Lecture 8: Visualizing and modelling spatial data

Contents

Lecture Outline

Lecture Learning Objectives

- Make informed choices about how to plot your spatial data, e.g., scattered, polygons, 3D, etc..
- Plot spatial data using libraries such as `geopandas`, `plotly`, and `keplergl`.
- Interpolate unobserved spatial data using deterministic methods such as nearest-neighbour interpolation.
- Interpolate data from one set of polygons to a different set of polygons using areal interpolation.

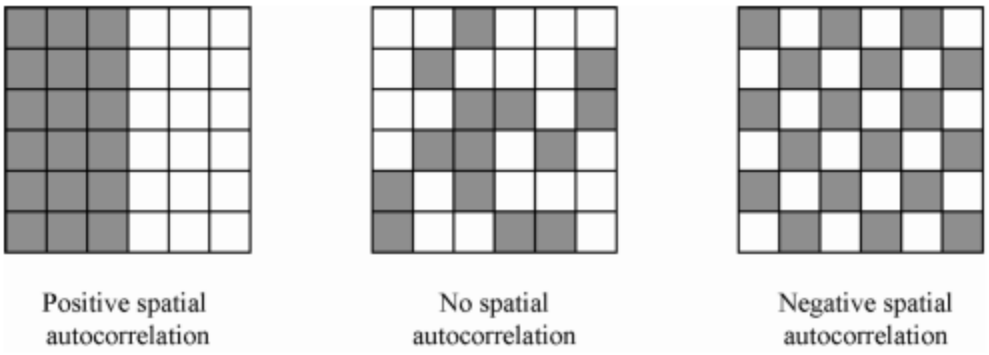
Imports

```
import warnings
import keplergl
import numpy as np
import osmnx as ox
import pandas as pd
import geopandas as gpd
import plotly.express as px
import IPython
import base64
from skgstat import Variogram
import matplotlib.pyplot as plt
from shapely.geometry import Point
from pykrige.ok import OrdinaryKriging
from scipy.interpolate import NearestNDInterpolator
from tobler.area_weighted import area_interpolate
# Custom functions
from scripts.utils import pixel2poly
# Plotting defaults
plt.style.use('ggplot')
px.defaults.height = 400; px.defaults.width = 620
plt.rcParams.update({'font.size': 16, 'axes.labelweight': 'bold', 'figure.figs
```

1. Spatial visualization

1.1. Geopandas

Spatial visualization can help us gain insights about spatial correlation:



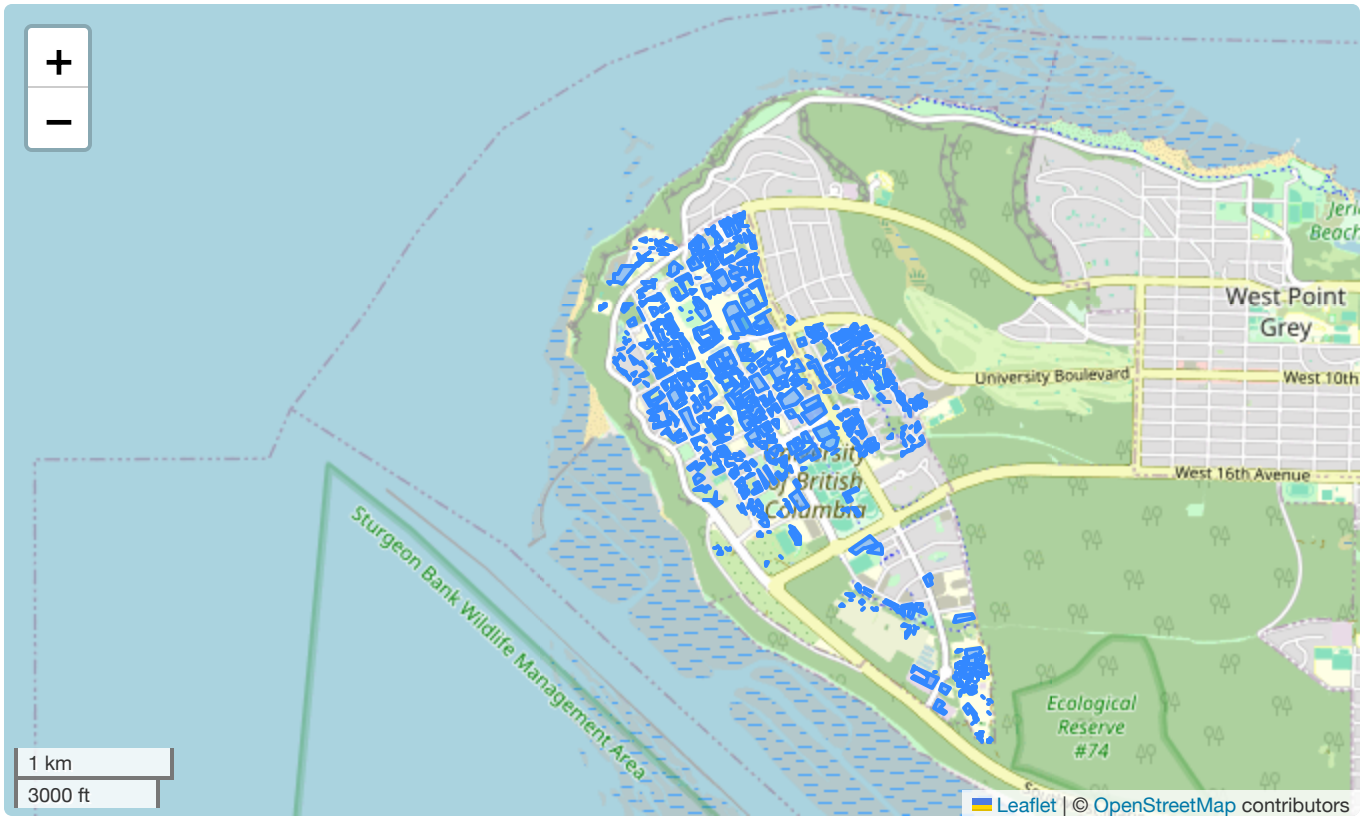
Source: https://www.r-bloggers.com/2015/09/correction-for-spatial-and-temporal-auto-correlation-in-panel-data-using-r-to-estimate-spatial-hac-errors-per-conley/

- We saw last lecture how to easily plot geospatial data using the `geopandas` method `.plot()`
- This workflow is useful for making quick plots, exploring your data, and easily layering geometries
- Let's import some data of UBC buildings using `osmnx` (our Python API for accessing OpenStreetMap data) and make a quick plot:

```
ubc = (ox.features.features_from_place("University of British Columbia, Canada
                                       tags={'building':True}) # Just keep building f
          .loc[:, ["geometry"]]                 # just keep the geometry column
          .assign(Label="Building Footprints")  # assign a label for later use
          .reset_index(drop=True)               # reset to 0 integer indexing
       )

ubc = ubc.loc[ubc.geometry.geometry.type=='Polygon'] # Include only polygons
ubc.head()
```

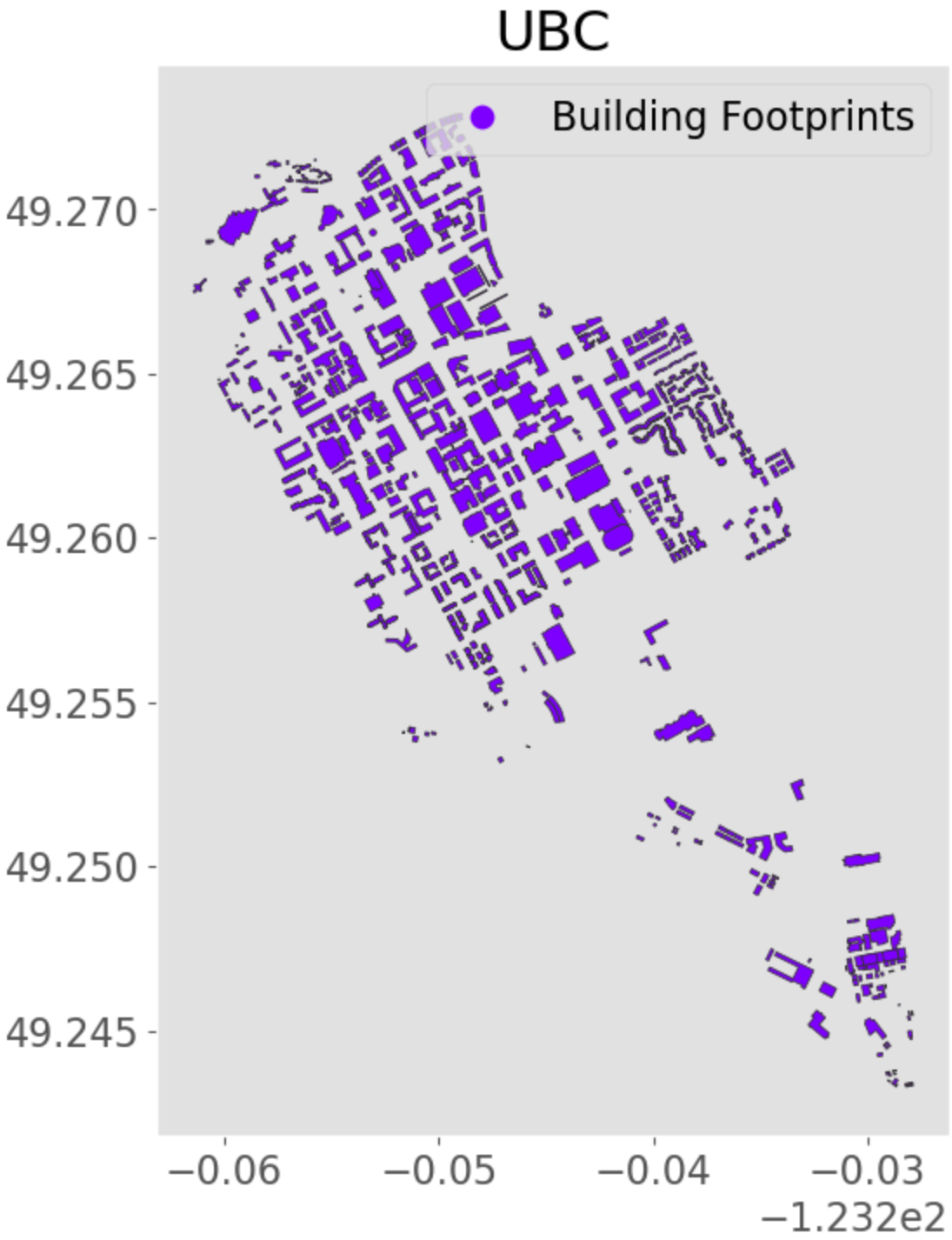
	geometry	Label
5	POLYGON ((-123.23018 49.24778, -123.23016 49.2...	Building Footprints
6	POLYGON ((-123.24738 49.26469, -123.24734 49.2...	Building Footprints
7	POLYGON ((-123.24738 49.26469, -123.24723 49.2...	Building Footprints
8	POLYGON ((-123.25786 49.27035, -123.25798 49.2...	Building Footprints
9	POLYGON ((-123.25782 49.27018, -123.25769 49.2...	Building Footprints

```
ubc.explore()
```



- Recall that we can make a plot using the `.plot()` method on a `GeoDataFrame`:

```
ax = ubc.plot(figsize=(8, 8), column="Label", legend=True,
              edgecolor="0.2", markersize=200, cmap="rainbow")
plt.title("UBC");
```



- Say I know the "point" location of my office but I want to locate the building footprint (a "polygon"). That's easily done with `geopandas`!
- I'll use `shapely` (the Python geometry library `geopandas` is built on) to make my point, but you could also use the `geopandas` function `gpd.points_from_xy()` like we did last lecture:

```
point_office = Point(-123.25203756532703,49.26314716306668)
point_office
```

- I can use the `.contains()` method to find out which building footprint my office resides in:

```
ubc.contains(point_office)
```

```
5      False
6      False
7      False
8      False
9      False
       ...
629    False
630    False
631    False
632    False
633    False
Length: 629, dtype: bool
```

```
ubc[ubc.contains(point_office)]
```

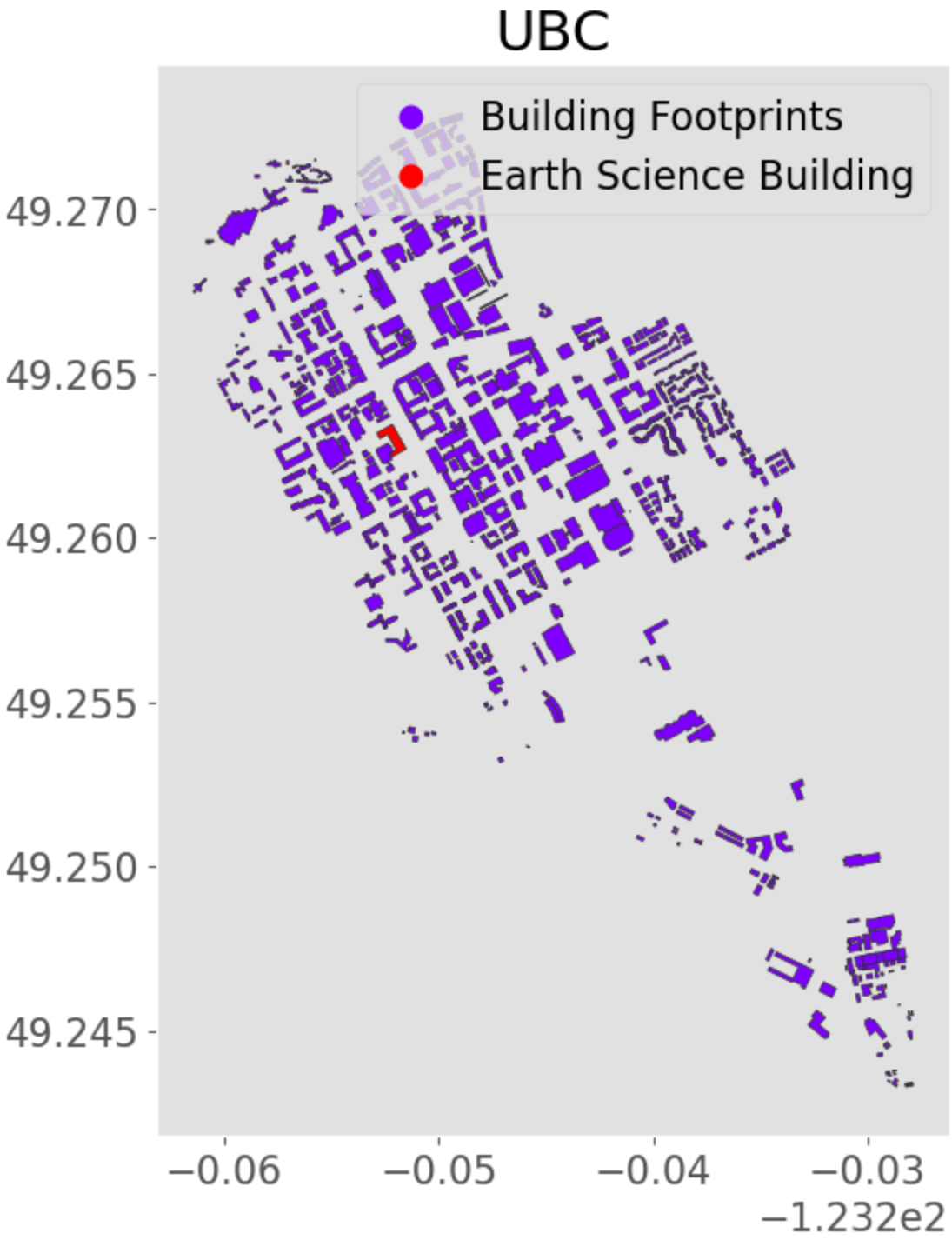
	geometry	Label
61	POLYGON ((-123.25217 49.26345, -123.25196 49.2...	Building Footprints

- Looks like it's index 61! I'm going to change the label of that one to "Earth Science Building":

```
ubc.loc[61, "Label"] = "Earth Science Building"
```

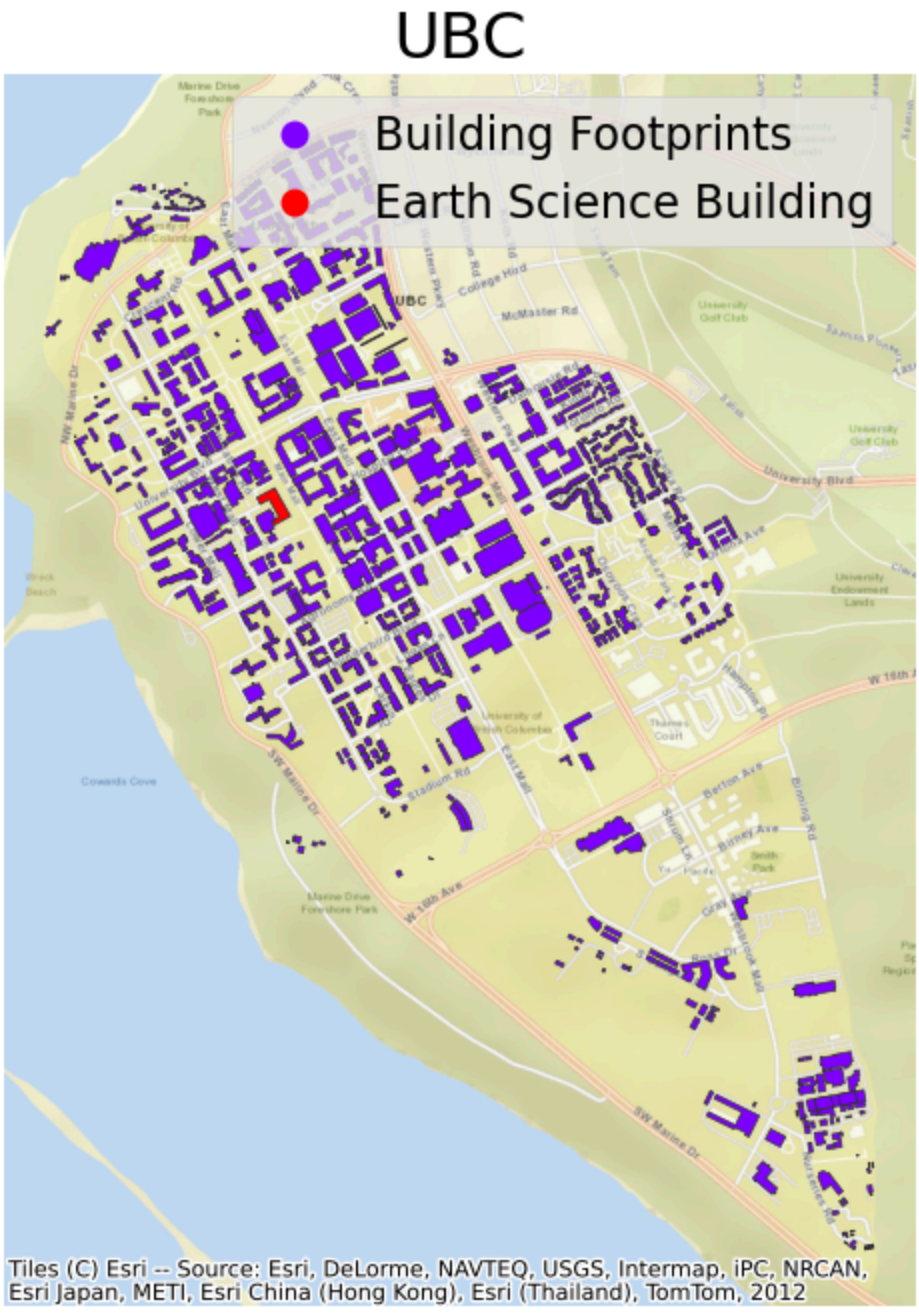
- Now let's make a plot!

```
ax = ubc.plot(figsize=(8, 8), column="Label", legend=True,
              edgecolor="0.2", markersize=200, cmap="rainbow")
plt.title("UBC");
```



- We can add more detail to this map by including a background map
- For this, we need to install the contextily package
- Note that most web providers use the Web Mercator projection, "EPSG:3857" (interesting article on that here) so I'll convert to that before plotting:

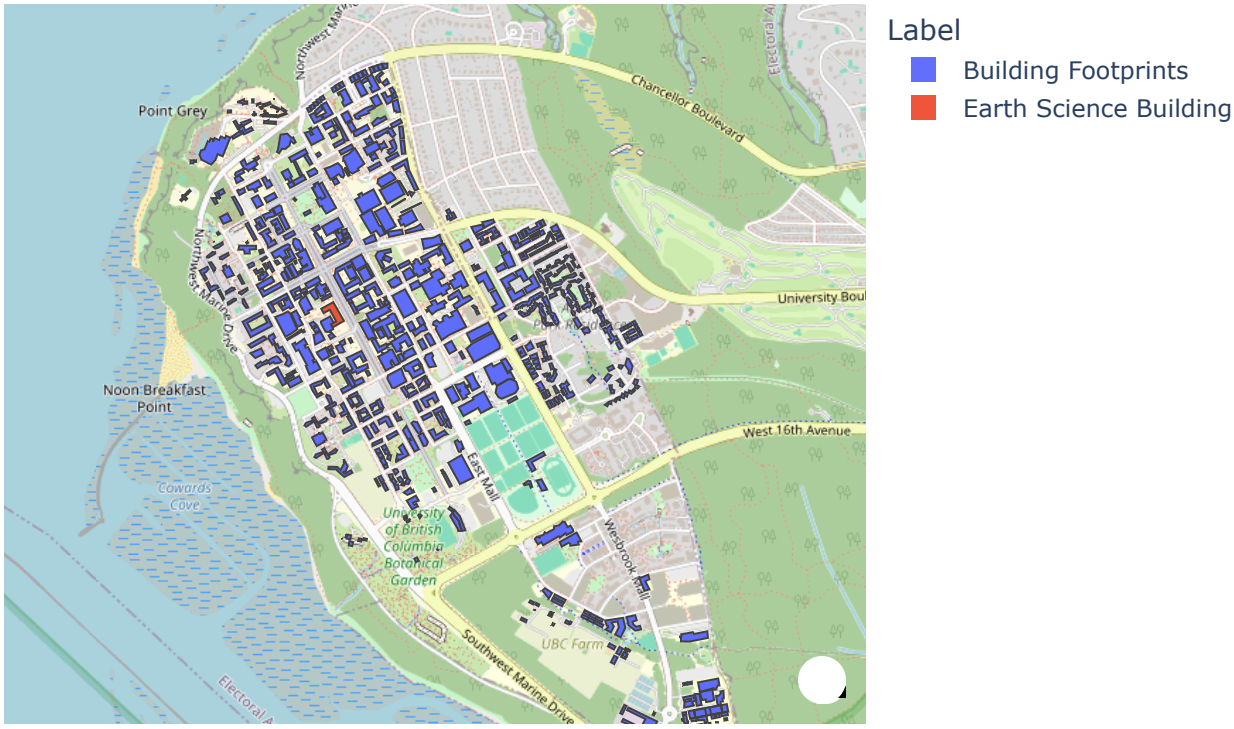
```
import contextily as cx

ax = (ubc.to_crs("EPSG:3857")
         .plot(figsize=(10, 8), column="Label", legend=True,
               edgecolor="0.2", markersize=200, cmap="rainbow")
     )
cx.add_basemap(ax, source=cx.providers.Esri.WorldStreetMap) # See all provides
plt.axis("off")
plt.title("UBC");
```



1.2. Plotly

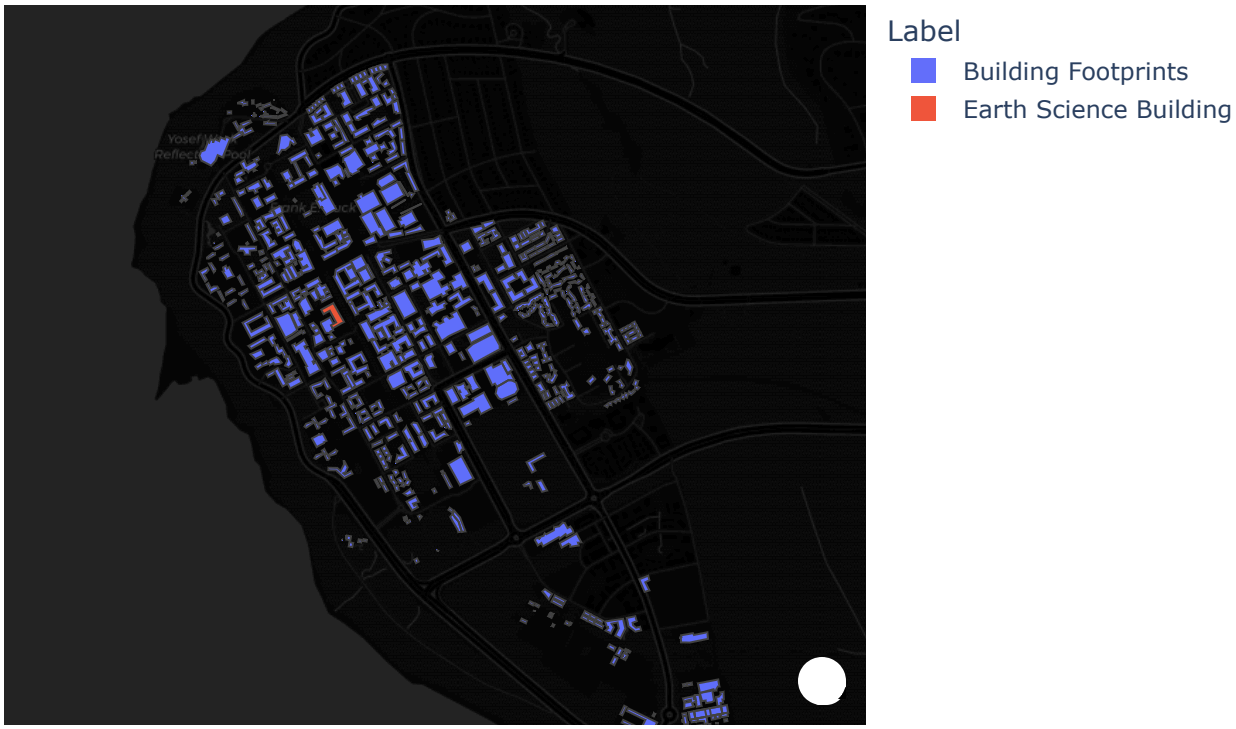
- The above map is nice, but it sure would be helpful in some cases to have interactive functionality for our map don't you think? Like the ability to zoom and pan?
- Well there are several packages out there that can help us with that, but as you know, I like to use `plotly` and it's one of the best for this kind of mapping. `plotly` supports plotting of maps backed by MapBox (a mapping and location data cloud platform)
- Here's an example using the `plotly.express` function `px.choropleth_mapbox()`:

```
fig = px.choropleth_mapbox(ubc, geojson=ubc.geometry, locations=ubc.index, col
                           center={"lat": 49.261, "lon": -123.246}, zoom=12.5,
                           mapbox_style="open-street-map")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
```



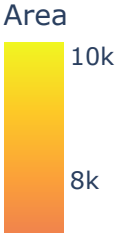
- You can pan and zoom above as you desire!
- You can change the `mapbox_style` as desired. Possible options that don't require a Mapbox API token include'open-street-map', 'white-bg', 'carto-positron', and 'carto-darkmatter'.

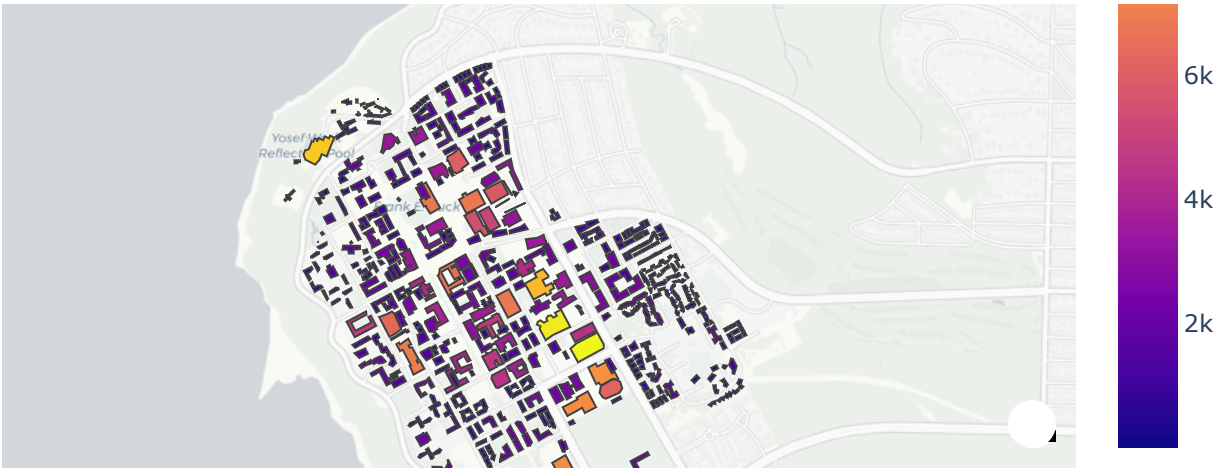
```
fig = px.choropleth_mapbox(ubc, geojson=ubc.geometry, locations=ubc.index, col
                           center={"lat": 49.261, "lon": -123.246}, zoom=12.5,
                          mapbox_style = "carto-darkmatter")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
```



- We are just colouring our geometries based on a label at the moment, but we could of course do some cooler stuff
- Let's colour by building area:

```
# Calculate area
ubc["Area"] = ubc.to_crs(epsg=3347).area  # note I'm projecting to EPSG:3347 (

# Make plot
fig = px.choropleth_mapbox(ubc, geojson=ubc.geometry, locations=ubc.index, col
                           center={"lat": 49.261, "lon": -123.246}, zoom=12.5,
                           mapbox_style="carto-positron")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
```





- Check out the `plotly` documentation for more - there are many plotting options and examples to learn from!
- Other popular map plotting options include altair (doesn't support interactivity yet), folium, and bokeh

1.3. Kepler.gl

- The above mapping was pretty cool, but are you ready for more power?
- Time to introduce kepler.gl!
- `keplergl` is a web-based tool for visualizing spatial data. Luckily, it has a nice Python API and Jupyter extension for us to use (see the install instructions)
- The way `keplergl` works is:
    1. We create an instance of a map with `keplergl.KeplerGl()`
    2. We add as much data to the map as we like with the `.add_data()` method
    3. We customize and configure the map in any way we like using the GUI (graphical user interface)

```
from keplergl import KeplerGl
import json

with open('keplergl/ubc_config.json', 'r') as file:
    ubc_config = json.load(file)

ubc_map = keplergl.KeplerGl(height=500)
ubc_map.add_data(data=ubc.copy(), name="Building heights")
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

```
# # # # render keplergl in jupyter lab
orig_html = str(ubc_map._repr_html_(),'utf-8')
b64d_html = base64.b64encode(orig_html.encode('utf-8')).decode('utf-8')
framed_html = f'<iframe src="data:text/html;base64,{b64d_html}" style="width:9
IPython.display.HTML(framed_html)
```

- As you can see from the plot above, there were some extra data that locate outside of our UBC campus.
- I'm going to combine this with our `ubc` data and do a bit of clean-up and wrangling. You'll do this workflow in your lab too so I won't spend too much time here:

```
warnings.filterwarnings("ignore")

ubc_bldg_heights = (gpd.sjoin(ubc.drop(columns='Label'), ubc_bldg_heights[["hgt_agl",
                      .drop(columns="index_right")
                      .rename(columns={"hgt_agl": "Height"})
                      .reset_index()
                      .dissolve(by="index", aggfunc="mean")  # dissolve is like "grou
                    )

ubc_bldg_heights
```

index	geometry	Area	Height
5	POLYGON ((-123.23027 49.24767, -123.23026 49.2...	1868.062531	6.965
6	POLYGON ((-123.24729 49.2648, -123.24731 49.26...	4090.167782	11.031
7	POLYGON ((-123.24723 49.26452, -123.2472 49.26...	1805.445252	16.905
8	POLYGON ((-123.25786 49.27035, -123.25798 49.2...	381.585003	8.610
9	POLYGON ((-123.25769 49.27023, -123.25763 49.2...	583.877733	8.020
...	...	...	...
620	POLYGON ((-123.22917 49.24463, -123.22901 49.2...	138.215510	3.240
621	POLYGON ((-123.22956 49.2443, -123.22953 49.24...	181.104079	4.180
622	POLYGON ((-123.22817 49.24542, -123.22809 49.2...	101.933506	5.260
631	POLYGON ((-123.25329 49.27101, -123.25318 49.2...	572.424976	6.290
632	POLYGON ((-123.23298 49.25247, -123.23304 49.2...	1704.929092	4.560

513 rows × 3 columns

- The reason why we used `dissolve(by="index", aggfunc="mean")` is that when joining these two data sets, sometimes there could be multiple buildings, hence multiple heights, within the same polygon. Therefore, after computing the means, we can take the average height as the final value of height for that polygon.
- Another example would be when we want to join datasets to calculate the building area of the west campus, we can use `dissolve(by="index", aggfunc="sum")` to sum up the areas of all buildings located in the west campus.

- Now I'll make a new map and configure it to be in 3D using the GUI!

```
import json
with open('keplergl/ubc_config.json', 'r') as file:
    ubc_config = json.load(file)
```

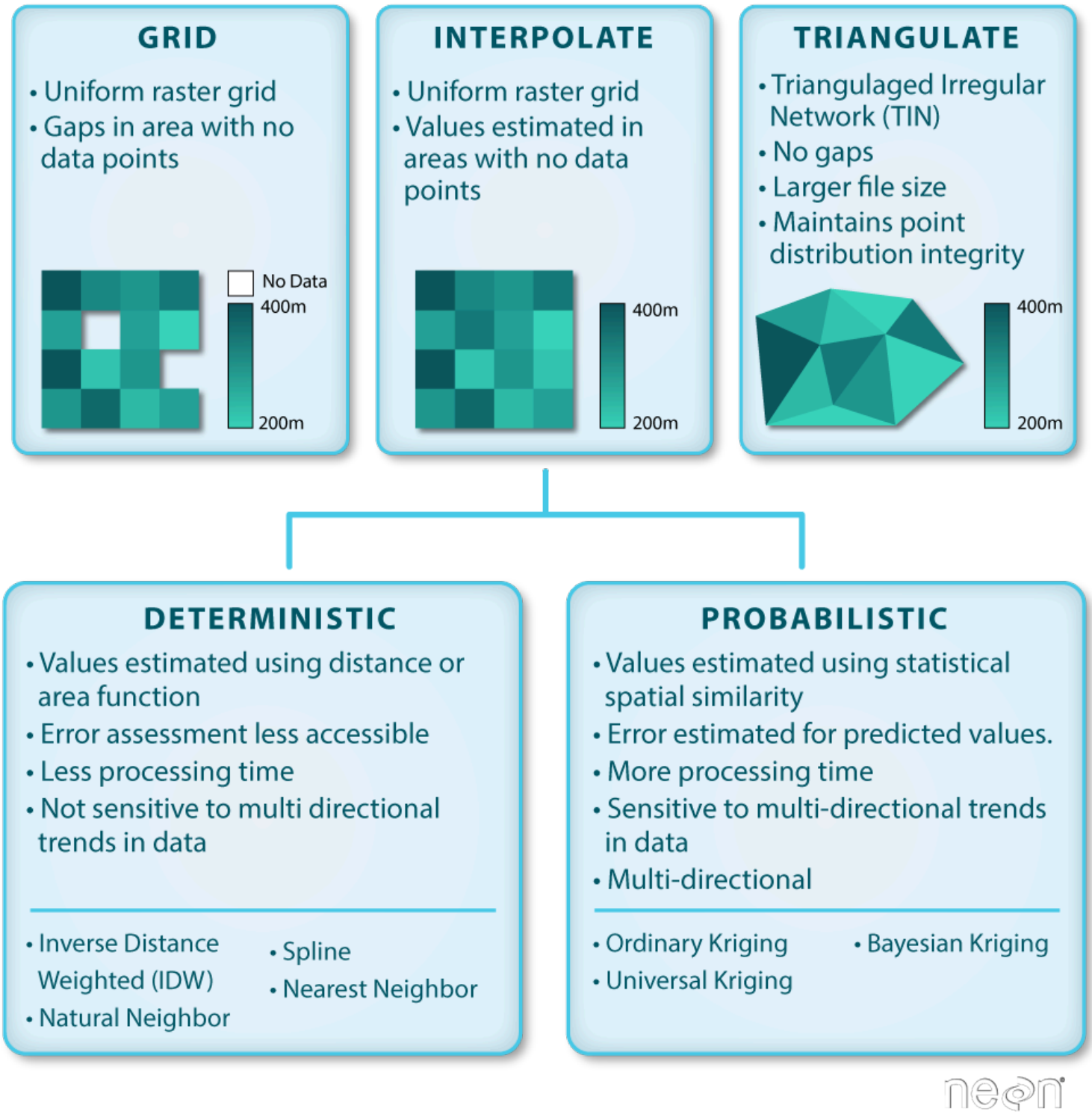
```
ubc_height_map = keplergl.KeplerGl(height=500,config=ubc_config)
ubc_height_map.add_data(data=ubc_bldg_heights.copy(), name="Building heights")

# # # render keplergl in jupyter lab
# orig_html = str(ubc_height_map._repr_html_(),'utf-8')
# b64d_html = base64.b64encode(orig_html.encode('utf-8')).decode('utf-8')
# framed_html = f'<iframe src="data:text/html;base64,{b64d_html}" style="width:95%; he
# IPython.display.HTML(framed_html)
```

```
User Guide: https://docs.kepler.gl/docs/keplergl-jupyter
```

- You can save your configuration and customization for re-use later (see the docs here)

2. Spatial modelling

- There's typically two main ways we might want to "model" spatial data:
    1. Spatial interpolation: use a set of observations in space to estimate the value of a spatial field
    2. Areal interpolation: project data from one set of polygons to another set of polygons
- Both are based on the fundamental premise: "everything is related to everything else, but near things are more related than distant things." (Tobler's first law of geography)



Source: https://www.neonscience.org/resources/learning-hub/tutorials/spatial-interpolation-basics

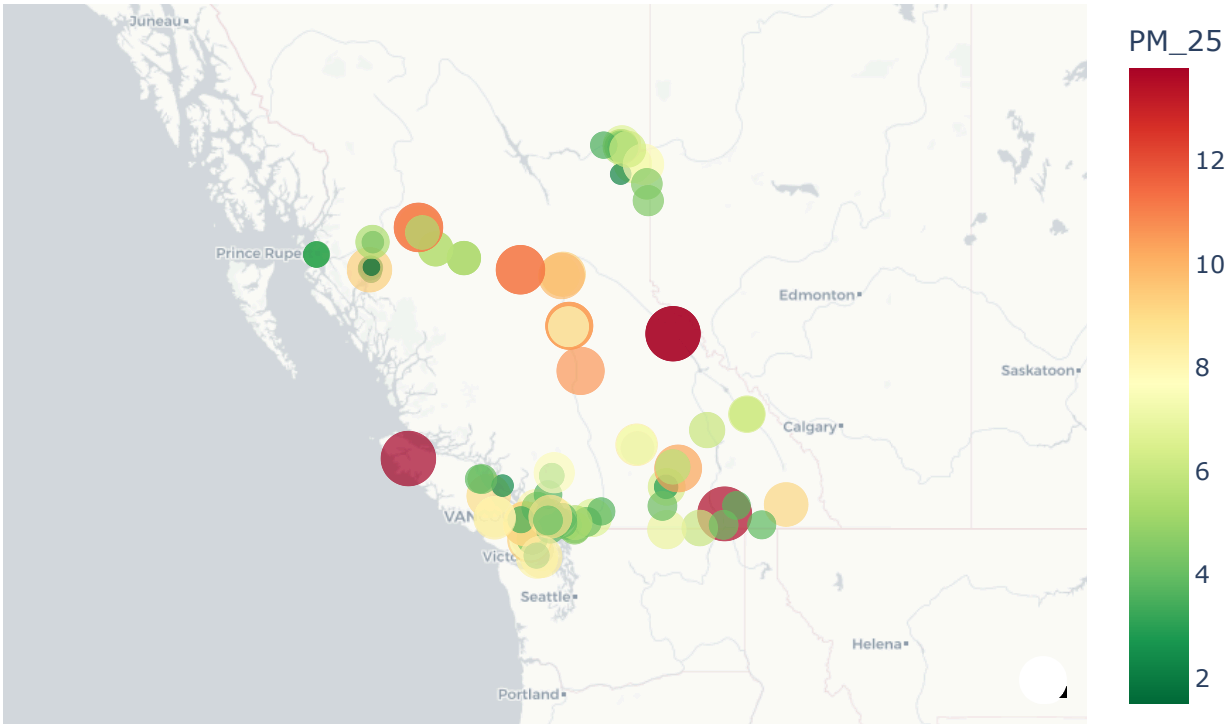
- Both are based on the fundamental premise: "*everything is related to everything else, but near things are more related than distant things.*" (Tobler's first law of geography)
- To demonstrate some of these methods, we'll look at the annual average air pollution (PM 2.5) recorded at stations across BC during 2020 (which I downloaded from DataBC):

```
pm25 = pd.read_csv("data/bc-pm25.csv")
pm25.head()
```

	Station Name	Lat	Lon	EMS ID	PM_25
0	Abbotsford A Columbia Street	49.021500	-122.326600	E289309	4.7
1	Abbotsford Central	49.042800	-122.309700	E238212	4.6
2	Agassiz Municipal Hall	49.238032	-121.762334	E293810	6.7
3	Burnaby South	49.215300	-122.985600	E207418	3.8
4	Burns Lake Fire Centre	54.230700	-125.764300	E225267	5.4

- The goal is to interpolate these point measurements to estimate the air pollution for all of BC

```
fig = px.scatter_mapbox(pm25, lat="Lat", lon="Lon", color="PM_25", size="PM_25",
                        color_continuous_scale="RdYlGn_r",
                        center={"lat": 52.261, "lon": -123.246}, zoom=3.5,
                        mapbox_style="carto-positron", hover_name="Station Name")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
fig.show()
```



2.1. Deterministic spatial interpolation

- Create surfaces directly from measured points using a mathematical function
- Common techniques (all available in the `scipy` module `interpolate`) are:
  - Inverse distance weighted interpolation
  - Nearest neighbour interpolation
  - Polynomial interpolation
  - Radial basis function (RBF) interpolation
- Let's try nearest neighbour interpolation now using `scipy.interpolate.NearestNDInterpolator`
- As angular coordinates (lat/lon) are not the best for measuring distances, I'm going to first convert my data to the linear, meter-based Lambert projection recommend by Statistics Canada and extract the `x` and `y` locations as columns in my `GeoDataFrame` ("Easting" and "Northing" respectively):

```
gpm25 = (gpd.GeoDataFrame(
            pm25,
            crs="EPSG:4326", # specify the CRS
            geometry=gpd.points_from_xy(pm25["Lon"], pm25["Lat"])) # create geometry f
        .to_crs("EPSG:3347") # convert to a different CRS
        )
gpm25["Easting"], gpm25["Northing"] = gpm25.geometry.x, gpm25.geometry.y
gpm25.head()
```

	Station Name	Lat	Lon	EMS ID	PM_25	geometry	Easting
0	Abbotsford A Columbia Street	49.021500	-122.326600	E289309	4.7	POINT (4056510.813 1954658.509)	4.056511e+06
1	Abbotsford Central	49.042800	-122.309700	E238212	4.6	POINT (4058698.635 1956191.479)	4.058699e+06
2	Agassiz Municipal Hall	49.238032	-121.762334	E293810	6.7	POINT (4104120.445 1957264.772)	4.104120e+06
3	Burnaby South	49.215300	-122.985600	E207418	3.8	POINT (4023977.036 1996103.398)	4.023977e+06

	Station Name	Lat	Lon	EMS ID	PM_25	geometry	Easting
4	Burns Lake Fire Centre	54.230700	-125.764300	E225267	5.4	POINT (4128403.079 2571148.393)	4.128403e+06

- Now let's create a grid of values (a raster) to interpolate over
- I'm just going to make a square grid of fixed resolution that spans the bounds of my observed data points (we'll plot this shortly so you can see what it looks like):

```
resolution = 2500  # cell size in meters, smaller cell size = smaller pixel = higher r
gridx = np.arange(gpm25.bounds.minx.min(), gpm25.bounds.maxx.max(), resolution)
gridy = np.arange(gpm25.bounds.miny.min(), gpm25.bounds.maxy.max(), resolution)
```

```
gpm25.bounds
```

	minx	miny	maxx	maxy
0	4.056511e+06	1.954659e+06	4.056511e+06	1.954659e+06
1	4.058699e+06	1.956191e+06	4.058699e+06	1.956191e+06
2	4.104120e+06	1.957265e+06	4.104120e+06	1.957265e+06
3	4.023977e+06	1.996103e+06	4.023977e+06	1.996103e+06
4	4.128403e+06	2.571148e+06	4.128403e+06	2.571148e+06
...	...	...	...	...
101	4.314675e+06	1.981401e+06	4.314675e+06	1.981401e+06
102	3.958860e+06	1.933369e+06	3.958860e+06	1.933369e+06
103	4.066041e+06	2.083101e+06	4.066041e+06	2.083101e+06
104	4.073994e+06	2.086230e+06	4.073994e+06	2.086230e+06
105	4.226070e+06	2.254141e+06	4.226070e+06	2.254141e+06

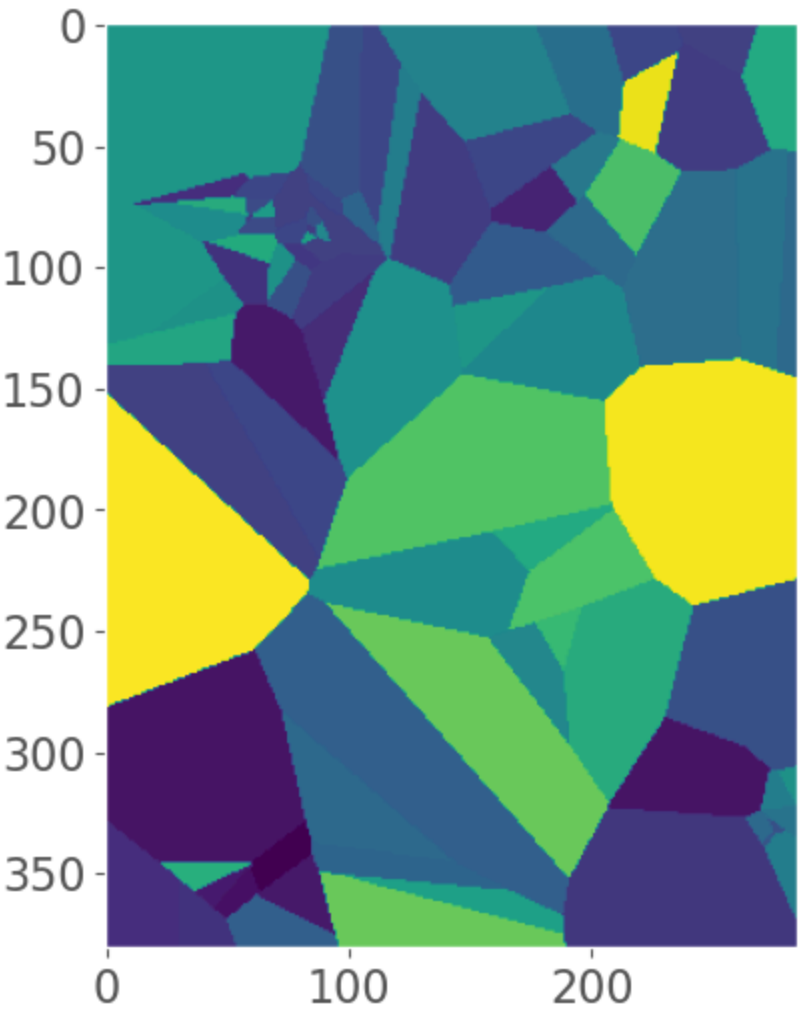
106 rows × 4 columns

```
gpm25.bounds.minx.min()
```

```
3811693.0769138555
```

- So now let's interpolate using a nearest neighbour method (the code below is straight from the scipy docs):

```
model = NearestNDInterpolator(x = list(zip(gpm25["Easting"], gpm25["Northing"])),
                              y = gpm25["PM_25"])
z = model(*np.meshgrid(gridx, gridy))
plt.imshow(z);
```

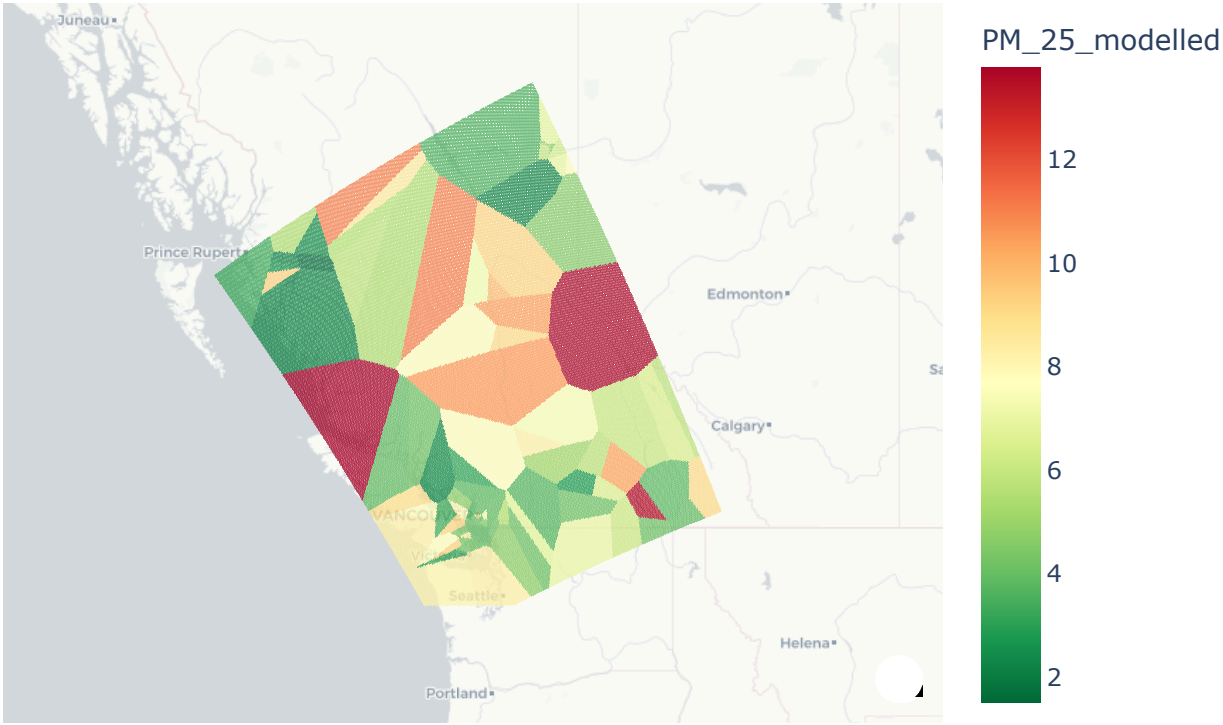


- Okay so it looks like we successfully interpolated our made-up grid, but let's re-project it back to our original map
- To do this I need to convert each cell in my raster to a small polygon using a simple function I wrote called `pixel2poly()` which we imported at the beginning of the notebook:

```
polygons, values = pixel2poly(gridx, gridy, z, resolution)
```

- Now we can convert that to a `GeoDataFrame` and plot using `plotly`:

```
pm25_model = (gpd.GeoDataFrame({"PM_25_modelled": values}, geometry=polygons, crs="EPS
                .to_crs("EPSG:4326")
              )

fig = px.choropleth_mapbox(pm25_model, geojson=pm25_model.geometry, locations=pm25_moc
                           color="PM_25_modelled", color_continuous_scale="RdYlGn_r",
                           center={"lat": 52.261, "lon": -123.246}, zoom=3.5,
                           mapbox_style="carto-positron")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
fig.update_traces(marker_line_width=0)
```



- Very neat!
- Also, notice how the grid distorts a bit once we project it back onto an angular (degrees-based) projection from our linear (meter-based) projection

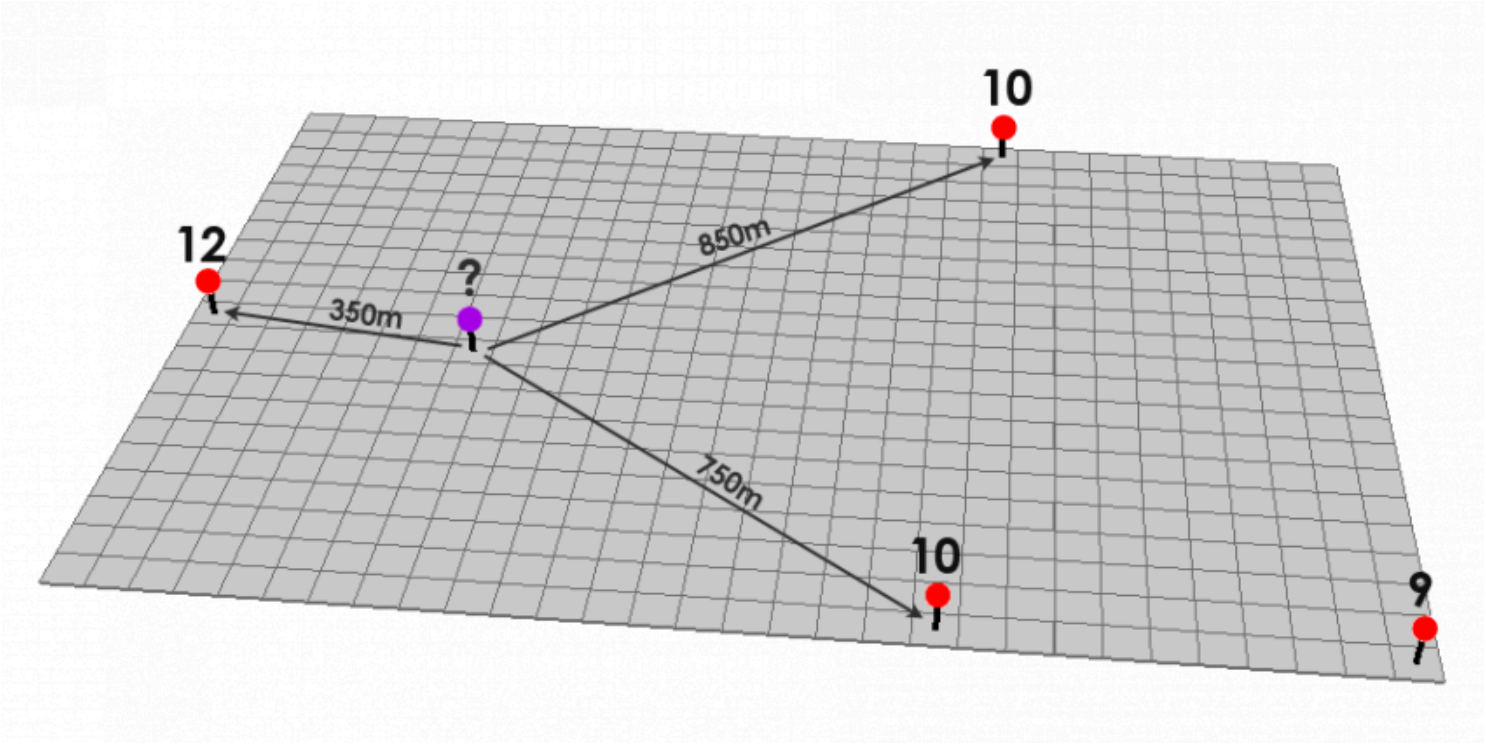
- Inversed disance weighted interpolation (IDW) is another popular method
- Weights are proportional to the inverse of the distance (between the data point and the prediction location) raised to the power value p

$$z_p = \frac{\sum_{i=1}^{n} \left(\frac{z_i}{d_i^p} \right)}{\sum_{i=1}^{n} \left(\frac{1}{d_i^p} \right)}$$

where

- z is the value
- d is the distance
- p is the power

Let's take an example



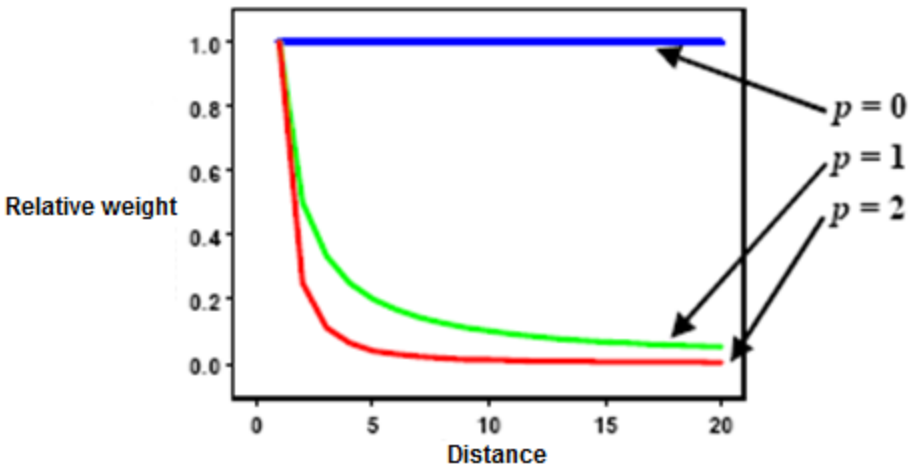
Here's what the table looks like for these 3 distances and values:

Distance	Value
350m	12
750m	10

Distance	Value
850m	10

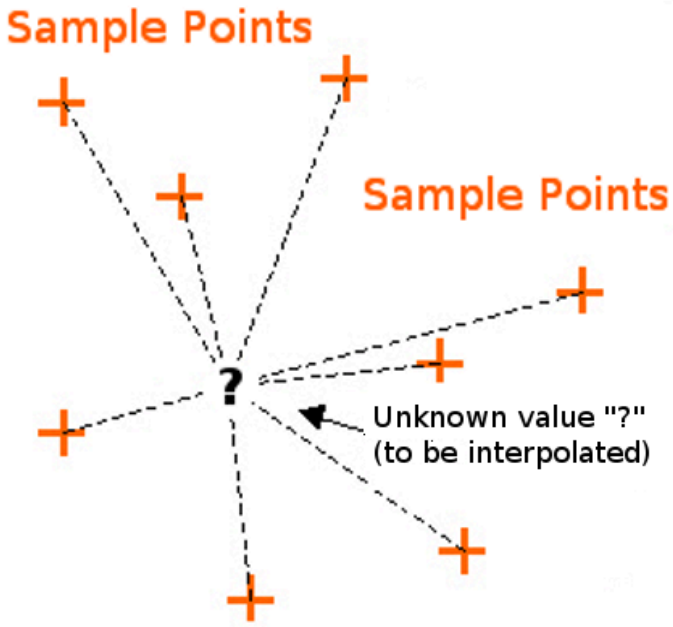
For $p = 1$, that cell value is equal to:

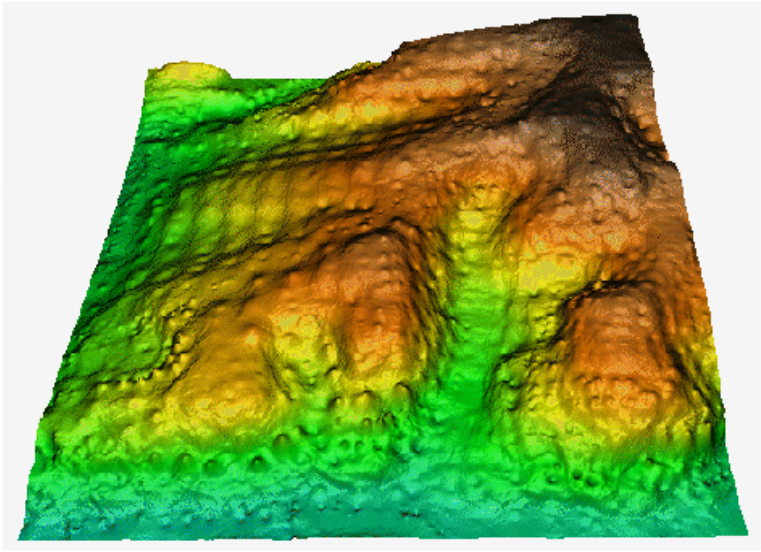
$$((12/350) + (10/750) + (10/850))/((1/350) + (1/750) + (1/850)) = 11.1$$



- Lower p, weights are distributed more equally
- Higher p, place more weights on closer observations
- Does it ring a bell? Does it sound similar to the α parameter in simple exponential smoothing?

2.2. Probabilistic (geostatistical) spatial interpolation



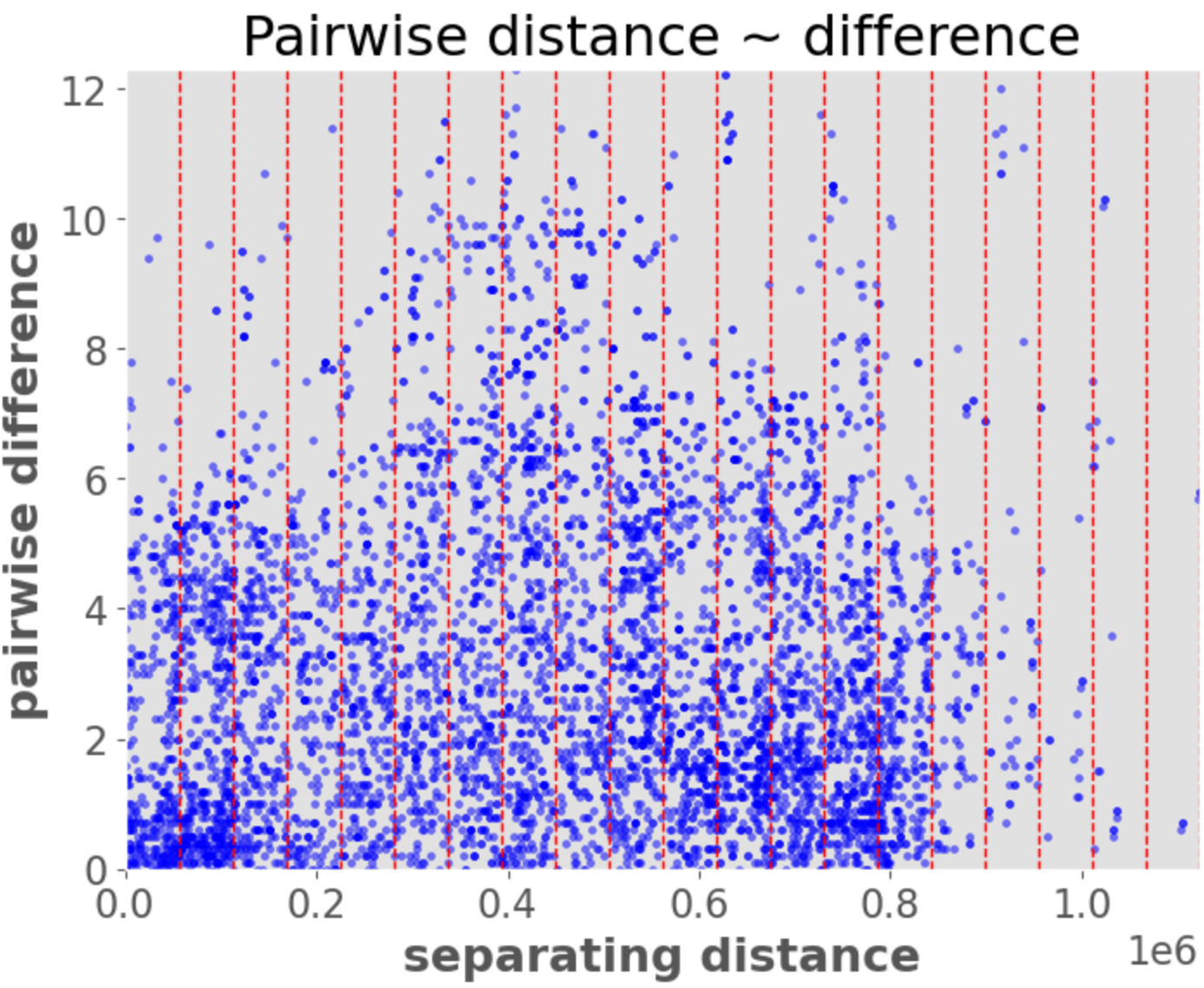


Source: https://www.neonscience.org/resources/learning-hub/tutorials/spatial-interpolation-basics

- Geostatistical interpolation (called "Kriging") differs to deterministic interpolation in that we interpolate using statistical models that include estimates of spatial autocorrelation
- There's a lot to read about kriging, I just want you to be aware of the concept in case you need to do something like this in the future. Usually we'd do this kind of interpolation in GIS software like ArcGIS or QGIS, but it's possible to do in Python too.
- The basic idea is that if we have a set of observations $Z(s)$ at locations s, we estimate the value of an unobserved location (s_0) as a weighted sum: $\hat{Z}(s_0) = \sum_{i=0}^{N} \lambda_i Z(s_i)$
- Where N is the size of s (number of observed samples) and λ is an array of weights.

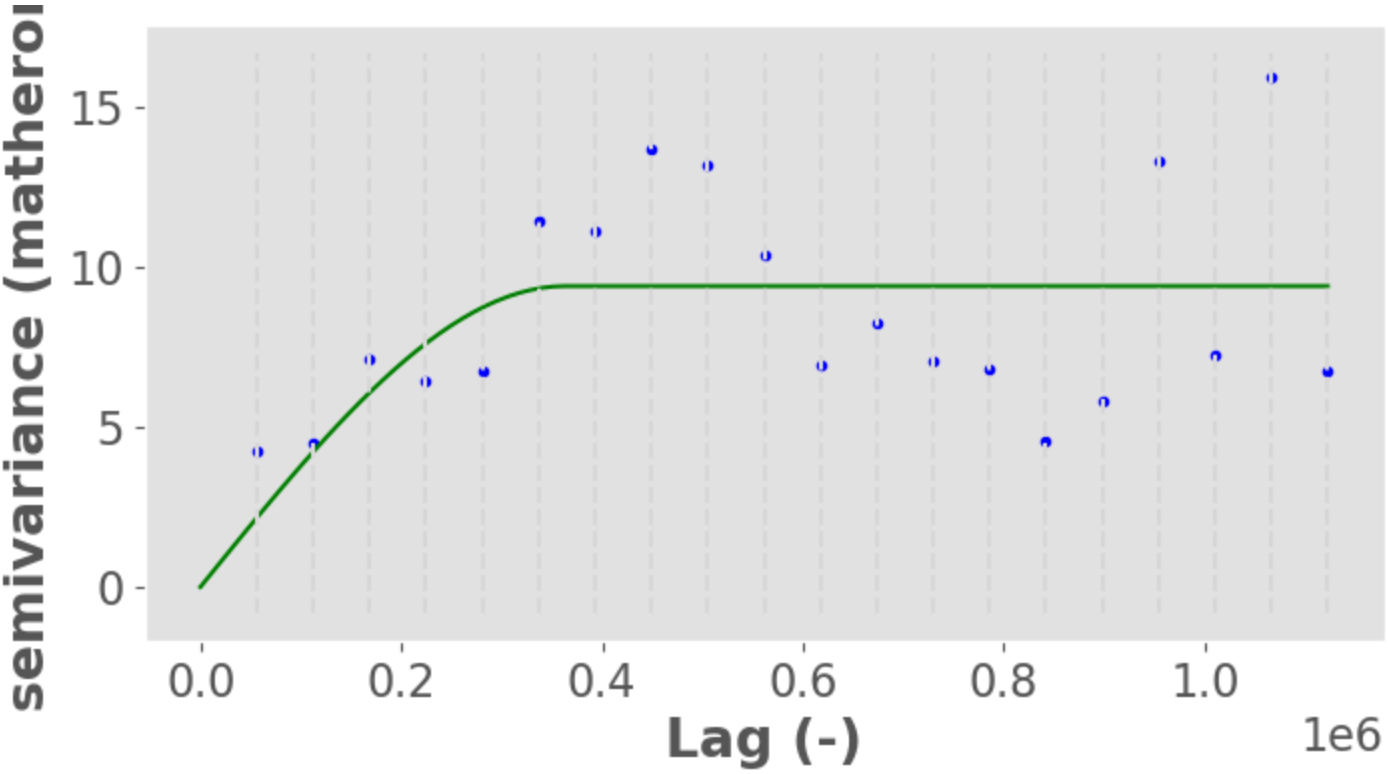
- The key is deciding which weights λ to use
- Kriging uses the spatial autocorrelation in the data to determine the weights
- Spatial autocorrelation is calculated by looking at the squared difference (the variance) between points at similar distances apart, let's see what that means:

```
warnings.filterwarnings("ignore")  # Silence some warnings
vario = Variogram(coordinates=gpm25[["Easting", "Northing"]],
                  values=gpm25["PM_25"],
                  n_lags=20)
vario.distance_difference_plot();
```



- The idea is to then fit a model to this data that describes how variance (spatial autocorrelation) changes with distance ("lag") between locations
- We look at the average variance in bins of the above distances/pairs and fit a line through them
- This model is called a "variogram" and it's analogous to the autocorrelation function for time series. It is similar as ACF plot for time series, but instead of time, it has distance on the x-axis.
- It defines the variance (autocorrelation structure) as a function of distance:

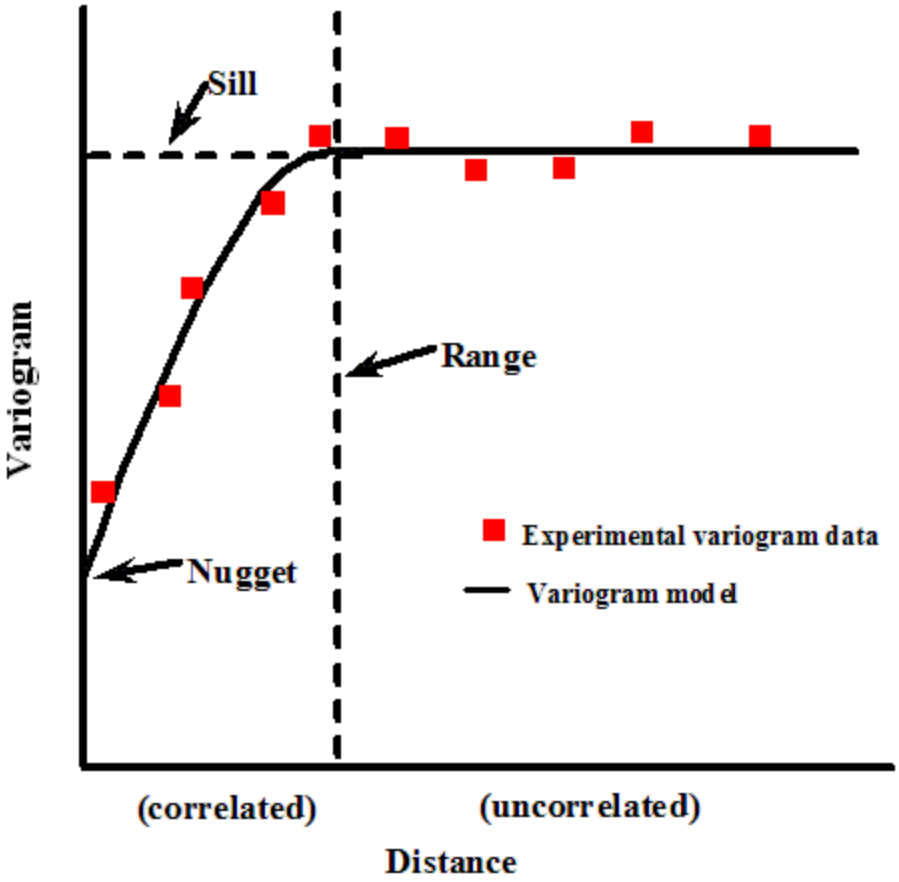
```
vario.plot(hist=False);
```



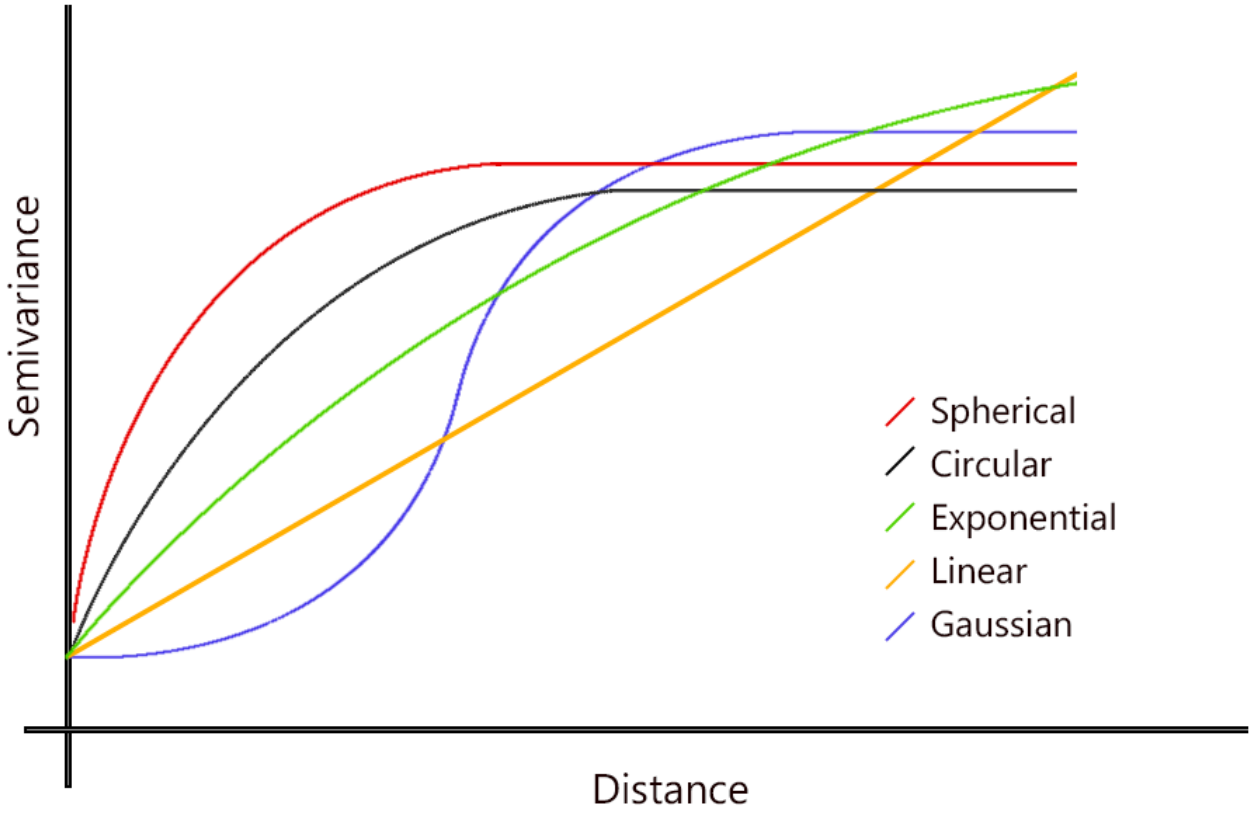
- The above plot basically shows that:
  - at small distances (points are close together), variance is reduced because points are correlated
  - but at a distance around 400,000 m the variance flattens out indicating points are two far away to have any impactful spatial autocorrelation. This location is called the “range” while the variance at the “range” is called the “sill” (like a ceiling). We can extract the exact range:

```
vario.describe()["effective_range"]
```

```
363018.9096524524
```



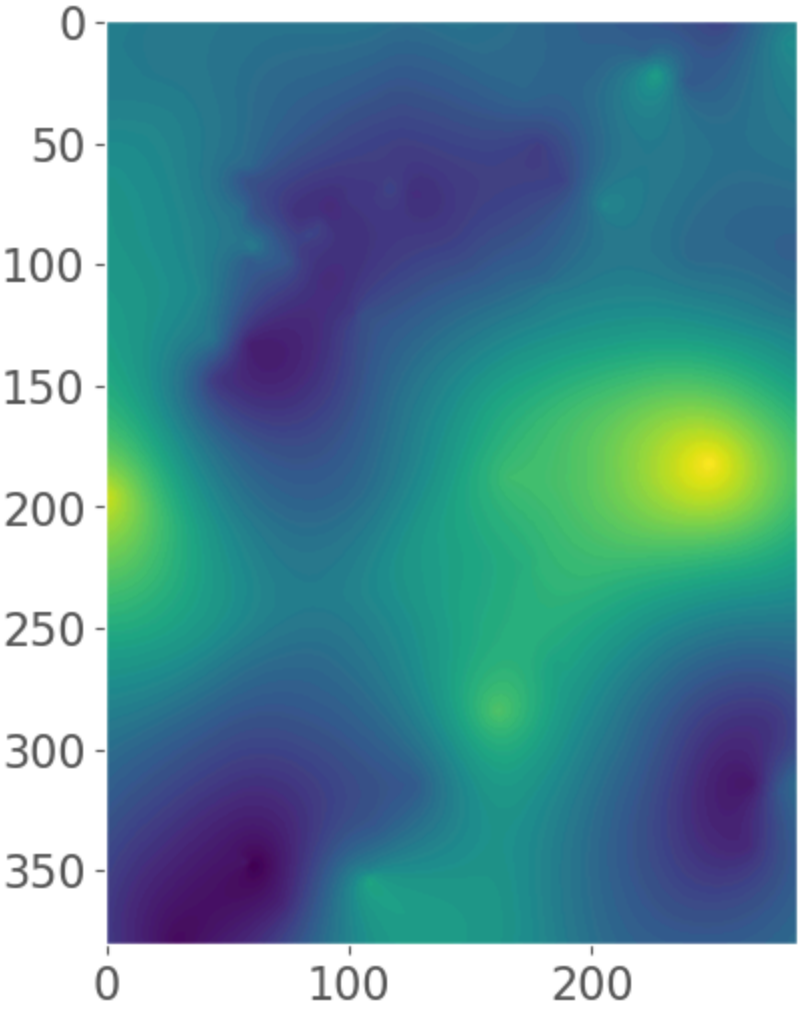
- A **nugget** is a parameter in a variogram that represents the variance or variability of a spatial process at a distance of zero. Think of it as random error or measurement error
- A **sill**, on the other hand, is the maximum variance or variability of the spatial process being modeled. It represents the amount of spatial autocorrelation that exists between observations at a distance that is large enough to capture the underlying trend of the process.



- By the way, we call it the semi-variance because there is a factor of 0.5 in the equation to account for the fact that variance is calculated twice for each pair of points (read more here)
- Remember, our variogram defines the spatial autocorrelation of the data (i.e., how the locations in our region affect one another)
- Once we have it, we can use it to estimate the weights in our kriging model. I won't go into detail on how this is done.

- You can think of the variogram as the ACF in time series modelling. It defines our autocorrelation. We might use it to pick orders for an autoregressive model which we then fit to some data. Here, the variogram defines spatial autocorrelation, which we can then use to fit a kriging model
- I'll use the pykrige library to do this for us
- First, I'll make a higher resolution grid because it will look cooler!

```
krig = OrdinaryKriging(x=gpm25["Easting"], y=gpm25["Northing"], z=gpm25["PM_25"], vari
z, ss = krig.execute("grid", gridx, gridy)
plt.imshow(z);
```



- Now let's convert our raster back to polygons so we can map it
- I'm also going to load in a polygon of BC using `osmnx` to clip my data so it fits nicely on my map this time:

```
polygons, values = pixel2poly(gridx, gridy, z, resolution)
pm25_model = (gpd.GeoDataFrame({"PM_25_modelled": values}, geometry=polygons, crs="EPS
                  .to_crs("EPSG:4326")
                  )
bc = ox.geocode_to_gdf("British Columbia, Canada")
pm25_model = gpd.clip(pm25_model, bc)
```

```
fig = px.choropleth_mapbox(pm25_model, geojson=pm25_model.geometry, locations=pm25_moc
                           color="PM_25_modelled", color_continuous_scale="RdYlGn_r",
                           center={"lat": 52.261, "lon": -123.246}, zoom=3.5,
                           mapbox_style="carto-positron")
fig.update_layout(margin=dict(l=0, r=0, t=30, b=10))
fig.update_traces(marker_line_width=0)
```

- I used an "ordinary kriging" interpolation above which is the simplest implementation of kriging and I show how it's derived in Appendix C if you're interested
- The are many other forms of kriging too that can account for underlying trends in the data ("universal kriging"), or even use a regression or classification model to make use of additional explanatory variables. `pykrige` supports most variations. In particular for the latter, `pykrige` can accept `sklearn` models which is useful!

2.3. Areal interpolation

- Areal interpolation is concerned with mapping data from one polygonal representation to another
- Imagine I want to map the air pollution polygons I just made to FSA polygons (recall FSA is "forward sortation area", which are groups of postal codes).
- The most intuitive way to do this is to distribute values based on area proportions, hence "areal interpolation"
- I'll use the tobler library for this
- First, load in the FSA polygons:

```
van_fsa = gpd.read_file("data-spatial/van-fsa")
ax = van_fsa.plot(edgecolor="0.2")
plt.title("Vancouver FSA");
```

- Now I'm just going to made a higher resolution inteprolation so we can see some of the details on an FSA scale:

```
resolution = 10_000  # cell size in meters
gridx = np.arange(gpm25.bounds.minx.min(), gpm25.bounds.maxx.max(), resolution)
gridy = np.arange(gpm25.bounds.miny.min(), gpm25.bounds.maxy.max(), resolution)
krig = OrdinaryKriging(x=gpm25["Easting"], y=gpm25["Northing"], z=gpm25["PM_25"], vari
z, ss = krig.execute("grid", gridx, gridy)
polygons, values = pixel2poly(gridx, gridy, z, resolution)
pm25_model = (gpd.GeoDataFrame({"PM_25_modelled": values}, geometry=polygons, crs="EPS
                .to_crs("EPSG:4326")
                )
```

```
z, ss = krig.execute("grid", gridx, gridy)
plt.imshow(z);
```

- Now we can easily do the areal interpolation using the function `area_interpolate()`:

```
areal_interp = area_interpolate(pm25_model.to_crs("EPSG:3347"),
                                van_fsa.to_crs("EPSG:3347"),
                                intensive_variables=["PM_25_modelled"]).to_crs("EPSG:4
areal_interp.plot(column="PM_25_modelled", figsize=(8, 8),
                  edgecolor="0.2", cmap="RdBu", legend=True)
plt.title("FSA Air Pollution");
```

- There are other methods you can use for areal interpolation too, that include additional variables or use more advanced interpolation algorithms. The tobbler documentation describes some of these

3. Shortest path analysis

Load a walking network in UBC using the function `ox.graph_from_place()`. You can specify the network type using the `network_type` argument. Possible options are “all_private”, “all”, “bike”, “drive”, “drive_service”, “walk”G = ox.graph_from_place(“UBC, Vancouver”, network_type=”walk”) G

```
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

G = ox.graph_from_place("UBC, Vancouver",
                                  network_type="walk")
G
```

```
# Plot the graph with OSMnx
ox.graph_to_gdfs(G, nodes=False).explore()
```

Create origin and destination

```
# Origin
orig_address = "UBC bookstore, Vancouver"
orig_y, orig_x = ox.geocode(orig_address)  # notice the coordinate order (y, x)!

# Destination
dest_address = "Orchard Commons Student Residence, Vancouver"
dest_y, dest_x = ox.geocode(dest_address)

print("Origin coords:", orig_x, orig_y)
print("Destination coords:", dest_x, dest_y)
```

```
fig, ax = ox.plot_graph(G, show=False, close=False)
ax.scatter(orig_x, orig_y, c='red')
ax.scatter(dest_x, dest_y, c='red')
plt.show()
```

Find the closest nodes for origin and destination

```
orig_node_id, dist_to_orig = ox.distance.nearest_nodes(G, X=orig_x, Y=orig_y, return_c
dest_node_id, dist_to_dest = ox.distance.nearest_nodes(G, X=dest_x, Y=dest_y, return_c

print("Origin node-id:", orig_node_id, "and distance:", dist_to_orig, "meters.")
print("Destination node-id:", dest_node_id, "and distance:", dist_to_dest, "meters.")
```

Find the the shortest path by length using `nx.shortest_path()`. The default method for computing the shortest path in nx.shortest_path is the `dijkstra` algorithm. If you are curious about how it works, here's a good video that explains the algorithm:

```
from IPython.display import HTML
# Youtube
HTML('<iframe width="560" height="315" src="https://www.youtube.com/embed/EFg3u_E6eHU"
```

To summarize the video, these are the main steps of Dijkstra's algorithm:

- Start with a chosen town (Town A), assign it a distance of 0, and mark it as visited.
- Choose the unvisited town with the smallest distance.
- Iterate through unvisited towns (Towns B-F), updating their distances as you go with the shortest path.
- Continue until all towns are visited or the destination is reached.
- Choose the path from A to B with the smallest distance.

```
# Find shortest path (by distance)
import networkx as nx
route = nx.shortest_path(G, orig_node_id, dest_node_id, weight="length")

# Plot the shortest path
# ox.plot_route_folium(G, route, weight=3)
ox.plot.plot_graph_route(G, route)
```

4. Goodbye!

- This is our last lecture together. 😢 What a journey it's been!
- Take a moment after this lecture to reflect on everything you've achieved, how much you've learned, and how much work you've put in. You are amazing!
- But this isn't really the final goodbye - I'm looking forward to watching you apply all the data science skills we've learnt to some super amazing capstone projects.
- So, until next time!