# Mercury concentrations in biota in Mediterranean Sea – interpolation with Inverse Distance Weighting technique and Python

Environmental protection is the topic which has taken a lot from spatial interpolation techniques. Imagine situation when you are analyzing soil near some open pit mine. It’s extremely expensive to measure each square meter of area if there are thousands of meters to analyze, so you sample only one percent of area and that should be enough to model whole area for contaminant’s concentration. You can interpolate missing values with various techniques, but in this lesson we implement Inverse Distance Weighting (IDW) algorithm.

We use IDW for interpolation of mercury concentrations near the coast of the Mediterranean Sea.

Our task is to:

* Implement Inverse Distance Weighting algorithm in Python and numpy,
* Transform spatial data for modeling purposes.

**Data source**

Dataset in this article comes from:

C: Cinnirella, Sergio; Bruno, Delia Evelina; Pirrone, Nicola; Horvat, Milena; Živković, Igor; Evers, David; Johnson, Sarah; Sunderland, Elsie (2019): Mercury concentrations in biota in the Mediterranean Sea, a compilation of 40 years of surveys. PANGAEA, <https://doi.org/10.1594/PANGAEA.899723>

And I’ve found this dataset and article by the GEOSS search engine:

C: Geoportal: <https://www.geoportal.org>

You may check article notebooks and data here:

C: Szymon’s Github: <https://github.com/szymon-datalions/geoprocessing/tree/master/data%20analysis/notebooks/idw_01>

**Theoretical background**

Mercury is a neurotoxin. It accumulates in sea organisms and indirectly in our bodies as a methylmercury (MeHg). U.S. EPA guidelines tells us that methylmercury in high concentrations:

* affects our sight (loss of peripheral vision),
* affects our muscles (weakness of muscle, impaired coordination, problems with speech, hearing and walking).
* changes our neural responses and creates anxious sensation of “pins and needles”, usually in the hands, feet, and around the mouth.
* And it is very dangerous for unborn infants’ brains and nervous systems [1].

There are some publications that MeHg may cause tumors, but there are others which shows no significant effects of exposure to MeHg and cancer development so we should be cautious with conclusions on this topic; however the central nervous system damage risk for adults and extremely high threat for fetus brain and neural development is enough to monitor mercury concentrations in sea organisms which, for some world populations, are the main source of food.

**Mercury concentrations in sea organisms in Mediterranean Sea: data modeling pipeline and programming environment preparation**

Data modeling pipeline for each data science problem is very similar. Figure 1 shows steps from programming environment preparation up to the model development. In this tutorial we cover the violet part of pipeline. I expect that you can set up Anaconda and it’s environment and install necessary Python packages. (If I’m wrong please tell mi in a comment section, then I’ll prepare short article how to set up environment and install Anaconda in MacOS and/or Linux systems).

Obraz zawierający zrzut ekranu

Opis wygenerowany automatycznie

Our programming environment has five Python packages:

* numpy for scientific computation,
* pandas for tabular data processing and analysis,
* geopandas for spatial data processing and analysis,
* matplotlib for data visualization,
* seaborn for data visualization.

Install them from the conda-forge channel with Python version from 3.6 above. Then download data from publication (csv file) <https://doi.org/10.1594/PANGAEA.899723> along with Mediterranean coastline shapefile from my Github: [LINK TO GITHUB]. We are ready to go!

**Data exploration**

First thing first. Before we start modeling it is a good idea to look into a dataset. In our case it is a quick look. We read csv file with pandas, slice it and store as a new csv file for model development. We do not need all information provided by authors for our purposes.

We import pandas and read csv file (Listing 1).

>> import pandas as pd

>> df = pd.read\_csv('190516\_m2b\_marine\_biota.csv', sep=';', encoding = "ISO-8859-1", index\_col='id')

>> df.head()

As you see, our dataframe has 36 columns. What columns are avaliable? And how many rows has our dataset? Let’s check it (Listing 2).

>> print('Columns in dataframe:')

>> for idx, col in enumerate(df.columns):

>> print(idx+1, col)

>> print('Dataframe length:', len(df))

From those columns we are interested in:

* lat (latitude),
* lon (longitude),
* mea\_ug\_kg\_orig (mean Hg concentrations in a sampled organism).

But still, we have more than 24 thousand rows of data... It is a good idea to take a part of DataFrame for a model development. Do not use entire dataset when you are building model from scratch. Why? Two main reasons are:

1. New code is (usually) not optimized from the computational perspective and if you test its output and logic you need to do it as fast as possible,
2. Sometimes dataset is flawed in some way and it is hard to find it out, you may think that your model is wrong but dataset may have erroneous records which distort modeling algorithm.

How do we divide our dataset? We can do it by the date of sampling and limit records to those sampled in 21st century. The icing on the cake here is that measurements taken in the less distant past from today should be considered as closer to the current state of Hg concentrations and more important from the policy-making perspective. Listing 3 shows process of data division by time with pandas time-indexing. In the first step we create new column with sampling dates formatted as datetime pandas object. Next we check how many records are available after 2000 year and in the last step we create new DataFrame with limited number of records.

>> df['samp\_date'] = pd.to\_datetime(df['samp\_date'], format='%Y-%M-%d')

>> print(len(df['samp\_date'][df['samp\_date'] > '2000-01-01']))

>> updated\_df = df[df['samp\_date'] > '2000-01-01']

Before we export this new DataFrame to the csv file we will slice it we leave only lat, lon and mea\_ug\_kg\_orig columns. Then we save our final DataFrame to the csv (Listing 4).

>> ndf = updated\_df[['lat', 'lon', 'mea\_ug\_kg\_orig']]

>> ndf.to\_csv('prepared\_data\_mercury\_concentrations.csv', encoding='utf-8', sep=',')

Well done! Data is prepared, time for modeling!

**IDW model**

*Point data visualization*

Open new Jupyter Notebook file and import: numpy, pandas, geopandas, matplotlib.pyplot and seaborn. Read prepared dataframe and limit it to the records with zero and positive value of mean concentration of mercury (Listing 5).

>> import numpy as np

>> import pandas as pd

>> import geopandas as gpd

>> import matplotlib.pyplot as plt

>> import seaborn as sns

>>

>> df = pd.read\_csv('prepared\_data\_mercury\_concentrations.csv', index\_col='id')

>> df = df[df['mea\_ug\_kg\_orig'] >= 0]

We use seaborn’s scatterplot() to plot recorded samples and their respective values. Scatterplot takes many arguments, but we use:

* x – longitude coordinates of samples,
* y – latitude coordinates of samples,
* hue – mean mercury concentrations (scatterplot’s points are colored due to the hue),
* hue\_norm – custom hue limits of numerical data, we use this to emphasize places where mean mercury concentrations exceed 460 micrograms per kilogram in a fish tissue [2],
* alpha – parameter for transparency control for better visual presentation (0.7),
* data – baseline DataFrame.

A code is presented in the Listing 6 and output image is presented in the Figure 2.

>> epa\_norm = 460

>> hue\_norm\_custom = (0, epa\_norm)

>> plt.figure(figsize=(12, 6))

>> sns.scatterplot(x='lon', y='lat', hue='mea\_ug\_kg\_orig',

hue\_norm=hue\_norm\_custom, alpha=.7, data=df)

>> plt.show()

Obraz zawierający mapa

Opis wygenerowany automatycznie

You may observe that mercury concentrations near Italy’s coastline exceeds EPA limits and the most measurements are taken near Italy. We shouldn’t conclude anything from it, but it is a good starting point for further analysis.

*Shapefile preparation*

We need canvas to perform interpolation. Or, in other words, a set of points for which we interpolate values. It can be a numpy array, but in our geospatial world, vector files are more useful, and we use shapefile with points along Mediterranean Sea coastline (without borders of isles). File medi\_coastline.shp (from here: <https://github.com/szymon-datalions/geoprocessing/tree/master/data%20analysis/notebooks/idw_01>) contains prepared points, let’s read it with geopandas and look into a spatial DataFrame. Listing 7 is a read operation of GeoPandas and print of GeoDataFrame object and its coordinate reference system. The main difference between GeoDataFrame and DataFrame is that the first one has additional geometry column and the coordinate reference system (crs) property.

Listing 7:

>> gdf = gpd.read\_file('medi\_coastline.shp')

>> print(gdf.head())

>> print(gdf.crs)

Geometry in our GeoDataFrame is a MultiPoint object from shapely package. We can calculate distance between MultiPoint objects and our latitude / longitude pairs of float values in baseline DataFrame… but it will make distance calculation more complex. Instead of writing a single complicated function which is transforming MultiPoint into floats and calculating distance between those points we create separate function which transform geometry field MultiPoint objects into latitude / longitude floats. Listing 8 is a function implementation. Note that we get float point values by the .x and the .y Point and MultiPoint instance variables.

Listing 8

def set\_coordinate(point\_geometry, coordinate: str):

"""

Function returns coordinate x or y from multipoint geometry in geodataframe.

:param point\_geometry: MultiPoint or Point geometry,

:param coordinate: x (longitude) or y (latitude).

"""

if coordinate == 'x':

try:

x\_coo = point\_geometry.x

return x\_coo

except AttributeError:

x\_coo = point\_geometry[0].x

return x\_coo

elif coordinate == 'y':

try:

y\_coo = point\_geometry.y

return y\_coo

except AttributeError:

y\_coo = point\_geometry[0].y

return y\_coo

else:

raise KeyError('Available coordinates: "x" for longitude or "y" for latitude')

gdf['lat'] = gdf['geometry'].apply(set\_coordinate, args=('y'))

gdf['lon'] = gdf['geometry'].apply(set\_coordinate, args=('x'))

print(gdf.head())

Finally, we have spatial data prepared and we are able to move to the modeling step.

*IDW function*

Let’s recall IDW equation:

where:

* is a value at unknown location,
* is i-th known location,
* is a value at known location,
* is a weight assigned to the known location.

Weight between a points pair is inversely proportional to the distance to the p-th power:

where:

* is a distance from known point to the unknown point,
* is a hyperparameter which controls how strong is a relationship between known point and unknown point.

There is a *hidden* function, because distance between points must be calculated and the most general way to do so is to use Euclidean distance function:

where:

1. are coordinates of a point where we want to estimate value.

Then we have two functions. One is for distance calculation and second is for inverse distance weighting of unknown values. Distance calculation is really simple one and we can do it fast with Numpy array operations. Figure 3 shows block diagram of a function. Listing 9 is its implementation.

Obraz zawierający zegar

Opis wygenerowany automatycznie

You may see in the Figure 3 that we calculate power of differences separately for longitudes and latitudes (x and y respectively). We use Numpy operations so we can do it for whole array. Then we sum those two arrays and get square root of each element of summed array. That’s our distances.

Listing 9

def calculate\_distances(all\_points, unknown\_point):

# Calculate distances

d\_lat = (all\_points[:, 0] - unknown\_point[0])\*\*2

d\_lon = (all\_points[:, 1] - unknown\_point[1])\*\*2

dists = np.sqrt(d\_lon + d\_lat)

return dists

IDW model is trickier. We pass into function unknown point coordinates, known points’ coordinates and values, power of distance and number of closest distances to the unknown points which are affecting it. The latter is not included in the equation (1), but it is optimization for our calculations, and it allows us to control space of influence for each known point. It may be very unlikely from the physical perspective that measurements taken near Egyptian coast are correlated with measurements near Croatia. Figure 4 is a block diagram of IDW function. As you may notice it seems to be complex but in reality, it is a simple waterfall-style diagram.

Obraz zawierający mapa, tekst

Opis wygenerowany automatycznie

Figure 4: IDW algorithm diagram.

IDW function works as follow:

1. Calculate distance between unknown point and all known points.
2. Append calculated distances as a new column to the Numpy array with known distances.
3. Sort this array by the last column.
4. Take ndist number of records for analysis. It may be easily done with Numpy indexing.
5. Check if the first distance from a sorted list is equal to 0. If yes, then return point value from this position and end calculations. If no, then calculate weights.
6. Single weight is a ratio of 1 and a distance raised to the n-th power. Calculate weights for each element in the array.
7. Calculate numerator of equation (1). It is a product of weights and known points values.
8. Estimate interpolated point value as a ratio of a sum of all weight-value products and a sum of weights.

Listing 10 is an implementation of this process. Additionally, below function we are using it with a spatial GeoDataFrame which we have prepared earlier. We use lambda expression to do it, set power to 3 and ndist to 4.

def inverse\_distance\_weighting(unknown\_point, points, power, ndist=10):

"""

Function estimates values in unknown points with with inverse weighted interpolation technique.

INPUT:

:param unknown\_point: lat, lon coordinates of unknown point,

:param points: (array) list of points [lat, lon, val],

:param power: (float) constant used to calculate IDW weight -> weight = 1/(distance\*\*power),

:param ndist: (int) how many closest distances are included in weighting,

OUTPUT:

:return interpolated\_value: (float) interpolated value by IDW method.

Inverse distance weighted interpolation is:

est = SUM(WEIGHTS \* KNOWN VALS) / SUM(WEIGHTS)

and

WEIGHTS = 1 / (DISTANCE TO UNKNOWN\*\*power)

where:

power is a constant hyperparameter which tells how much point is influenced by other points.

"""

distances = calculate\_distances(points, unknown\_point)

points\_and\_dists = np.c\_[points, distances]

# Sort and get only 10 values

points\_and\_dists = points\_and\_dists[points\_and\_dists[:, -1].argsort()]

vals\_for\_idw = points\_and\_dists[:ndist, :]

# Check if first distance is 0

if vals\_for\_idw[0, -1] == 0:

return vals\_for\_idw[0, 2]

else:

# If its not perform calculations

weights = 1 / (vals\_for\_idw[:, -1]\*\*power)

numerator = weights \* vals\_for\_idw[:, 2]

interpolated\_value = np.sum(numerator) / np.sum(weights)

return interpolated\_value

# Interpolate points

known\_points\_array = df.to\_numpy()

power = 3

nd = 4

gdf['est'] = gdf.apply(lambda col: inverse\_distance\_weighting(

[col['lat'], col['lon']],

known\_points\_array, power, ndist=nd), axis=1)

Yes, that’s all! We can now check our interpolated data (Listing 11) and print it, or use it during a conference.

Listing 11

plt.figure(figsize=(15 ,9))

sns.scatterplot(x='lon', y='lat', hue='est', alpha=0.7, palette='coolwarm',

hue\_norm=hue\_norm\_custom, data=gdf)

plt.show()

You probably saw final output in the article header but here it is again:

Obraz zawierający mapa, tekst

Opis wygenerowany automatycznie

**Exercises:**

1. Use different shapefile for analysis provided here: <https://github.com/szymon-datalions/geoprocessing/tree/master/data%20analysis/notebooks/idw_01> named exercise\_points.shp. What do you think about the output?
2. Check what happens when power value is set between 0 and 1, or below 0.
3. Check what happens if you use all points for the unseen point value estimation.

**Next lesson:**

*Spatial and vertical interpolation and visualization of mercury concentrations in Mediterranean Sea with IDW and Python* (planned release date: 2020-06-23).

**Bibliography:**

[1] Health Effects of Exposures to Mercury. United States Environmental Protection Agency. Link: <https://www.epa.gov/mercury/health-effects-exposures-mercury>

[2] EPA-FDA Fish Advice: Technical Information. United States Environmental Protection Agency. Link: <https://www.epa.gov/fish-tech/epa-fda-fish-advice-technical-information>

[3] Github with article’s notebooks and shapefiles: <https://github.com/szymon-datalions/geoprocessing/tree/master/data%20analysis/notebooks/idw_01>