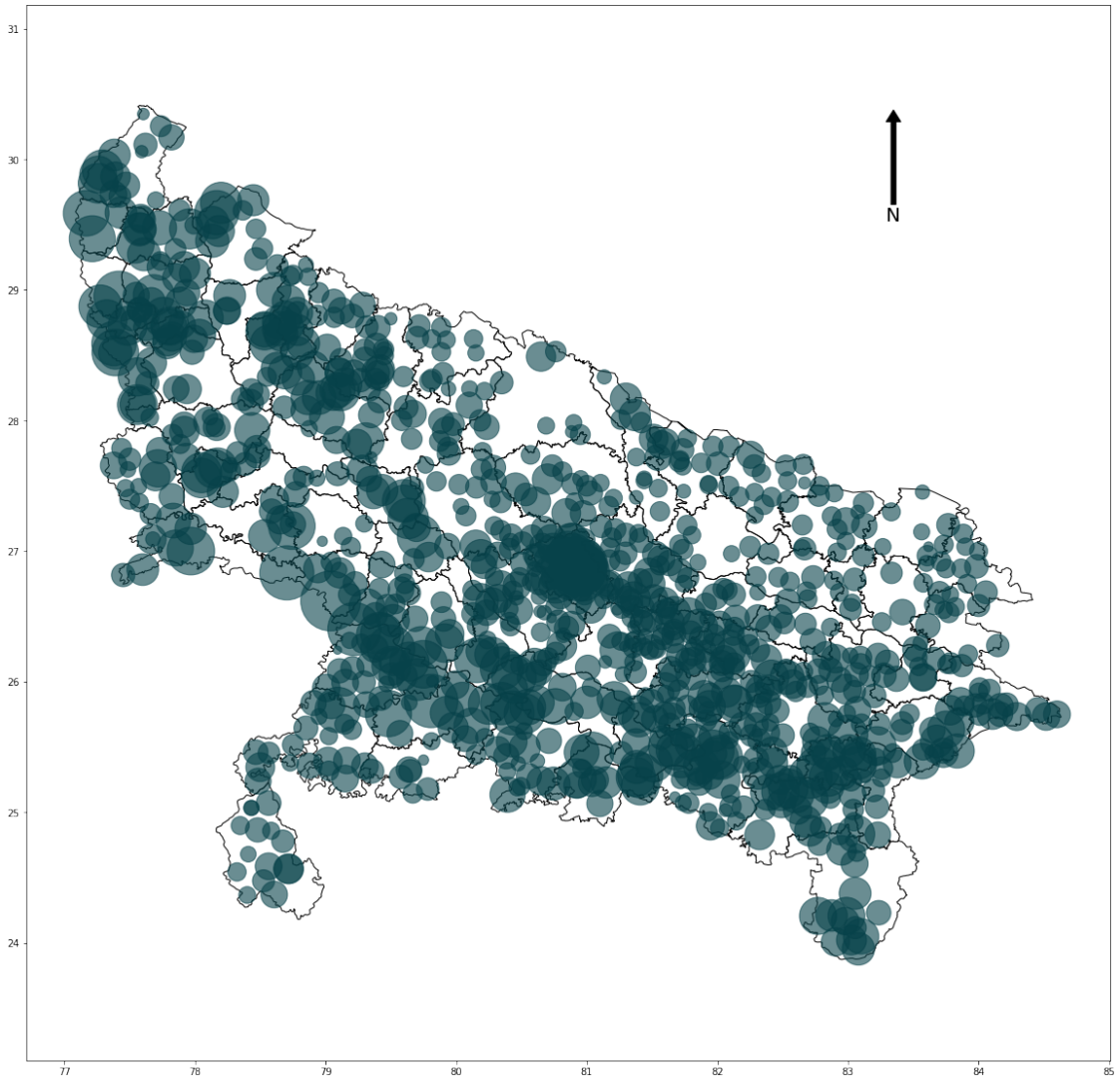
[ ]:
```
import matplotlib.pyplot as plt
import plotly_express as px

gdf_point_inter["grnd_2"] = gdf_point_inter["Grnd_lvl_2"] * 100

ax = UP_2.plot(figsize=(20, 20))

for x, y, label in zip(UP_2.geometry.x, UP_2.geometry.y, UP_2.NAME_2):
    ax.annotate(label, xy=(x, y), xytext=(5, 5), textcoords="offset points")
IND_UP.plot(ax=ax, edgecolor='k', facecolor='none')
gdf_point_inter.plot(ax=ax,cmap='afmhot_r', markersize="grnd_2",alpha=0.6,␣
 ↪categorical=False, legend=True )

gdf_inter.plot(ax=ax)
x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)


# creating ScalarMappable
sm = plt.cm.ScalarMappable(cmap='afmhot_r')


ax.axis("off")
plt.axis('equal')
plt.colorbar(sm, ax = ax)
plt.show()
```



[ ]:
```
gdf_point_inter_points = gdf_point_inter.copy()
# gdf_point_inter_points["geometry"] = gdf_point_inter_points["geometry"].
 ↪centroid
```

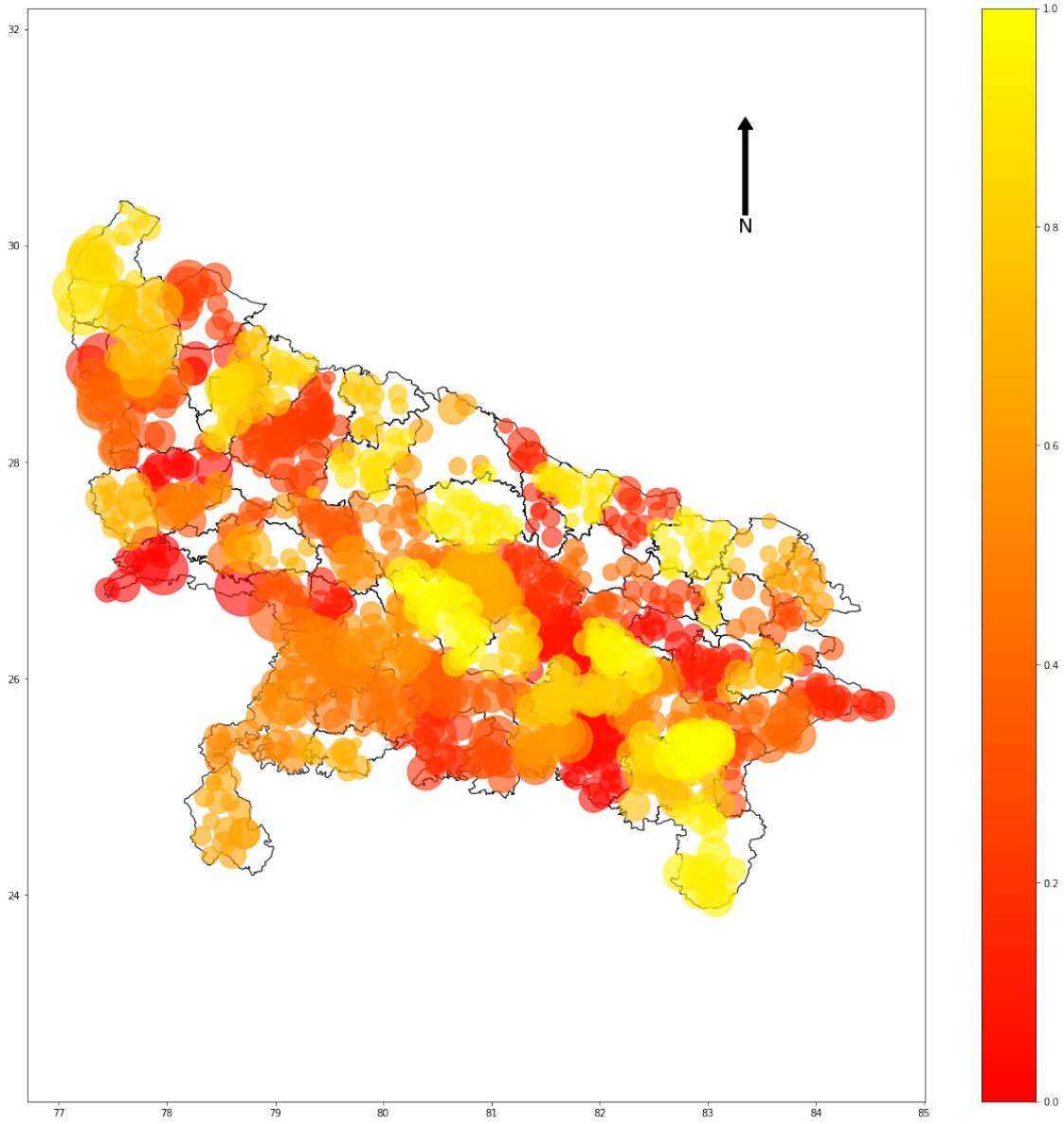
[ ]:
```
ax = IND_UP.plot(edgecolor='k', facecolor='none', figsize=(20, 20))
gdf_point_inter.plot(ax=ax,color='#07424A', markersize="grnd_2",alpha=0.6,␣
 ↪categorical=False, legend=True )

x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
```

```
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)

plt.axis('equal')
plt.show()
```



[ ]:
```
ax = IND_UP.plot(edgecolor='k', facecolor='none', figsize=(20, 20))
gdf_point_inter.plot(ax=ax,cmap='autumn', markersize="grnd_2",alpha=0.6,␣
 ↪categorical=False, legend=True )
# gdf_point_inter.plot(ax=ax, column="grnd_2", cmap="autumn", linewidth=0,␣
 ↪legend=True)
sm = plt.cm.ScalarMappable(cmap='autumn')
```

```
x, y, arrow_length = 0.8, 0.9, 0.1
ax.annotate('N', xy=(x, y), xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black', width=5, headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)

plt.axis('equal')
plt.colorbar(sm, ax = ax)
plt.show()
```

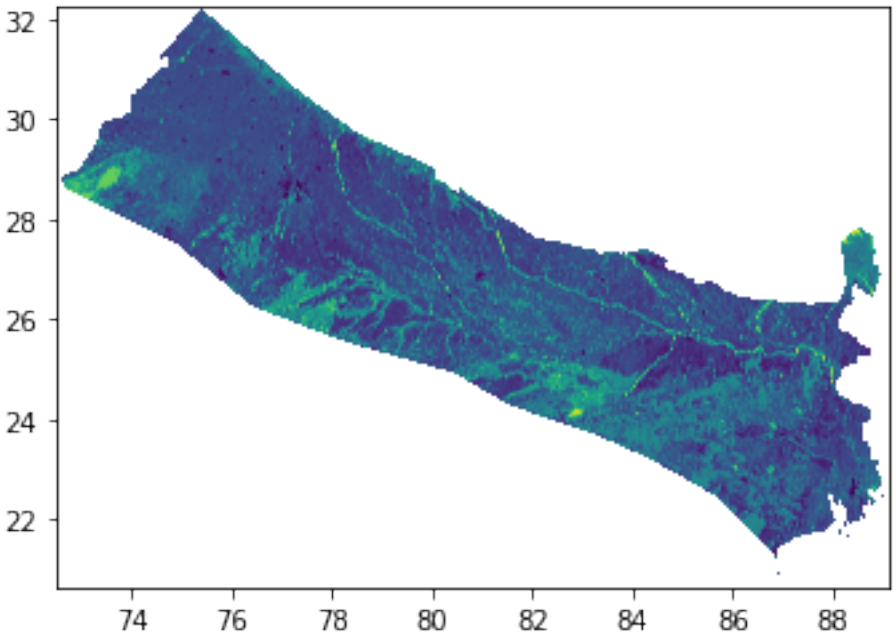


1.1 Working with Raster Data (LAB 5)

[ ]:
```
import rasterio

fp = r"lulc250k_0607_13861.tif"

# Open the file:
raster = rasterio.open(fp)

# Check type of the variable 'raster'
type(raster)
```

[ ]:
```
rasterio.io.DatasetReader
```

[ ]:
```
from rasterio.plot import show


raster = rasterio.open(fp)

show(raster)
```

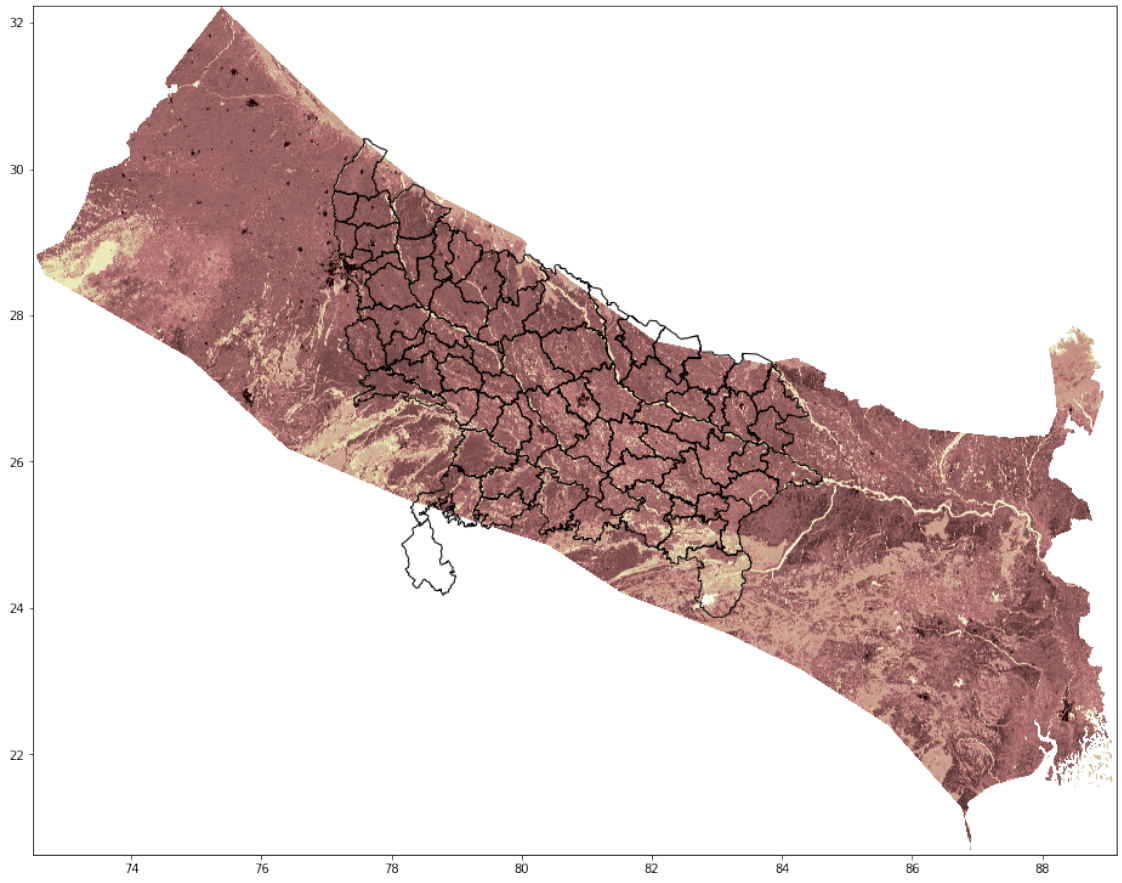


[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7223c0c6d0>
```

[ ]:
```
src = rasterio.open('lulc250k_0607_13861.tif')
```

[ ]:
```
from rasterio.plot import show

fig, ax = plt.subplots(1, figsize=(16,16))

# transform rasterio plot to real world coords
extent=[src.bounds[0], src.bounds[2], src.bounds[1], src.bounds[3]]
rasterio.plot.show(src, extent=extent, ax=ax, cmap='pink')

# gdf_point_inter.plot(ax=ax)
IND_UP.plot(edgecolor='k', facecolor='none',ax=ax)
```

[ ]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f7220b17a90>
```



1.2 Variogram and Krigging (LAB 6 and LAB 7)

Spatial Continuity **Spatial Continuity** is the correlation between values over distance.

- No spatial continuity – no correlation between values over distance, random values at each location in space regardless of separation distance.

- Homogenous phenomenon have perfect spatial continuity, since all values as the same (or very similar) they are correlated.

We need a statistic to quantify spatial continuity! A convenient method is the Semivariogram.

The Semivariogram Function of difference over distance.

- The expected (average) squared difference between values separated by a lag distance vector (distance and direction), h:

$$\gamma(\mathbf{h}) = \frac{\mathbf{1}}{\mathbf{2N(h)}} \sum_{\alpha=\mathbf{1}}^{\mathbf{N(h)}} (\mathbf{z}(\mathbf{u}_\alpha) - \mathbf{z}(\mathbf{u}_\alpha + \mathbf{h}))^{\mathbf{2}} \tag{1}$$

where $z(\mathbf{u}_\alpha)$ and $z(\mathbf{u}_\alpha + \mathbf{h})$ are the spatial sample values at tail and head locations of the lag vector respectively.

- Calculated over a suite of lag distances to obtain a continuous function.
- the $\frac{1}{2}$ term converts a variogram into a semivariogram, but in practice the term variogram is used instead of semivariogram.
- We prefer the semivariogram because it relates directly to the covariance function, $C_x(\mathbf{h})$ and univariate variance, σ_x^2:

$$C_x(\mathbf{h}) = \sigma_{\mathbf{x}}^{\mathbf{2}} - \gamma(\mathbf{h}) \tag{2}$$

Note the correlogram is related to the covariance function as:

$$\rho_x(\mathbf{h}) = \frac{\mathbf{C_x(h)}}{\sigma_{\mathbf{x}}^{\mathbf{2}}} \tag{3}$$

The correlogram provides of function of the $\mathbf{h} - \mathbf{h}$ scatter plot correlation vs. lag offset $\mathbf{h}$.

$$-1.0 \leq \rho_x(\mathbf{h}) \leq \mathbf{1.0} \tag{4}$$

Variogram Observations The following are common observations for variograms that should assist with their practical use.

Observation #1 - As distance increases, variability increase (in general). This is common since in general, over greater distance offsets, there is often more difference between the head and tail samples.

In some cases, such as with spatial cyclicity of the hole effect variogram model the variogram may have negative slope over somelag distance intervals

Negative slopes at lag distances greater than half the data extent are often caused by too few pairs for a reliable variogram calculation

Observation #2 - Calculated with over all possible pairs separated by lag vector, . We scan through the entire data set, searching for all possible pair combinations with all other data. We then calculate the variogram as one half the expectation of squared difference between all pairs.

More pairs results in a more reliable measure.

Observation #3 - Need to plot the sill to know the degree of correlation. Sill is the variance, σ_x^2

Given stationarity of the variance, σ_x^2, and variogram $\gamma(\mathbf{h})$:

we can define the covariance function:

$$C_x(\mathbf{h}) = \sigma_{\mathbf{x}}^{\mathbf{2}} - \gamma(\mathbf{h}) \tag{5}$$

The covariance measure is a measure of similarity over distance (the mirror image of the variogram as shown by the equation above).

Given a standardized distribution $\sigma_x^2 = 1.0$, the covariance, $C_x(\mathbf{h})$, is equal to the correlogram, $\rho_x(\mathbf{h})$:

$$\rho_x(\mathbf{h}) = \sigma_{\mathbf{x}}^{\mathbf{2}} - \gamma(\mathbf{h}) \tag{6}$$

Observation #4 - The lag distance at which the variogram reaches the sill is know as the range. At the range, knowing the data value at the tail location provides no information about a value at the head location of the lag distance vector.

Observation #5 - The nugget effect, a discontinuity at the origin Sometimes there is a discontinuity in the variogram at distances less than the minimum data spacing. This is known as **nugget effect**.

The ratio of nugget / sill, is known as relative nugget effect (%). Modeled as a discontinuity with no correlation structure that at lags, $h > \epsilon$, an infinitesimal lag distance, and perfect correlation at $\mathbf{h} = \mathbf{0}$. Caution when including nuggect effect in the variogram model as measurement error, mixing populations cause apparent nugget effect

This exercise demonstrates the semivariogram calculation with GeostatsPy. The steps include:

1. generate a 2D model with sequential Gaussian simulation
2. sample from the simulation
3. calculate and visualize experimental semivariograms

Objective In the PGE 383: Stochastic Subsurface Modeling class I want to provide hands-on experience with building subsurface modeling workflows. Python provides an excellent vehicle to accomplish this. I have coded a package called GeostatsPy with GSLIB: Geostatistical Library (Deutsch and Journel, 1998) functionality that provides basic building blocks for building subsurface modeling workflows.

The objective is to remove the hurdles of subsurface modeling workflow construction by providing building blocks and sufficient examples. This is not a coding class per se, but we need the ability to 'script' workflows working with numerical methods.

Load the required libraries The following code loads the required libraries.

[ ]:
```
import geostatspy.GSLIB as GSLIB                                    # GSLIB utilies,␣
↪visualization and wrapper
import geostatspy.geostats as geostats                              # GSLIB methods convert␣
↪to Python
```

We will also need some standard packages. These should have been installed with Anaconda 3.

Loading Tabular Data Here's the command to load our comma delimited data file in to a Pandas' DataFrame object.

[1]:
```
df_2 = pd.read_csv("Final_Data_excel.csv")                                  # read a .csv␣
↪file in as a DataFrame
```

[ ]:
```
df_2.describe().transpose()
```

Let's transform the data to normal score values (Gaussian distributed with a mean of 0.0 and variance of 1.0). This is required for sequential Gaussian simulation (common target for our variogram models) and the Gaussian transform assists with outliers and provides more interpretable variograms.

Let's look at the inputs for the GeostatsPy nscore program. Note the output include an ndarray with the transformed values (in the same order as the input data in Dataframe 'df' and column 'vcol'), and the transformation table in original values and also in normal score values.

[ ]:
```
geostats.nscore                                                     # see the input␣
↪parameters required by the nscore function
```

[ ]:
```
<function geostatspy.geostats.nscore(df, vcol, wcol=None, ismooth=False,
dfsmooth=None, smcol=0, smwcol=0)>
```

[ ]:
```
df_2['NGrnd_lvl_2'], tvGrnd, tnsGrnd = geostats.nscore(df_2, 'Grnd_lvl_2') #␣
↪nscore transform for all ground_level_data
```

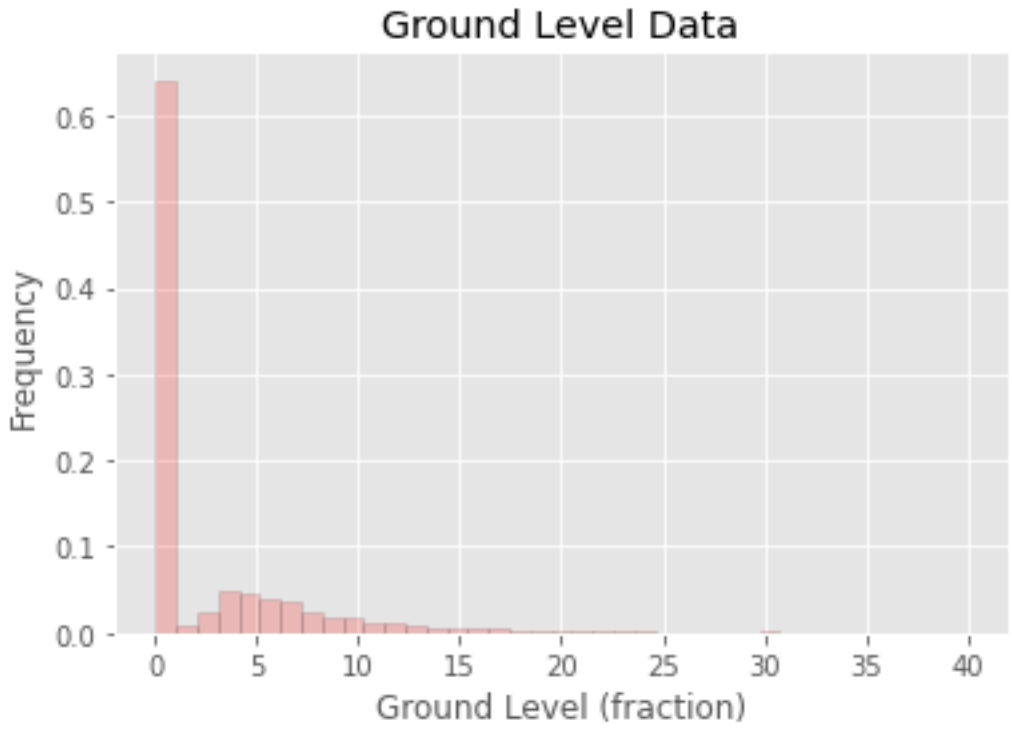
Let's look at a previous of one of the DataFrames to make sure that we now have the normal score Ground Level Data.

[ ]:
```
df_2.head()
```

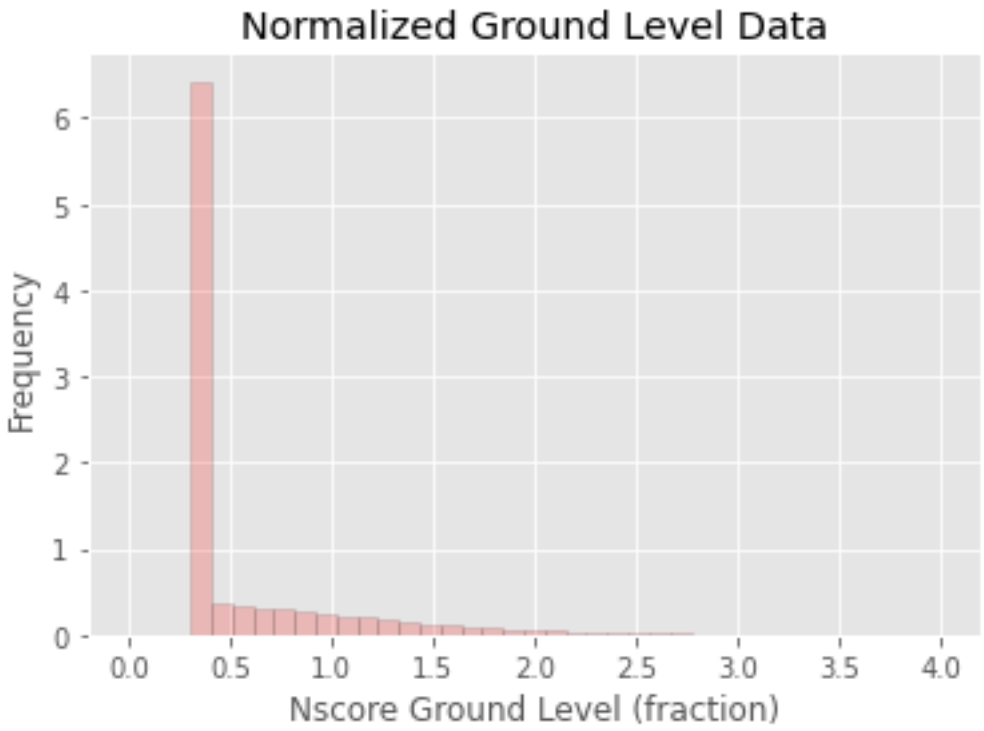
[ ]:
```
df_2.describe().transpose()
```

[ ]:
```
plt.hist(df_2['Grnd_lvl_2'], facecolor='red',bins=np.linspace(0.0,40.
↪0,40),alpha=0.2,density=True,edgecolor='black',label='Sand')
```

```
# plt.xlim([0.05,0.25]); plt.ylim([0,35.0])
plt.xlabel('Ground Level (fraction)'); plt.ylabel('Frequency'); plt.
 ↪title('Ground Level Data')
plt.grid(True)
```



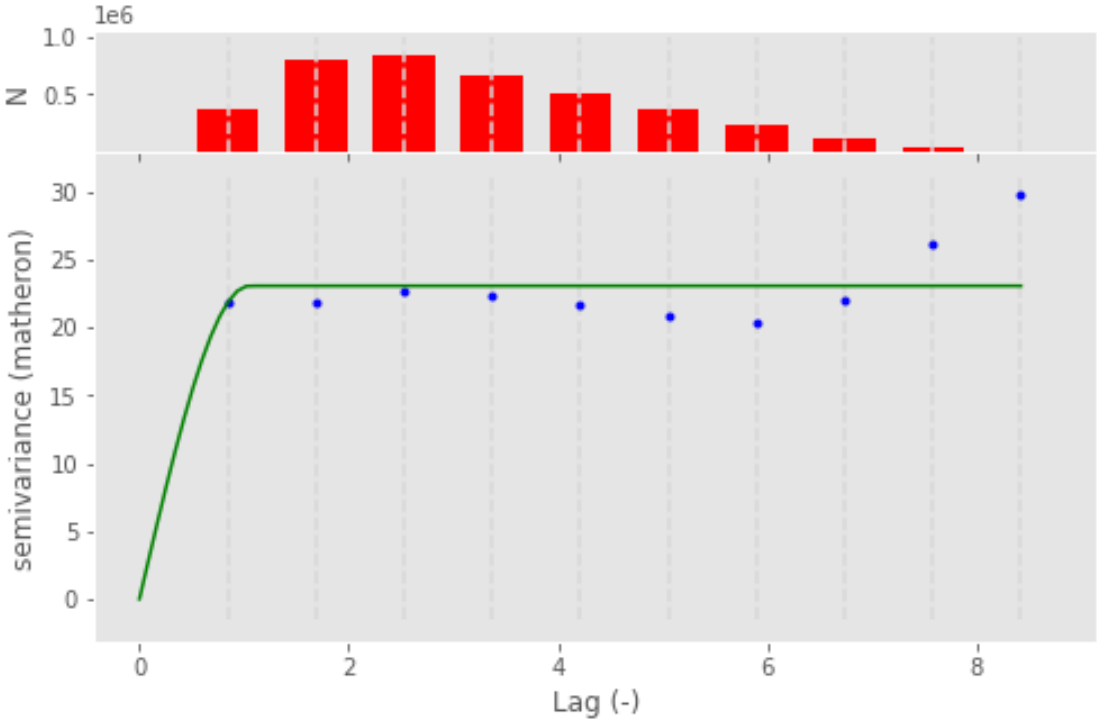
[ ]:
```
plt.hist(df_2['NGrnd_lvl_2'], facecolor='red',bins=np.linspace(0.0,4.
 ↪0,40),alpha=0.2,density=True,edgecolor='black',label='Sand')
# plt.xlim([0.05,0.25]); plt.ylim([0,35.0])
plt.xlabel('Nscore Ground Level (fraction)'); plt.ylabel('Frequency'); plt.
 ↪title('Normalized Ground Level Data')
plt.grid(True)
```

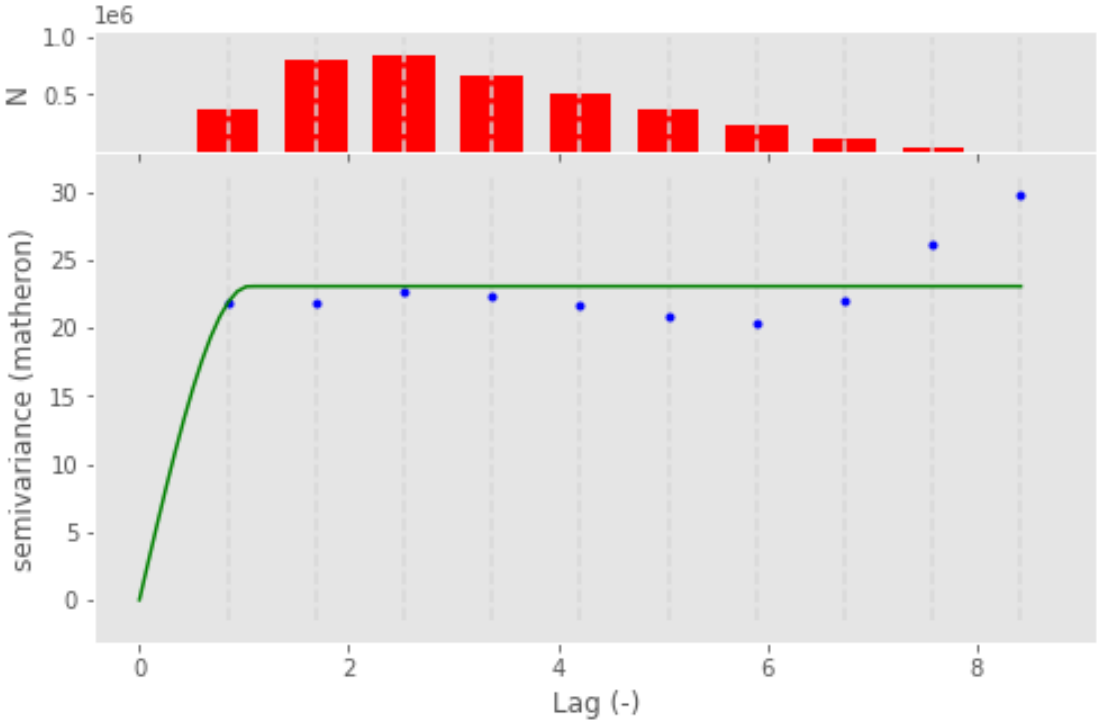


Let's see what the parameters are for the gamv, irregular data, experimental variogram calculation program.

[ ]:
```
import skgstat as skg

V = skg.Variogram(list(zip(df_2.Longtitude, df_2.Latitude_1)), df_2.Grnd_lvl_2.
 ↪values)

V.plot()
```

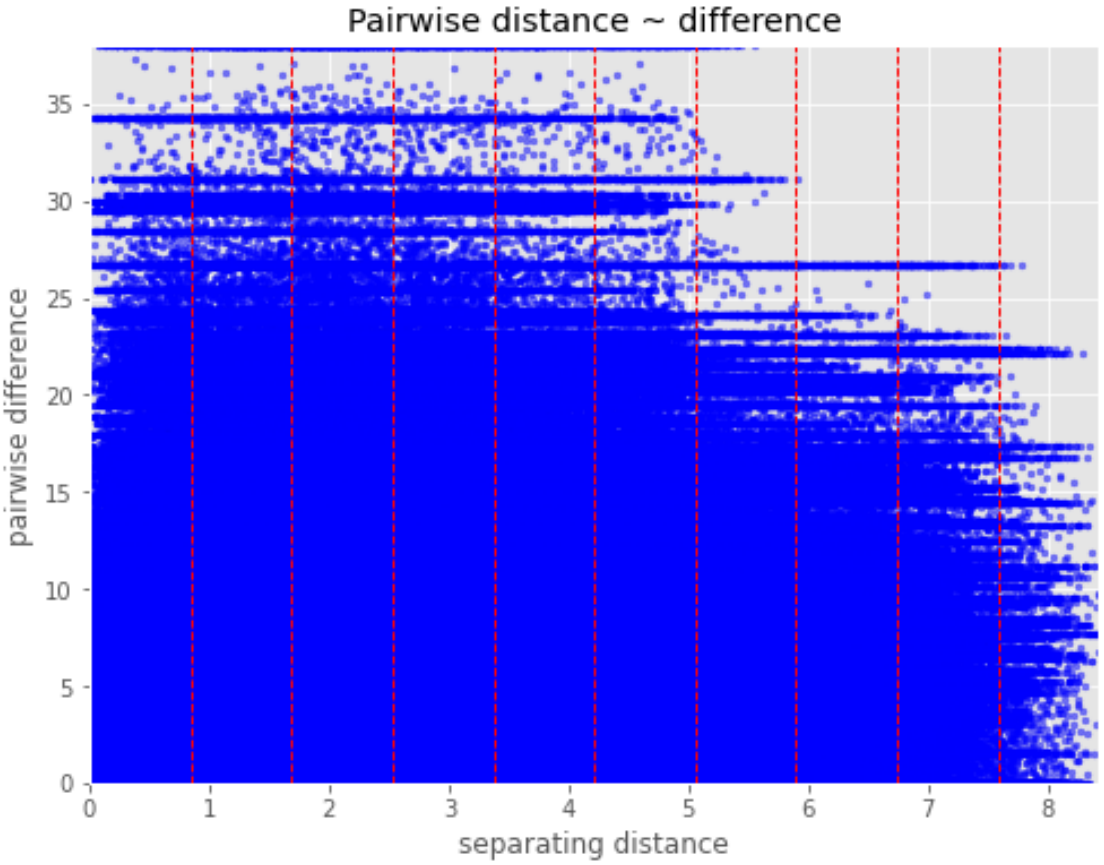
[ ]:

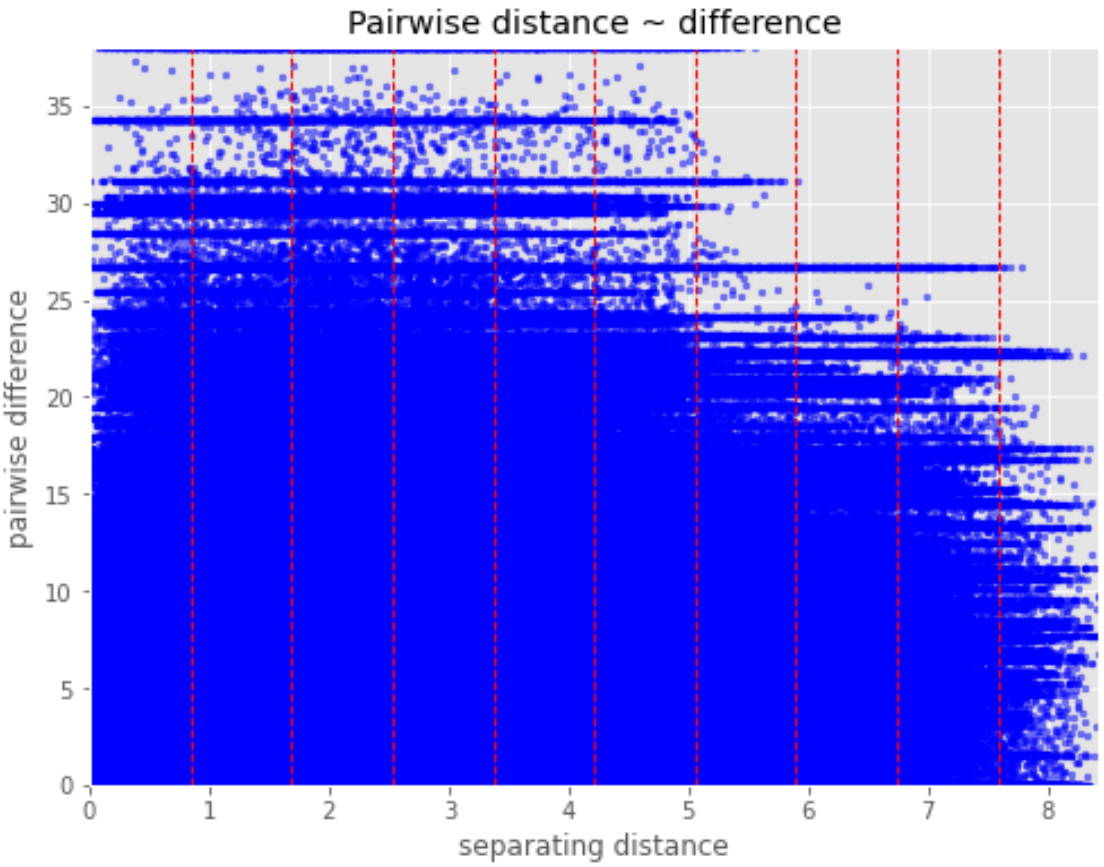

1e6
1.0
0.5
N
30
25
20
15
10
5
0
semivariance (matheron)
0
2
4
6
8
Lag (-)


1e6
1.0
0.5
N
30
25
20
15
10
5
0
semivariance (matheron)
0
2
4
6
8
Lag (-)

[ ]:
```
V.distance_difference_plot()
```

[ ]:





Experimental Variogran

[ ]:
```
df_3 = pd.read_csv("up_state_gwl_v6.csv")
```

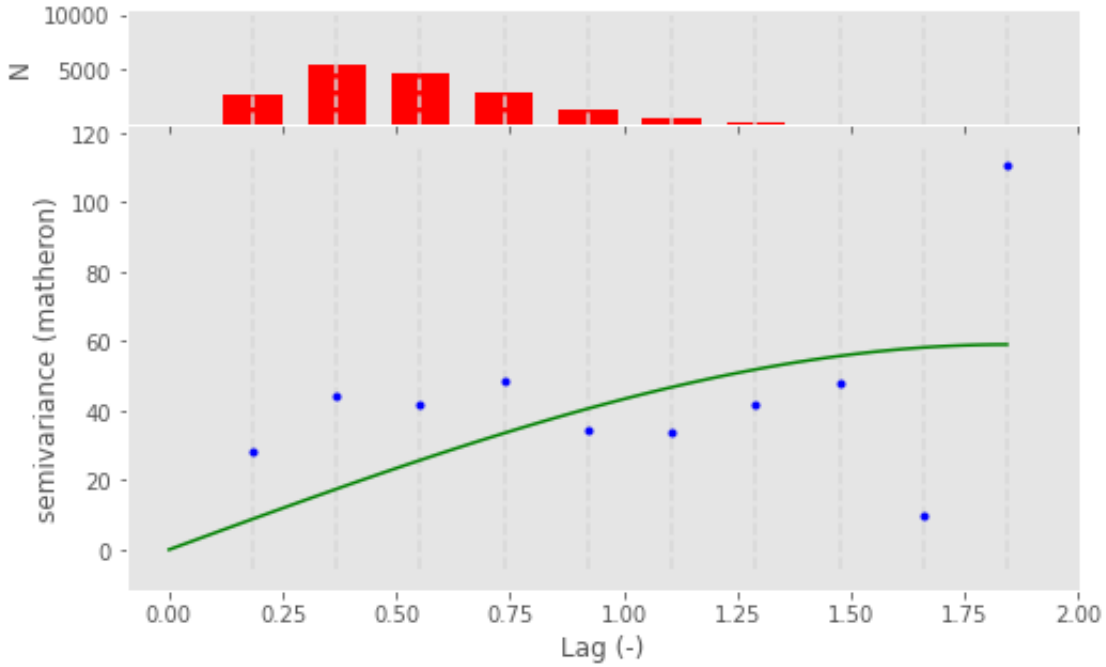
[ ]:
```
df_up = df_3[df_3["Year"] == 2015]
```

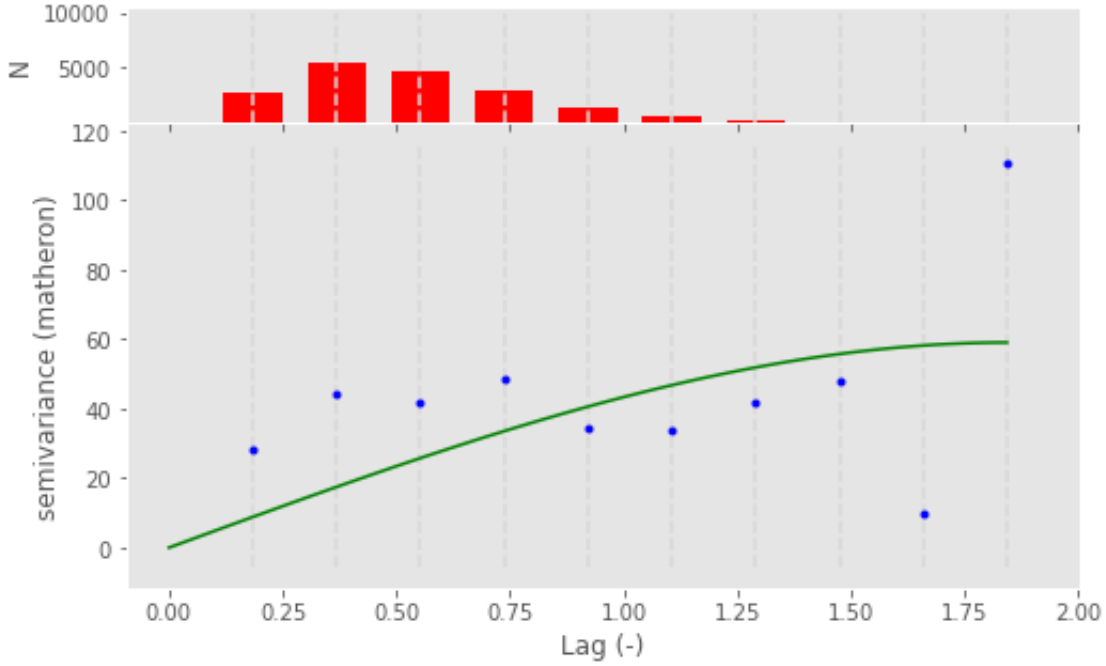
[ ]:
```
df_west_up = df_up[(df_up["District"] == "MUZAFFARNAGAR") | (df_up["District"]␣
↪== "GHAZIABAD") | (df_up["District"] == "BAGHPAT") | (df_up["District"] ==␣
↪"MEERUT") | (df_up["District"] == "HAPUR")]
```

[2]:
```
df_west_up = df_west_up.dropna()
df_west_up
```

[ ]:
```
V = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)), df_west_up.
↪PostMonsoon.values)

V.plot()
```

[ ]:





Variogram Models

Spherical Models

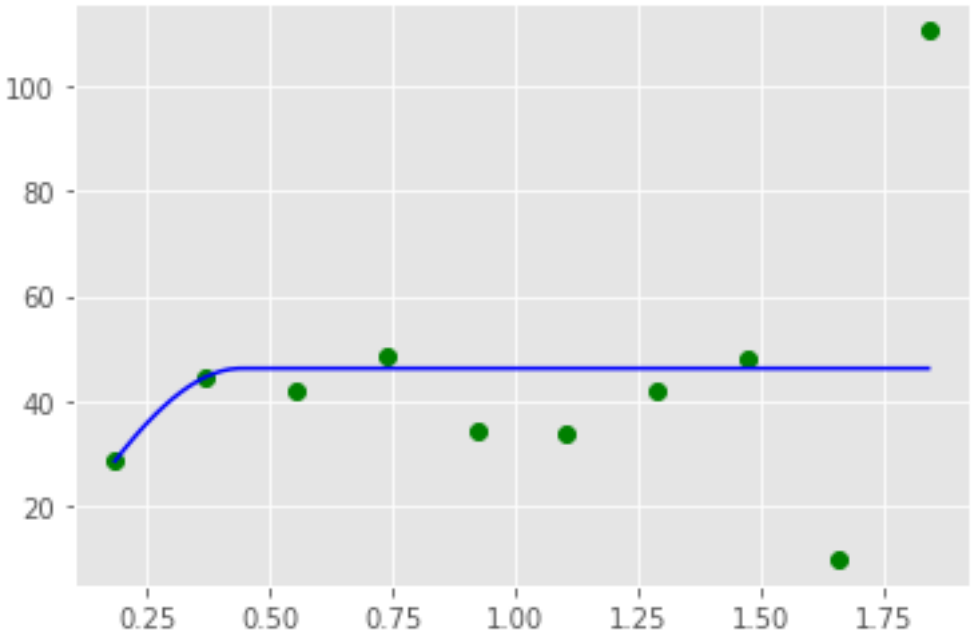
[ ]:
```
from skgstat import models

# set estimator back
V.estimator = 'matheron'

V.model = 'spherical'

xdata = V.bins

ydata = V.experimental

from scipy.optimize import curve_fit

# initial guess - otherwise lm will not find a range
p0 = [np.mean(xdata), np.mean(ydata), 0]

cof, cov = curve_fit(models.spherical, xdata, ydata, p0=p0)
```

[ ]:
```
print("range: %.2f   sill: %.f   nugget: %.2f" % (cof[0], cof[1], cof[2]))
```

```
range: 0.44   sill: 43   nugget: 3.14
```

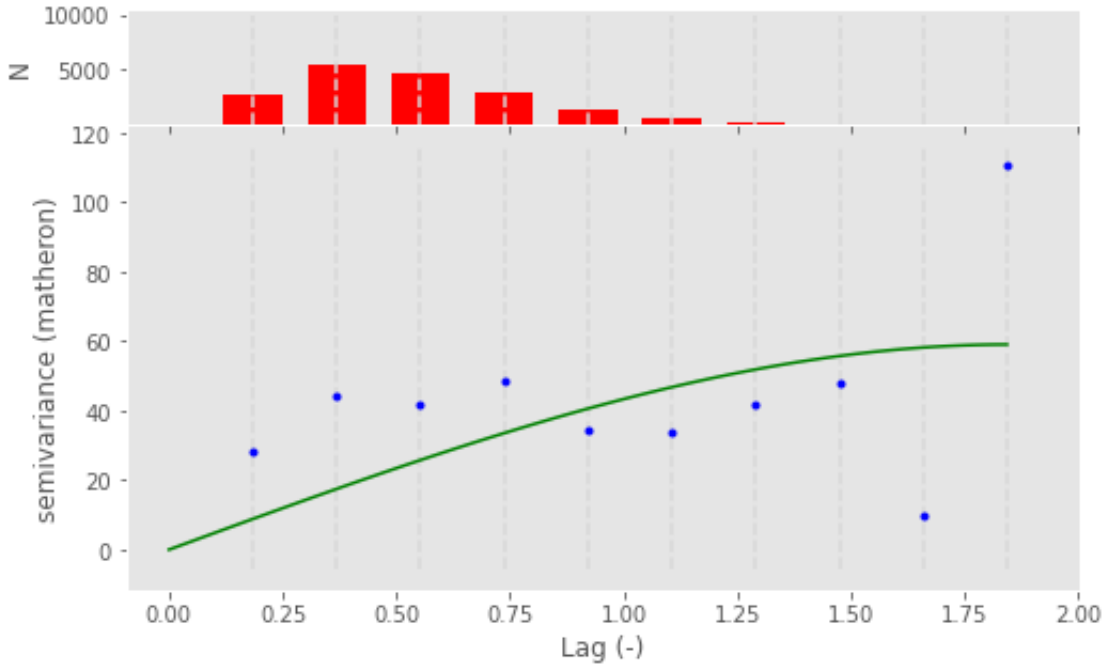
[ ]:
```
xi = np.linspace(xdata[0], xdata[-1], 100)

yi = [models.spherical(h, *cof) for h in xi]

plt.plot(xdata, ydata, 'og')

plt.plot(xi, yi, '-b')
```

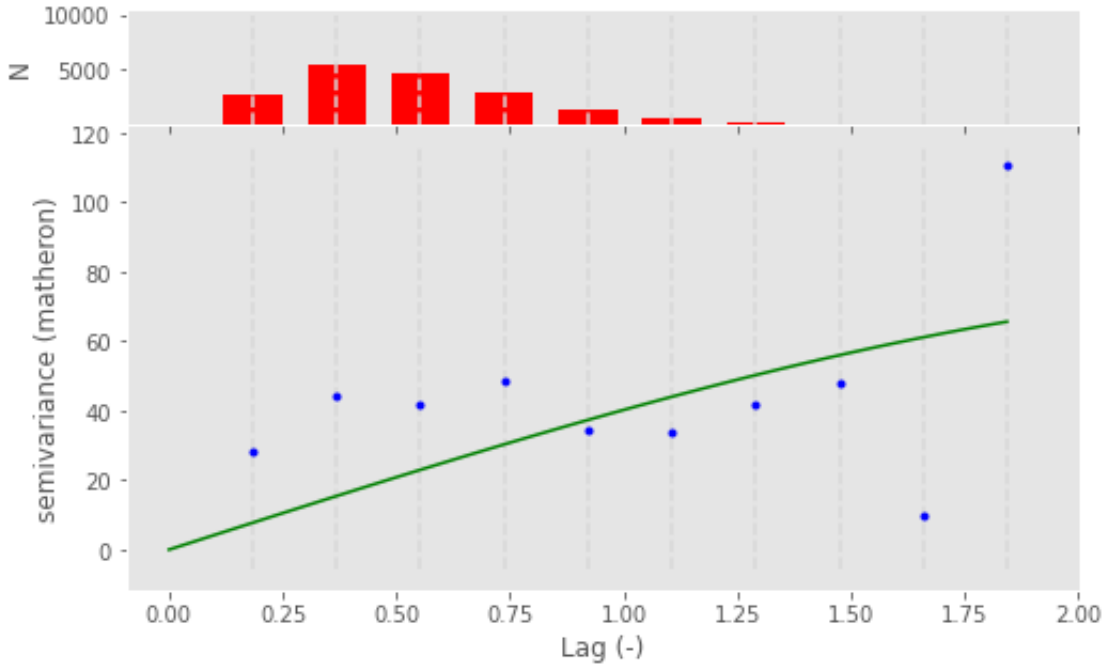
[ ]:
```
[<matplotlib.lines.Line2D at 0x7f721f879700>]
```



[ ]:
```
V.fit_method ='trf'

V.plot();

pprint(V.parameters)

V.fit_method ='lm'

V.plot();

pprint(V.parameters)
```

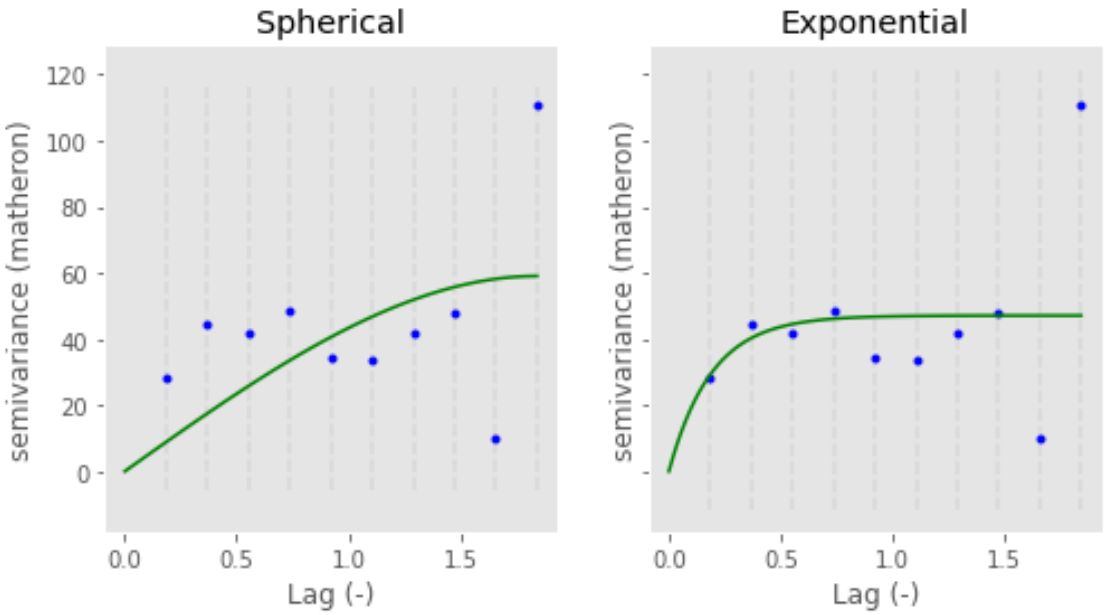
```
[1.842641044600634, 59.05712716397218, 0]
[2.7003943553274445, 75.9070546267365, 0]
```





[ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4), sharey=True)

axes[0].set_title('Spherical')
```

```
axes[1].set_title('Exponential')


V.fit_method = 'trf'

V.plot(axes=axes[0], hist=False)

# switch the model
V.model = 'exponential'

V.plot(axes=axes[1], hist=False);
```



[ ]:
```
# spherical
V.model = 'spherical'

rmse_sph = V.rmse

r_sph = V.describe().get('effective_range')

# exponential
V.model = 'exponential'

rmse_exp = V.rmse

r_exp = V.describe().get('effective_range')
```

```
print('Spherical   RMSE: %.2f' % rmse_sph)

print('Exponential RMSE: %.2f' % rmse_exp)
```

```
Spherical   RMSE: 26.41
Exponential RMSE: 24.08
```

[ ]:
```
print('Spherical effective range:    %.1f' % r_sph)

print('Exponential effective range:  %.1f' % r_exp)
```

```
Spherical effective range:    1.8
Exponential effective range:  0.6
```

[ ]:
```
df_west_up.describe().transpose()
```

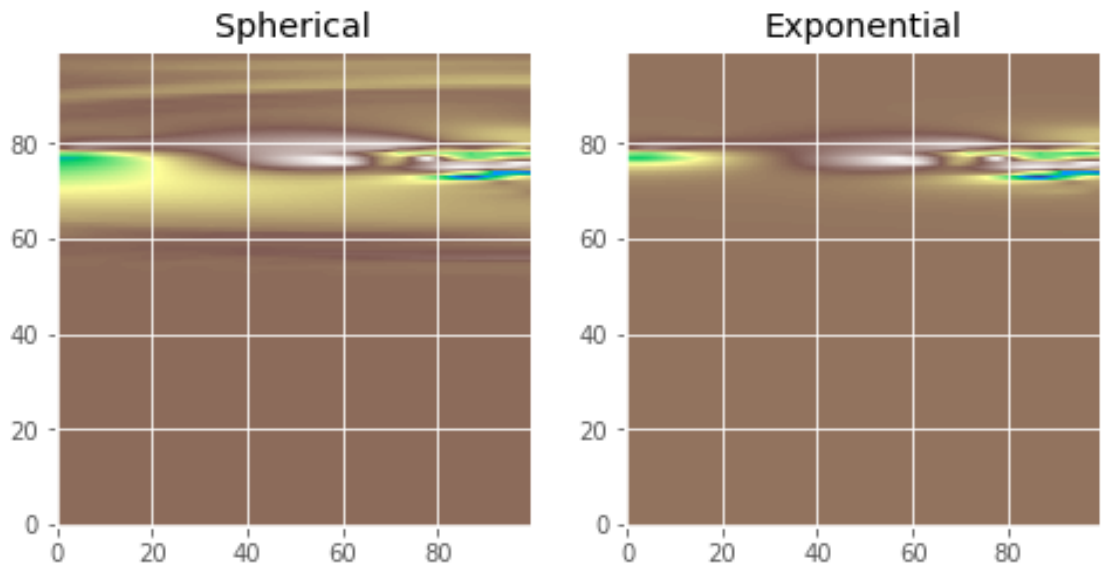
[ ]:
```
             count         mean       std          min          25%  \
Lon          193.0    77.625836  0.285003    77.005556    77.391111
Lat          193.0    28.963667  0.236686    28.008333    28.870000
Year         193.0  2015.000000  0.000000  2015.000000  2015.000000
PreMonsoon   193.0    12.976632  6.109389     1.900000     8.560000
PostMonsoon  193.0    12.728446  6.392181     1.380000     7.950000

                    50%          75%          max
Lon           77.700000    77.848611    78.250000
Lat           28.991667    29.129722    29.826389
Year        2015.000000  2015.000000  2015.000000
PreMonsoon    12.220000    17.330000    32.000000
PostMonsoon   11.520000    17.100000    32.130000
```

Krigging

[ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(8, 4))

V.model = 'spherical'

krige1 = V.to_gs_krige()

V.model = 'exponential'

krige2 = V.to_gs_krige()

# build a grid
x = np.arange(70, 80, 0.1)
y = np.arange(28, 29, 0.01)
```

```
# apply
field1, _ = krige1.structured((x, y))

field2, _ = krige2.structured((x, y))

# use the same bounds
vmin = np.min((field1, field2))

vmax = np.max((field1, field2))

# plot
axes[0].set_title('Spherical')

axes[1].set_title('Exponential')

axes[0].imshow(field1, origin='lower', cmap='terrain_r', vmin=vmin, vmax=vmax)

axes[1].imshow(field2, origin='lower', cmap='terrain_r', vmin=vmin, vmax=vmax)
```

[ ]: <matplotlib.image.AxesImage at 0x7f721ad02e50>

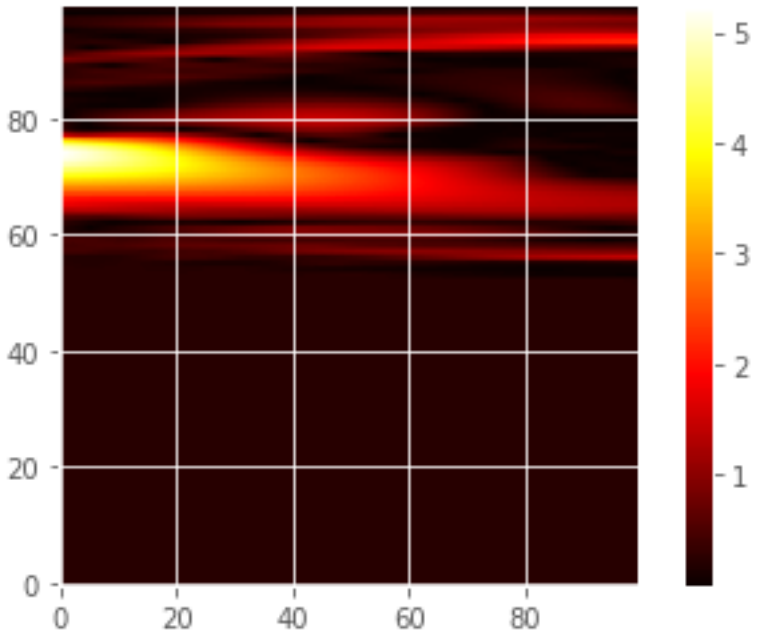


[ ]:
```
# calculate the differences
diff = np.abs(field2 - field1)

print('Mean difference:     %.1f' % np.mean(diff))

print('3rd quartile diffs.: %.1f' % np.percentile(diff, 75))
```

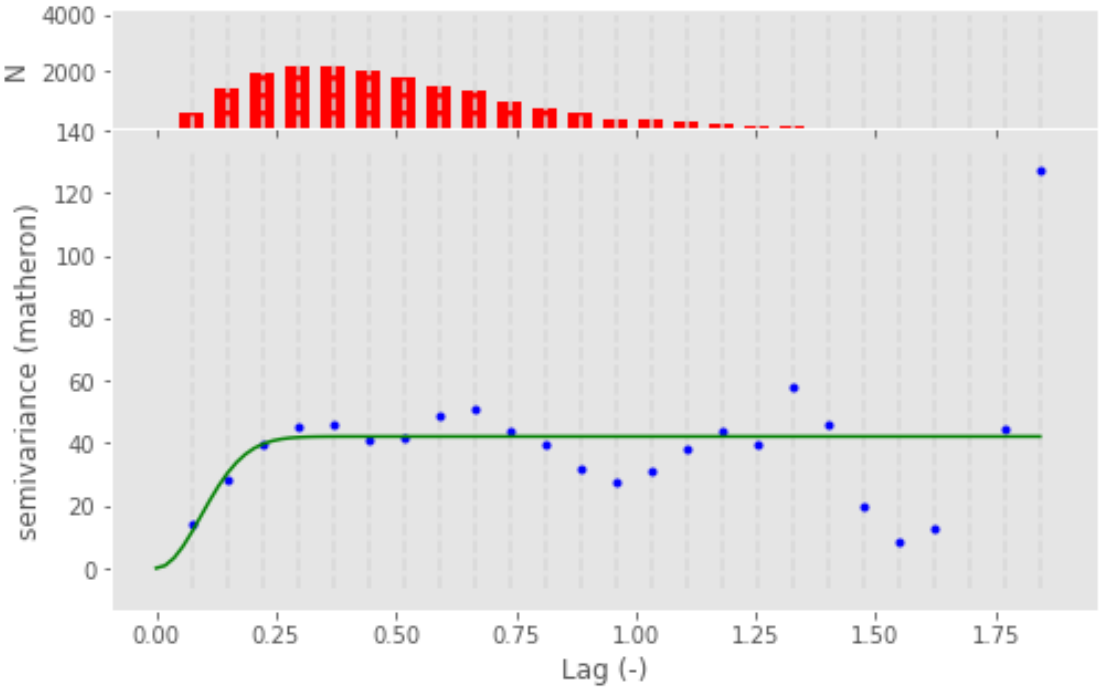
```
print('Max differences:     %.1f' % np.max(diff))

plt.imshow(diff, origin='lower', cmap='hot')

plt.colorbar()
```

```
Mean difference:     0.5
3rd quartile diffs.: 0.4
Max differences:     5.2
```

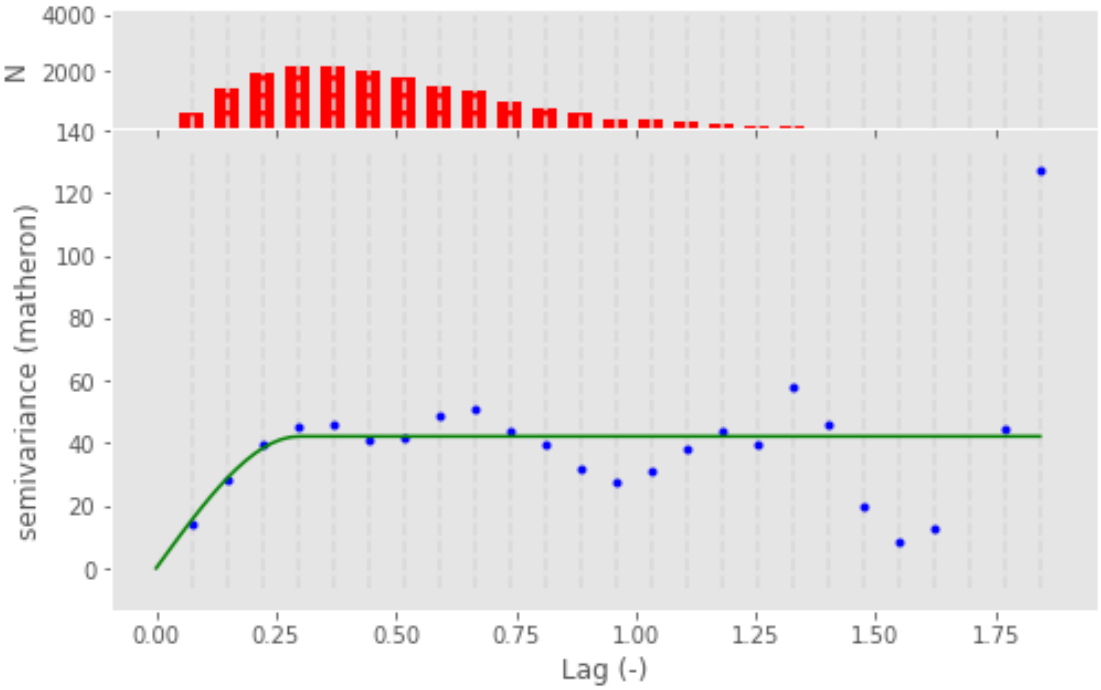
[ ]: <matplotlib.colorbar.Colorbar at 0x7f721f63ffa0>



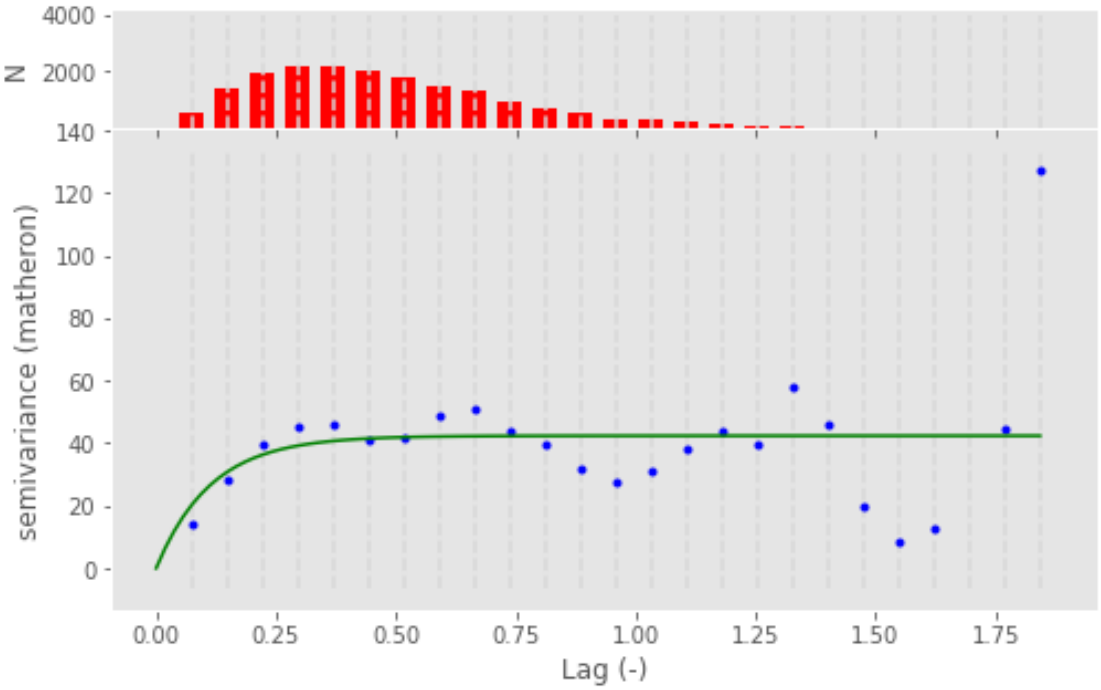
[ ]:
```
Vg = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)), df_west_up.
 ↪PostMonsoon.values,normalize=False, n_lags=25, maxlag=90, model='gaussian')
# V = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)), df_west_up.
 ↪PostMonsoon.values)


Vg.plot();
```



[ ]:
```
Vg = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)), df_west_up.
↪PostMonsoon.values,normalize=False, n_lags=25, maxlag=90, model='spherical')


Vg.plot();
```



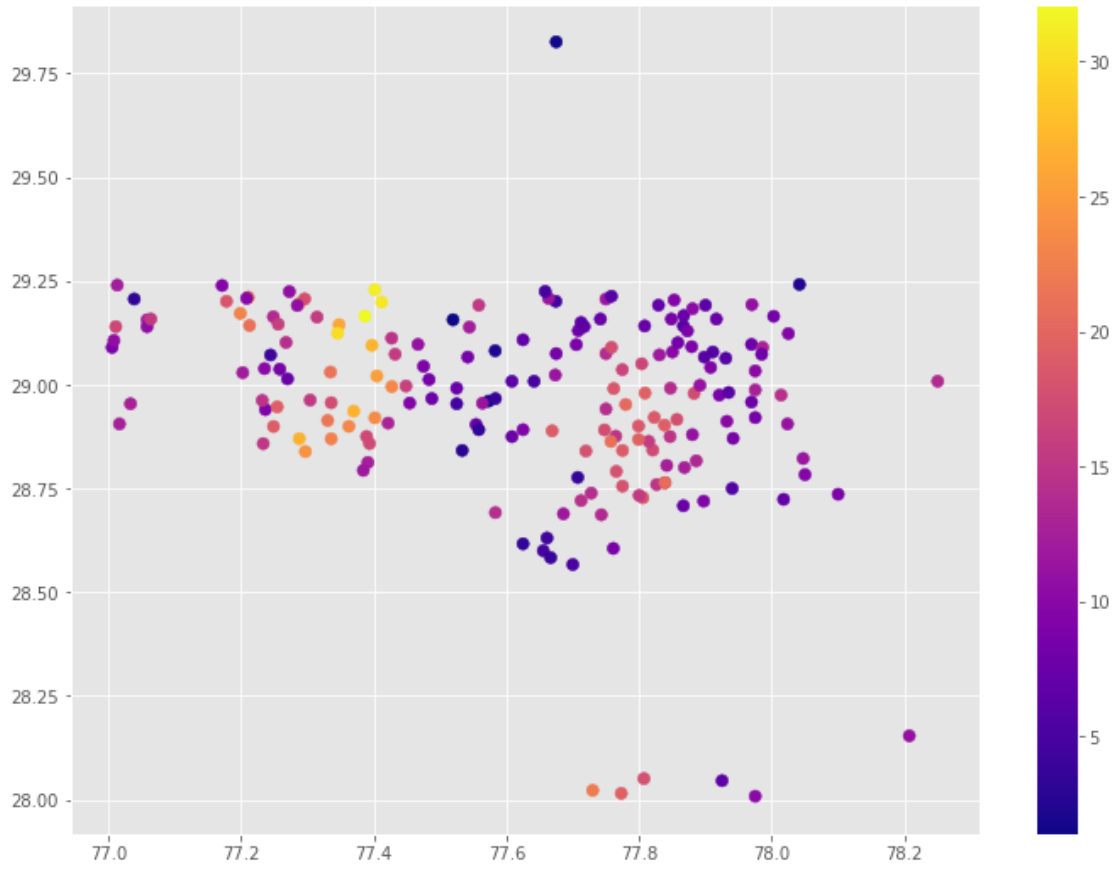
[ ]:
```
Vg = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)),
                   df_west_up.PostMonsoon.values,normalize=False,
                   n_lags=25, maxlag=90, model='exponential')


Vg.plot();
```



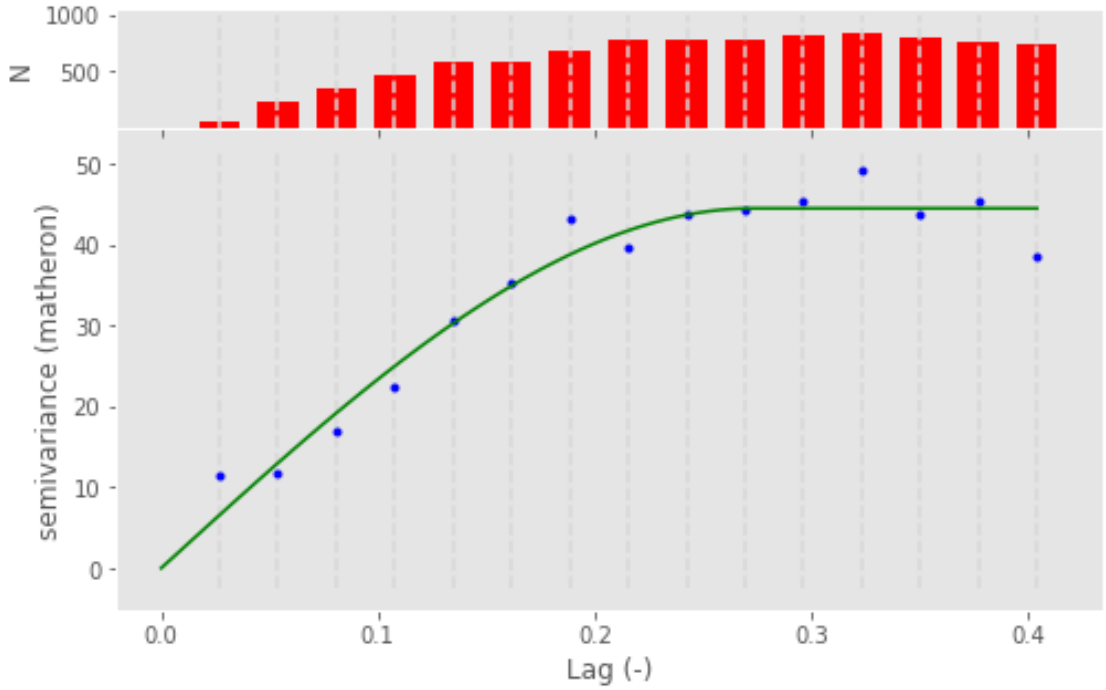
[ ]:
```
import skgstat as skg
```

[ ]:
```
fig, ax = plt.subplots(1, 1, figsize=(12, 9))
art = ax.scatter(df_west_up.Lon, df_west_up.Lat, s=50, c = df_west_up.
 ↪PostMonsoon.values, cmap='plasma')
plt.colorbar(art)
```

[ ]:
```
<matplotlib.colorbar.Colorbar at 0x7f7218aa9040>
```



[ ]:
```
V = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)),
                  df_west_up.PostMonsoon.values, maxlag='median', n_lags=15,␣
 ↪normalize=False)
fig = V.plot(show=False)
```



[ ]:
```
print('Sample variance: %.2f   Variogram sill: %.2f' % (df_west_up.PostMonsoon.
↪values.var(), V.describe()['sill']))
```

```
Sample variance: 40.65   Variogram sill: 44.52
```

[ ]:
```
pprint(V.describe())
```

```
{'dist_func': 'euclidean',
 'effective_range': 0.2729227005016303,
 'estimator': 'matheron',
 'kwargs': {},
 'model': 'spherical',
 'normalized_effective_range': 0.11027260660551338,
 'normalized_nugget': 0,
 'normalized_sill': 2189.9849701023877,
 'nugget': 0,
 'params': {'bin_func': 'even',
            'dist_func': 'euclidean',
            'estimator': 'matheron',
            'fit_method': 'trf',
            'fit_sigma': None,
            'maxlag': 0.40404336613566033,
            'model': 'spherical',
            'n_lags': 15,
            'normalize': False,
```

```
            'use_nugget': False,
            'verbose': False},
 'sill': 44.524502145743085}
```

[ ]:
```
print(V)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   0.27
Sill:              44.52
Nugget:            0.00
```

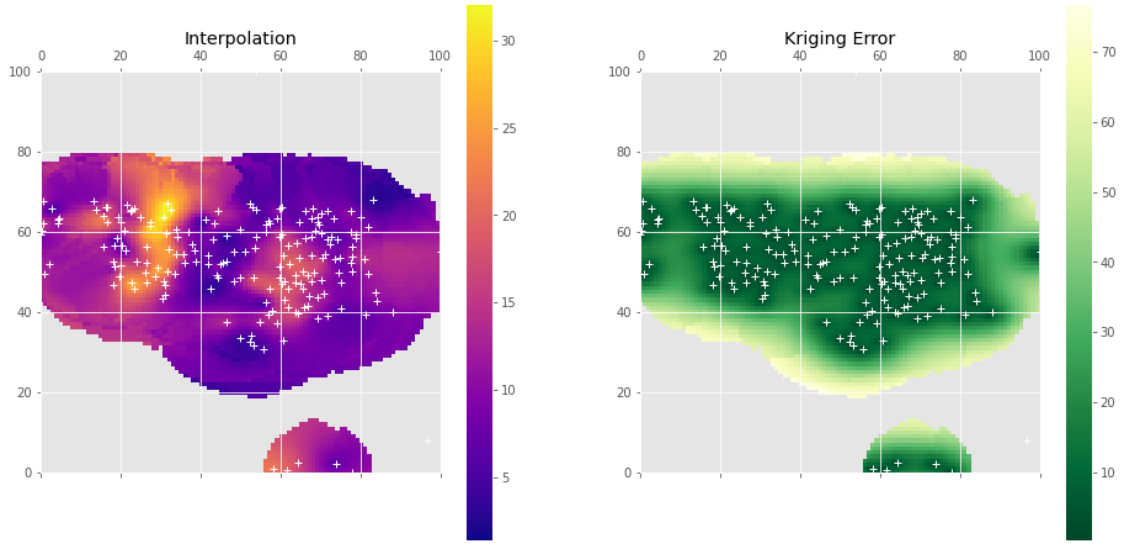
[ ]:
```
ok = skg.OrdinaryKriging(V, min_points=5, max_points=15, mode='exact')
```

[ ]:
```
x = df_west_up.Lon
y = df_west_up.Lat
xx, yy = np.mgrid[x.min():x.max():100j, y.min():y.max():100j]
field = ok.transform(xx.flatten(), yy.flatten()).reshape(xx.shape)
s2 = ok.sigma.reshape(xx.shape)
```

```
Warning: for 4763 locations, not enough neighbors were found within the range.
```

[ ]:
```
fig, axes = plt.subplots(1, 2, figsize=(16, 8))

# rescale the coordinates to fit the interpolation raster
x_ = (x - x.min()) / (x.max() - x.min()) * 100
y_ = (y - y.min()) / (y.max() - y.min()) * 100

art = axes[0].matshow(field.T, origin='lower', cmap='plasma', vmin=df_west_up.
 ↪PostMonsoon.values.min(), vmax=df_west_up.PostMonsoon.values.max())
axes[0].set_title('Interpolation')
axes[0].plot(x_, y_, '+w')
axes[0].set_xlim((0, 100))
axes[0].set_ylim((0, 100))
plt.colorbar(art, ax=axes[0])
art = axes[1].matshow(s2.T, origin='lower', cmap='YlGn_r')
axes[1].set_title('Kriging Error')
plt.colorbar(art, ax=axes[1])
axes[1].plot(x_, y_, '+w')
axes[1].set_xlim((0, 100))
axes[1].set_ylim((0, 100))
```

[ ]:
```
(0.0, 100.0)
```



[ ]:
```
df_3 = pd.read_csv("up_state_gwl_v6.csv")
```

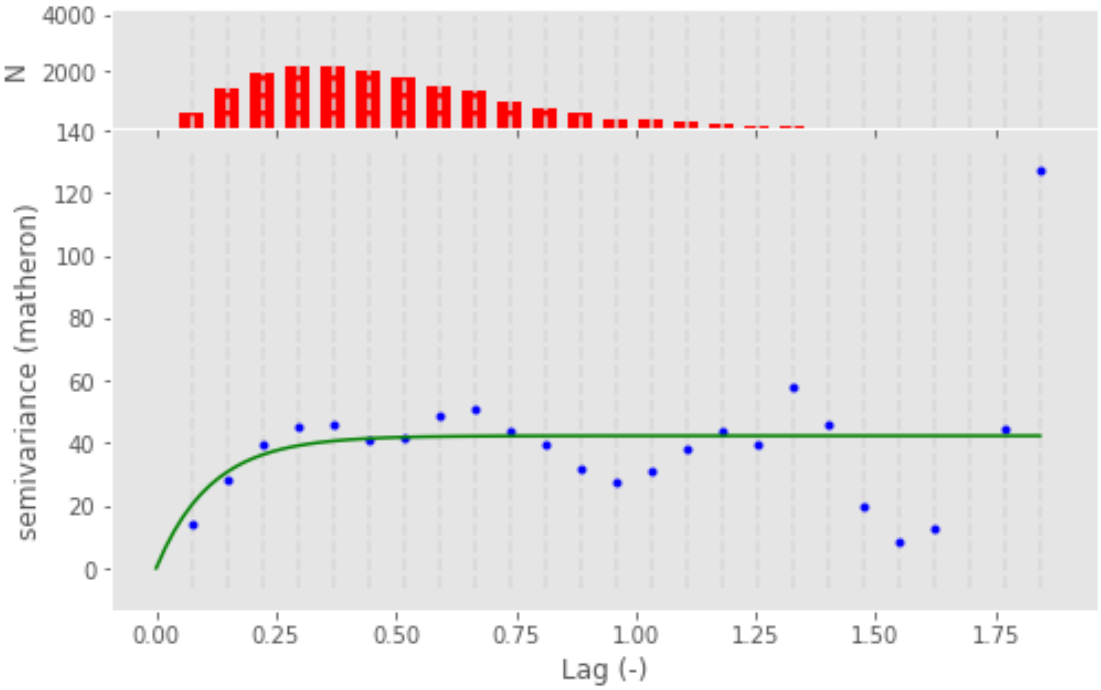
[ ]:
```
df_up = df_3[df_3["Year"] == 2015]
```

[ ]:
```
df_west_up = df_up[(df_up["District"] == "MUZAFFARNAGAR") | (df_up["District"]␣
↪== "GHAZIABAD") | (df_up["District"] == "BAGHPAT") | (df_up["District"] ==␣
↪"MEERUT") | (df_up["District"] == "HAPUR")]
```

[3]:
```
df_west_up = df_west_up.dropna()
df_west_up
```

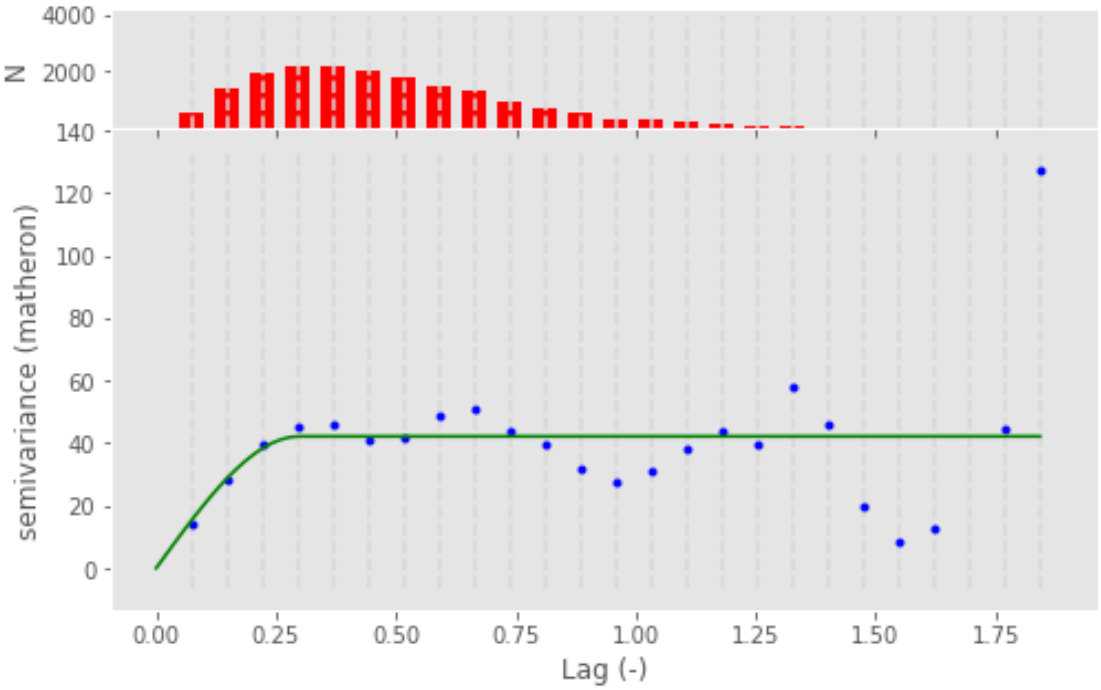
[ ]:
```
# Vg2_exp = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)),
                    df_west_up.PostMonsoon.values,normalize=False,
                    n_lags=25, maxlag=90, model='exponential')

plt.figure(figsize=(20, 20))
Vg2_exp.plot();
```

```
<Figure size 1440x1440 with 0 Axes>
```



[ ]:
```
Vg2_sph = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)),
                        df_west_up.PostMonsoon.values,normalize=False,
                        n_lags=25, maxlag=90, model='spherical')

plt.figure(figsize=(20, 20))
Vg2_sph.plot();
```

```
<Figure size 1440x1440 with 0 Axes>
```



[ ]:
```
print(Vg2_exp)
```

```
exponential Variogram
---------------------
Estimator:          matheron
Effective Range:    0.34
Sill:               42.32
Nugget:             0.00
```

[ ]:
```
print(Vg2_sph)
```

```
spherical Variogram
-------------------
Estimator:          matheron
Effective Range:    0.30
Sill:               42.14
Nugget:             0.00
```

1.2.1 Extra Stuff Related to PLotting

[ ]:
```
import skgstat as skg
from skgstat.plotting import backend
import numpy as np
import json
import warnings
import matplotlib.pyplot as plt
from plotly.subplots import make_subplots
warnings.filterwarnings('ignore')
```

Estimate a variogram, with a few more lag classes, as there are enough observation points available.

[ ]:
```
V = skg.Variogram(list(zip(df_west_up.Lon, df_west_up.Lat)),df_west_up.
↪PostMonsoon.values, n_lags=25)
print(V)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   0.30
Sill:              42.14
Nugget:            0.00
```

Estimate the directional variogram with a few more lag classes and an azimuth of 90°. The tolerance is set rather low to illustrate the graphs better (fewer point connections.):

[ ]:
```
DV = skg.DirectionalVariogram(list(zip(df_west_up.Lon, df_west_up.Lat)),␣
↪df_west_up.PostMonsoon.values, n_lags=20, azimuth=40., tolerance=15.0)
print(DV)
```

```
spherical Variogram
-------------------
Estimator:         matheron
Effective Range:   0.25
Sill:              36.16
Nugget:            0.00
```

1.2.2 Backend

You can switch to plotly as a plotting backend by calling the :mod:plotting.backend function and passing the name of the backend. Note that plotly is only a soft dependency and will not automatically be installed along with SciKit-GStat. You can install it like:

.. code-block:: bash

pip install plotly

Note that in a Jupyter environment you might want to use the plotly.offline environment to embed the needed Javascript into the notebook. In these cases you have to catch the Figure object and use the iplot function from the offline submodule.

1.2.3 Variogram

:func:Variogram.plot <skgstat.Variogram.plot> The :func:Variogram.plot <skgstat.Variogram.plot> is the main plotting function in SciKit-GStat. Before you use the variogram for further geostatistical methods, like kriging, or further analysis, make sure, that a suitable model was found and fitted to the experimental data. Further, you have to make sure that the statistical foundation of this estimation is sound, the lag classes are well designed and backed by a suiatable amount of data. Otherwise, any other geostatistical analysis or method will have to fail, no matter how nice the results might look like.

[ ]:
```
from skgstat.plotting import backend
backend('plotly')
```

Plotly

[ ]:
```
fig = V.plot(show=False)
fig
```

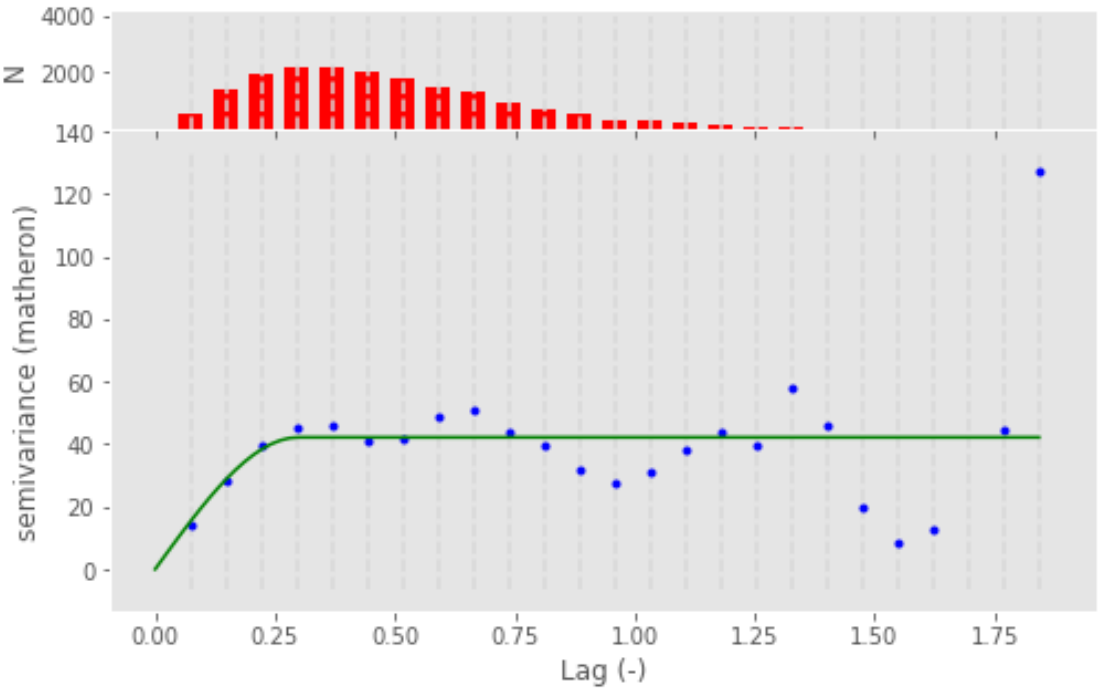
The :func:Variogram.plot <skgstat.Variogram.plot> functions is customizable and takes a lot of arguments. However, the same interface is used as for the matplotlib version of that function. Many matplotlib arguments are mapped to the corresponding plotly arguments. Beyond that, you can either try common plotly arguments, or update the figure afterwards:

[ ]:
```
fig = V.plot(show=False)

fig.update_layout(
    legend=dict(x=0.05, y=1.1, xanchor='left', yanchor='top', orientation='h'),
    template='plotly_dark',
    annotations=[dict(
        xref="paper",
        yref="paper",
        x=0.5,
        y=0.5,
        font=dict(color="white", size=100),
        textangle=-30,
        opacity=.3
    )]
```

```
)
fig
```

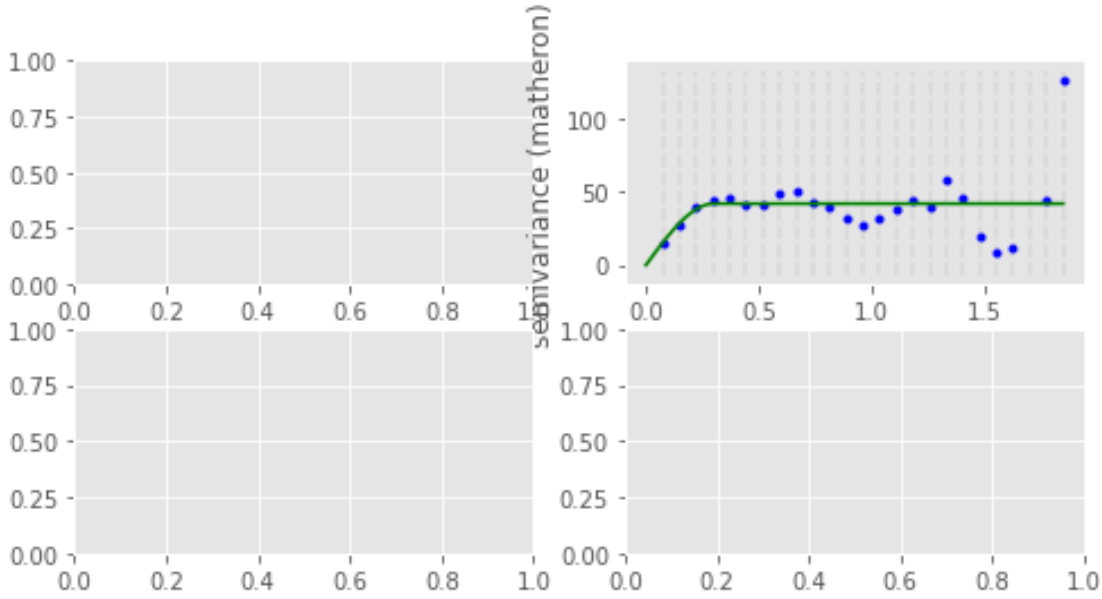
Matplotlib

[ ]:
```
backend('matplotlib')

fig = V.plot()
```



With matplotlib, you can set any matplotlib.AxesSubplot as axes to plot on other figures. You can send two axes, for the variogram and the histogram, or only one and disable the histogram plotting.

[ ]:
```
fig, ax = plt.subplots(2,2, figsize=(8, 4))

fig = V.plot(axes=ax.flatten()[1], hist=False)
```



Variogram.scattergram‘ You can plot a scattergram of all point pairs formed by the class. The pairs can be grouped by the lag classes, they were formed in. This way you can analyze how the two values of the point pait (head and tail) scatter and if this follows a pattern (i.e. anisotropy). It is recommended to use the 'plotly' backend, as you can click on the legend entries to hide a specific class, or double-click to show only the selected lag class. This makes it much easier to inspect the classes.

Plotly

[ ]:
```
backend('plotly')
fig = V.scattergram(show=False)
fig
```

It is, however possible to re-create the plot that was used up to SciKit-GStat version 0.3.0 with only one color. This is still the default for the 'matplotlib' backend.

[ ]:
```
fig = V.scattergram(single_color=True, show=False)
fig
```

Matplotlib backend('matplotlib')

[ ]:
```
fig = V.scattergram()
```

1.2.4 Variogram.location_trend

Another useful helper plot is the :func:location_trend <skgstat.Variogram.location_trend>. This will plot the observation values related to their coordinate position, for each coordinate dimension separatedly. With the 'plotly' backend, each dimension will appear as a coloured group in a single plot. By double-clicking the legend, you can inspect each group separately.

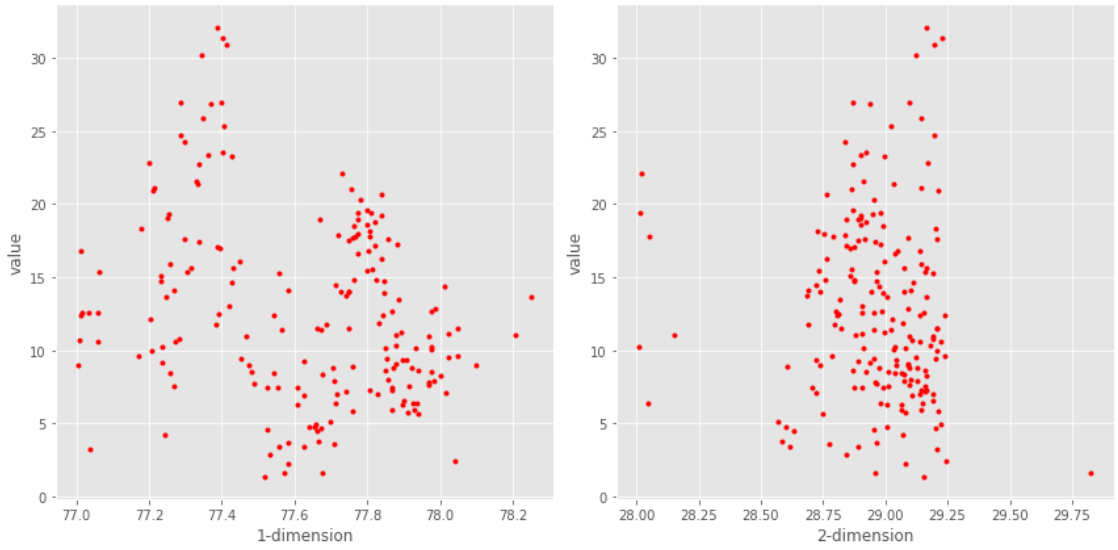
The 'plotly' backend will automatically switch the used plot type from a ordinary scatter-plot to a WebGL backed scatter-plot, if there are more than 5000 observations. This will add some startup-overhead for the plot to appear, but the interactivity actions (like pan, zoom) are speed up by magnitudes.

Plotly

[ ]:
```
backend('plotly')
fig = V.location_trend(show=False)
fig
```

Since version 0.3.5 the :func:location_trend <skgstat.Variogram.location_trend># function accepts a add_trend_line parameter, that defaults to False. If set to true, the class will fit linear models to each of the point clouds and output a trend line. It will also calculate the R^2, which you can use to either accept the input data as trend free or not (a high R^2 indicates a **linear** trend and hence you should decline using the input data).

[ ]:
```
fig = V.location_trend(add_trend_line=True, show=False)
fig

# Matplotlib
# """""""""""
#
# There is a difference between the ``'matplotlib'`` and ``'plotly'`` backend in
# this plotting function. As Plotly utilizes the legend by default to show and
# hide traces on the plot, the user can conveniently switch between the
# coordinate dimensions.
# In Matplotlib, the figures are not interactive by default and therefore
# SciKit-GStat will create one subplot for each coordinate dimension.
backend('matplotlib')
fig = V.location_trend()
```



1.2.5 :func:distance_difference plot <skgstat.Variogram.distance_difference_plot>

The final utility plot presented here is a scatter-plot that relates all pairwise-differences in value to the spatial distance of the respective point pairs. This can already be considered to be a variogram. For convenience, the plotting method will mark all upper lag class edges in the plot. This can already give you an idea, if the number of lag classes is chosen wisely, or if you need to adjust. To estimate valid, expressive variograms, this is maybe the most important preparation step. If your lag classes do not represent your data well, you will never find a useful variogram.

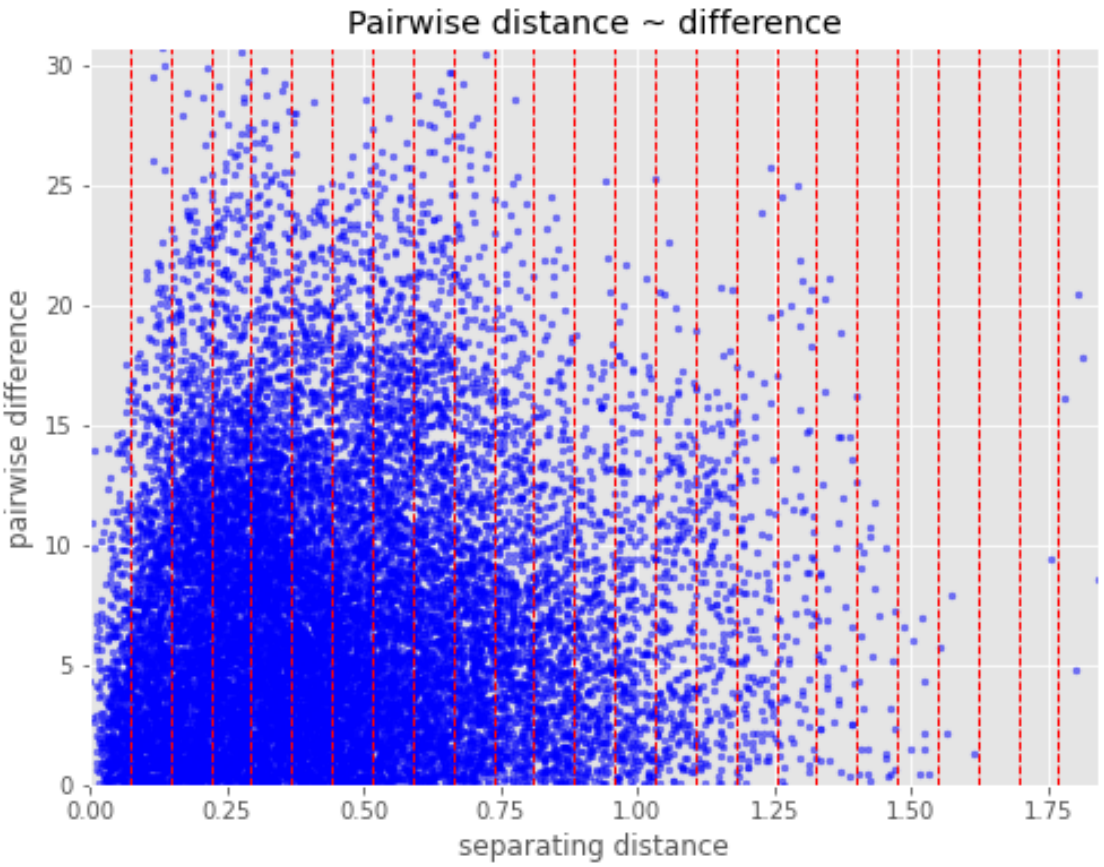
Plotly

[ ]:
```
backend('plotly')
fig = V.distance_difference_plot(show=False)
fig
```

You might also consider to adapt the maximum lag distance using this plot, to exclude distances that are not well backed by data. Alternatively, the binning method can be changed. Or both

[ ]:
```
Vcopy = V.clone()
Vcopy.bin_func = 'uniform'

fig = Vcopy.distance_difference_plot(show=False)
fig
```

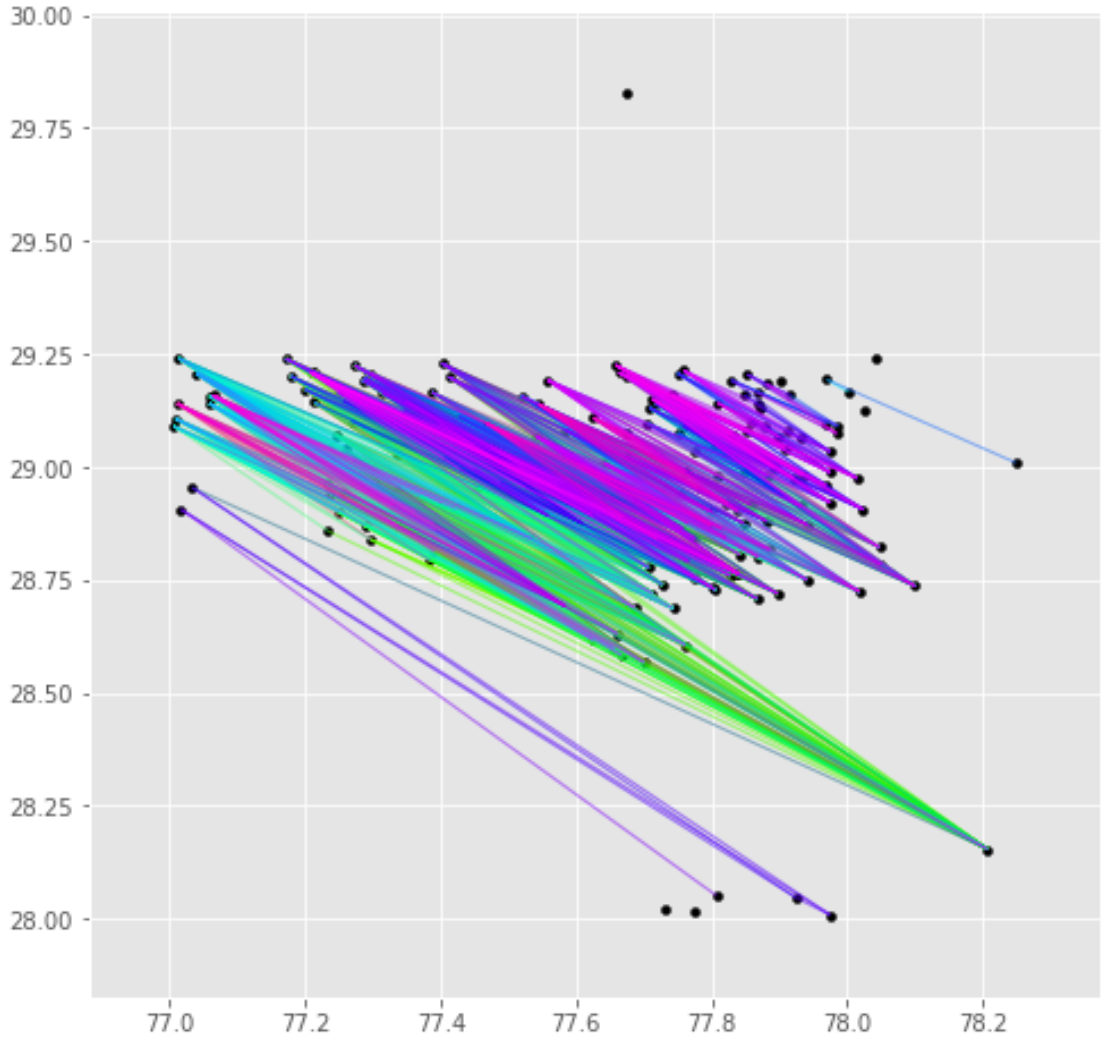
```
# Matplotlib
# """""""""""
backend('matplotlib')
fig = V.distance_difference_plot()
```



1.2.6 Directional Variogram

:func:pair_field <skgstat.DirectionalVariogram.pair_field> The :class:DirectionalVariogram <skgstat.DirectionalVariogram> class is inheriting from :class:Variogram <skgstat.Variogram>. Therefore all plotting method shown above are available for directional variograms, as well. Additionally, there is one more plotting method, :func:DirectionalVariogram.pair_field <skgstat.DirectionalVariogram.pair_field>. This function will plot all coordinate locations and draw a line between all point pairs, that were not masked by the directional mask array and will, thus, be used for variogram estimation. By default, the method will draw all lines for all point pairs and you will see nothing on the plot. But there is also the possibility to draw these lines only for a subset of the coordinate locations.

[ ]:
```
# Matplotlib
# """""""""""
backend('matplotlib')
fig = DV.pair_field()
```



1.3 Thank You

Compiled by: Tanay Singhal (MT21177)

1.4 References:

- https://geopandas.org/en/stable/
- https://scikit-gstat.readthedocs.io/en/latest/auto_examples/tutorial_06_gstools.html

- https://github.com/GeostatsGuy/GeostatsPy
- https://www.arcgis.com/home/index.html
- https://rasterio.readthedocs.io/en/latest/