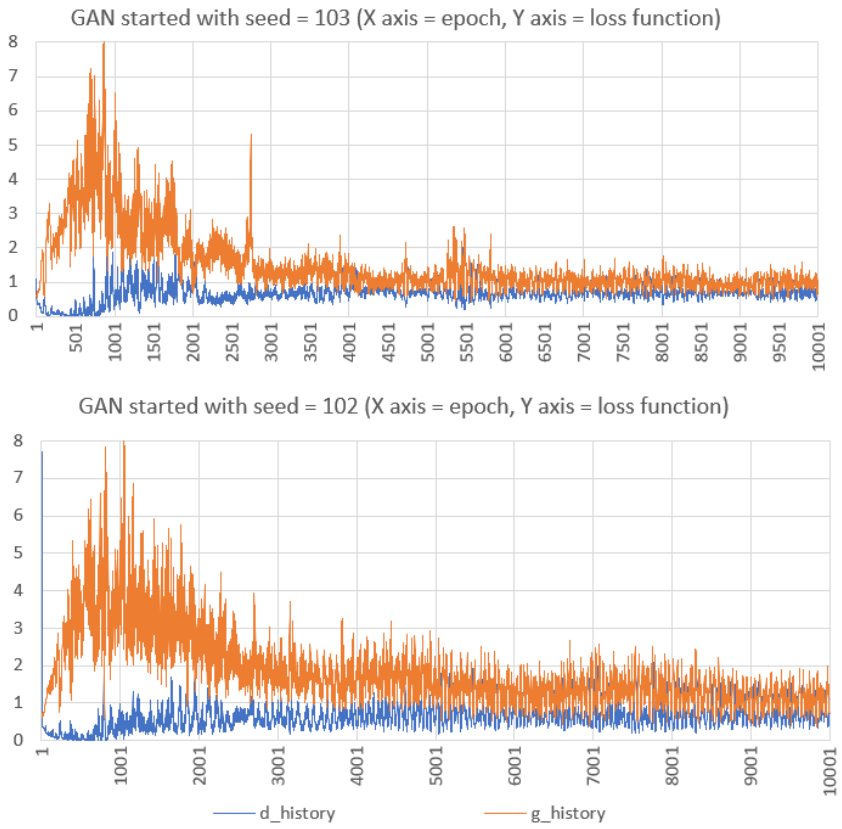
Practical AI & Machine Learning Projects and Datasets



Vincent Granville, Ph.D. | www.MLTechniques.com | Version 1.0, July 2023

Preface

This book is intended to participants in the AI and machine learning certification program organized by my AI/ML research lab MLtechniques.com. It is also an invaluable resource to instructors and professors teaching related material, and to their students. If you want to add serious, enterprise-grade projects to your curriculum, with deep technical dive on modern topics, you are welcome to re-use my projects in your classroom. I provide my own solution to each of them.

This book is also useful to prepare for hiring interviews. And for hiring managers, it is full of original and open questions, never posted before, encouraging candidates to think outside the box, with applications on real data. The amount of Python code accompanying the solutions is tremendous, using a vast array of libraries as well as home-made implementations showing the inner workings and improving existing black-box algorithms. By itself, this book constitutes a solid introduction to Python. The code is also on my GitHub repository.

The topics cover generative AI, synthetic data, machine learning optimization, scientific computing with Python, data visualizations and animations, time series and spatial processes, NLP and large language models, as well as graph applications and more. These projects based on real life data, with solution and enterprise-grade Python code, are a great addition to your portfolio (GitHub) when completed. Hiring managers and instructors can use them as as a complement to their battery of tests and projects, to differentiate themselves from competitors relying on overused, run-of-the-mill exercises.

To see how the program works and earn your certification(s), check out our FAQ posted here, or click on the "certification" tab on our website MLtechniques.com. Certifications can easily be displayed on your LinkedIn profile page in the credentials section, with just one click. Unlike many other programs, there is no exam or meaningless quizzes. Emphasis is on projects with real-life data, enterprise-grade code, efficient methods, and modern applications to build a strong portfolio and grow your career in little time. The guidance to succeed is provided by the founder of the company, one of the top and most well-known experts in machine learning, Dr. Vincent Granville. Jargon and unnecessary math are avoided, and simplicity is favored whenever possible. Nevertheless, the material is described as advanced by everyone who looked at it.

The related teaching and technical material (textbooks) can be purchased at MLtechniques.com/shop/. MLtechniques.com, the company offering the certifications, is a private, self-funded AI/ML research lab developing state-of-the-art open source technologies related to synthetic data, generative AI, cybersecurity, geospatial modeling, stochastic processes, chaos modeling, and AI-related statistical optimization.

About the author

Vincent Granville is a pioneering data scientist and machine learning expert, co-founder of Data Science Central (acquired by TechTarget), founder of MLTechniques.com, former VC-funded executive, author and patent owner.

Vincent's past corporate experience includes Visa, Wells Fargo, eBay, NBC, Microsoft, and CNET. Vincent is also a former post-doc at Cambridge University, and the National Institute of Statistical Sciences (NISS). He published in *Journal of Number Theory*, *Journal of the Royal Statistical Society* (Series B), and *IEEE Transactions on Pattern Analysis and Machine Intelligence*. He is also the author of multiple books, available here. He lives in Washington state, and enjoys doing research on stochastic processes, dynamical systems, experimental math and probabilistic number theory.

- To measure the quality of fit, it is tempting to use the correlation between simulated and real data. However this approach favors simulated data that is a replicate of the original data. To the contrary, comparing the two auto-correlation structures favors simulated data of the same type as the real data, but not identical. It leads to a richer class of synthetic time series, putting emphasis on structural and stochastic similarity, rather than being "the same". It also minimizes overfitting.
- Try different seeds for the random generator, and see how the solution changes based on the seed. Also, rather than using the sum of absolute value of differences between various autocorrelation lags, try the max, median, or assign a different weight to each lag (such as decaying weights). Or use transformed auto-correlations using a logarithm transform.
- A classic metric to assess the quality of synthetic data is the Hellinger distance, popular because it yields a value between 0 and 1. It measures the proximity between the two marginal distributions – here, that of the simulated and real time series. It is not useful for time series though, because you can have the same marginals and very different auto-correlation structures. Note that the metric I use also yields values between 0 and 1, with zero being best, and 1 being worst.
- The simulation was able to generate values outside the range of observed (real) values. Many synthetic data algorithms fail at that, because they use percentile-based methods (for instance, copulas) for data generation or to measure the quality (Hellinger is in that category). Empirical percentile distributions used for that purpose, including the Python version in the Statsmodels library, have this limitation.

3.2 Geospatial interpolation, kriging and smoothness metrics

The purpose here is twofold. First, using the pykrige Python library to performs ordinary kriging, to estimate temperatures outside the sensor locations for the Chicago temperature dataset `sensors.csv` available on GitHub, here. The details are in my book [7], in chapter 9. In addition, we will use the osmnx library (Open Street Map) to superimpose a map of the Chicago area on the final output, as seen in Figure 3.10.

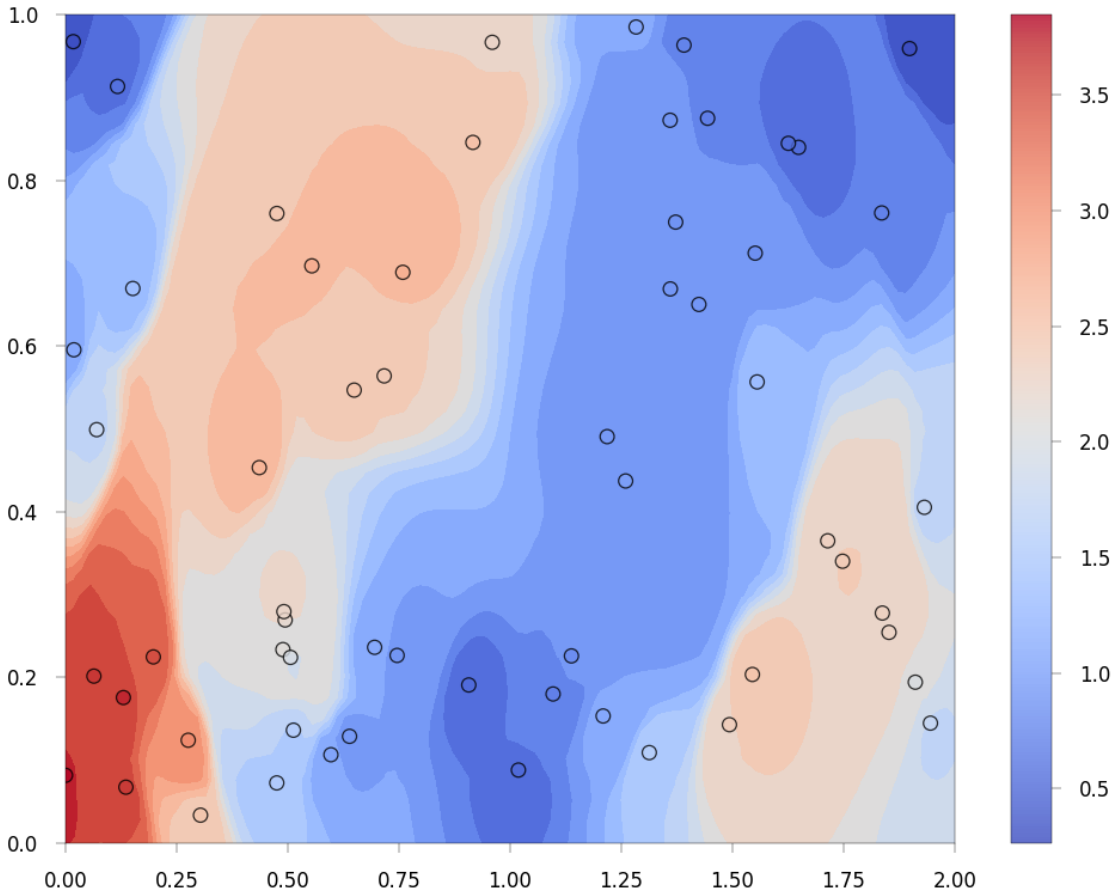


Figure 3.6: Interpolation of the entire grid; dots are training set locations

Then, we will use a generated dataset, with temperatures replaced by an arbitrary math function, in this case a mixture of bivariate Gaussian densities, or Gaussian mixture model. This allows us to simulate hundreds

or thousands of values at arbitrary locations, by contrast with the temperature dataset based on 31 observations only. The math function in question is pictured in Figure 3.7. In this case, rather than kriging, we explore an exact bivariate interpolation method, also featured in chapter 9 in my book. The goal is to compare with kriging, using cross-validation, as shown in Figure 3.8. The solution offered here is a lot faster than the code in my book, thanks to replacing loops with vector and matrix operations. Eventually, we interpolate the entire grid: see Figure 3.6. Picking up random locations on the grid, together with the corresponding interpolated values, produces synthetic geospatial data. It can be used to generate artificial elevation maps, with potential applications in video games.

Last but not least, we define the concept of spatial smoothness for geospatial data, using functions of second-order discrete derivatives. While it is easy to play with parameters of any given algorithm to produce various degrees of smoothness, it is a lot harder to define smoothness in absolute terms, allowing you to compare results produced by two different algorithms. Another important point is overfitting. Despite using exact interpolation for hundreds of locations – akin to using a regression model with hundreds of regression coefficients – we are able to avoid overfitting.

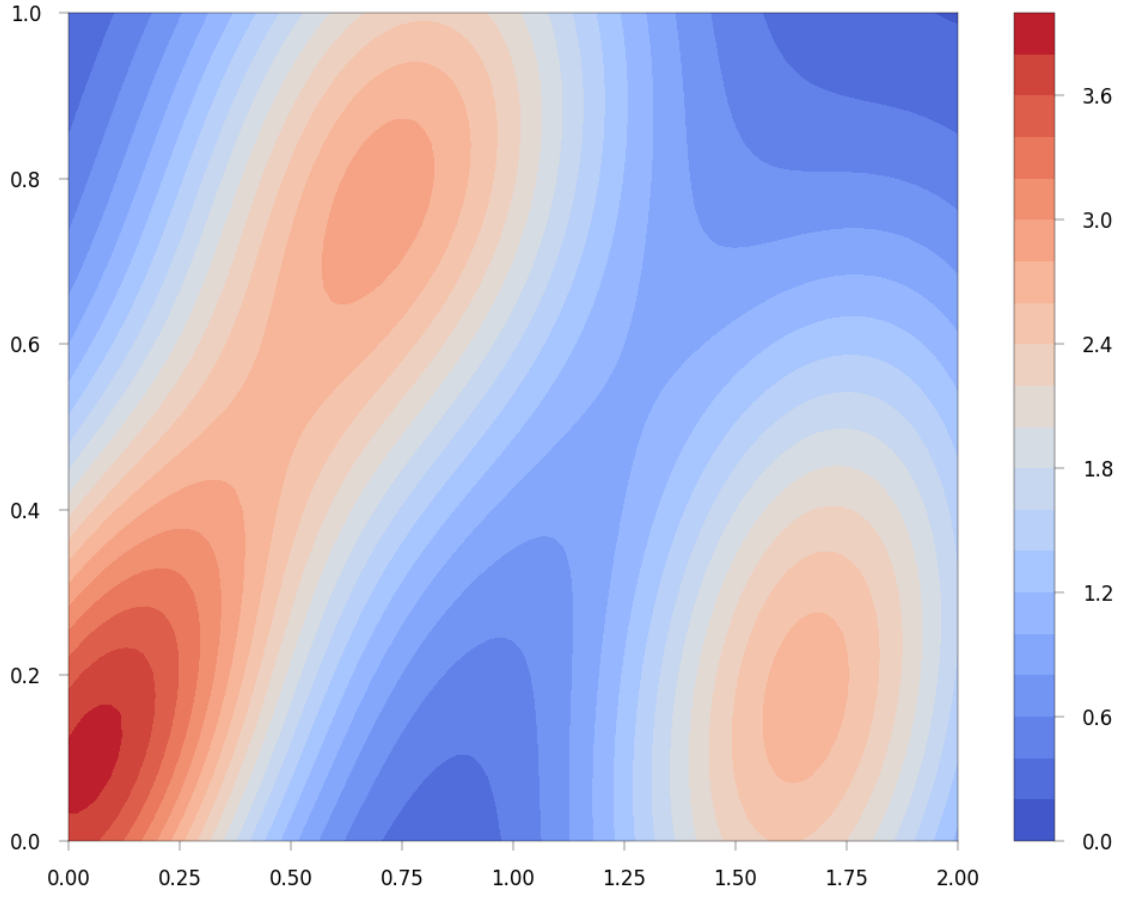


Figure 3.7: Math function used to sample training set locations and values

3.2.1 Project description

The Chicago temperature dataset `sensors.csv` is available here. It contains latitudes, longitudes and temperatures measured at 31 locations, at one point in time. We will use it to illustrate kriging. The algorithm for exact geospatial interpolation is in chapter 9 in my book [7]. The original study of the temperature data set is in the same chapter. A Jupyter notebook, published by the University of Illinois, can be found here.

The project consists of the following steps:

Step 1: Using the Pykrige library, write Python code to perform ordinary kriging on the temperature data set and interpolate values in the entire area. You may look at the Jypiter notebook aforementioned, to eventually produce a picture similar to Figure 3.10. Try different types of kriging and parameters. Does it always lead to smooth interpolation, similar to time series smoothing? What happens when extrapolating, that is, sampling far outside the training set?

Step 2: Let us consider the following mathematical function, a mixture of m bivariate Gaussian densities defined on $D = [0, 2] \times [0, 1]$ with weights w_k, centers (c_{xk}, c_{yk}), variances σ_{xk}, σ_{yk}, and correlations ρ_k:

$$f(x, y) \propto \sum_{k=1}^{m} \frac{w_k}{\sigma_{xk}\sigma_{yk}\sqrt{1-\rho_k^2}} \exp\left[-\left\{\frac{(x-c_{xk})^2}{\sigma_{xk}^2} - \frac{2\rho_k(x-c_{xk})(y-c_{yk})}{\sigma_{xk}\sigma_{yk}} + \frac{(y-c_{yk})^2}{\sigma_{yk}^2}\right\}\right]. \quad (3.3)$$

Here the symbol $\propto$ means "proportional to"; the proportionality constant does not matter.

Choose specific values for the parameters, and sample two sets of locations on D: the training set to define the interpolation formula based on the values of the above function at these locations, and the validation set to check how good the interpolated values are, outside the training set.

Step 3: Use the interpolation formula implemented in section 9.3.2 in my book [7] and also available here on GitHub, but this time on the training and validation sets obtained in step 2, rather than on the Chicago temperature dataset. Also interpolate the entire grid, using 10,000 locations evenly spread on D. Plot the results using scatterplots and contour maps. Compute the interpolation error on the validation set. Show how the error is sensitive to the choice of sampled locations and parameters. Also show that the contour maps for interpolated values are considerably less smooth than for the underlying math function, due to using exact interpolation. Would this be also true if using kriging instead? What are your conclusions?

Step 4: The original code in my book runs very slowly. Detect the bottlenecks. How can you improve the speed by several orders of magnitude? Hint: replace some of the loops by array operations, in Numpy.

Step 5: The goal is to define the concept of smoothness, to compare interpolations and contour maps associated to different algorithms, for instance kriging versus my exact interpolation technique. Unlike 1D time series, there is no perfect answer in 2D: many definitions are possible, depending on the type of data. For instance, it could be based on the amount of chaos or entropy. Here we will use a generalization of the 1D metric

$$S(f, D) = \int_D |f''(w)|^2 dw.$$

Explain why definitions based on first-order derivatives are not good. Search the Internet for a potential solution. I did not find any, but you can check out the answer to the question I posted on Mathoverflow, here. You may create a definition based on transformed gradients and Hessians, such as a matrix norm of the Hessian. These objects are respectively the first and second derivatives (a vector for the gradient, a matrix for the Hessian) attached to multivariate functions. Compute the smoothness on different interpolated grids, to see if your definition matches intuition. You will need a discrete version of the gradient or Hessian, as we are dealing with data (they don't have derivatives!) rather than mathematical functions. Numpy has functions such as `gradient` that do the computations in one line of code, when the input is a grid.

Step 6: Optional. How would you handle correlated bivariate values, such as temperature and pressure measured simultaneously at various locations? How about spatio-temporal data: temperatures evolving over time across multiple locations? Finally, turn Figure 3.6 into a video, by continuously modifying the parameters in the function defined by (3.3), over time, with each video frame corresponding to updated parameters. This is how agent-based modeling works.

3.2.2 Solution

The code to solve step 1 is in section 3.2.2.1. In addition, I used the Osmnx library to superimpose the Chicago street map on the temperature 2D grid. For step 2, see the `gm` function in the code featured in section 3.2.2.2. The function `interpolate` in the same program, is the implementation of the interpolation formula discussed in step 3. More about step 3 and step 4 can be found in the same section. As for step 5, I implemented a discrete version of the following formula, to define and compute the smoothness S on $[0, 2] \times [0, 1]$:

$$S = \int_0^1 \int_0^2 |\nabla(|\nabla z(x, y)|)| dxdy$$

where z is the value at location (x, y) on the grid, ∇ is the gradient operator, and $|\cdot|$ is the Eulidean norm. The discrete gradient on the grid is computed in one line of code, with the `gradient` function available in Numpy.

Regarding the difference between kriging and my interpolation method (last questions in step 3), kriging tends to produce much smoother results: it is good for measurements such as temperatures, with a smooth gradient. Chaotic processes, for instance the reconstruction of an elevation map or structures similar to Voronoi

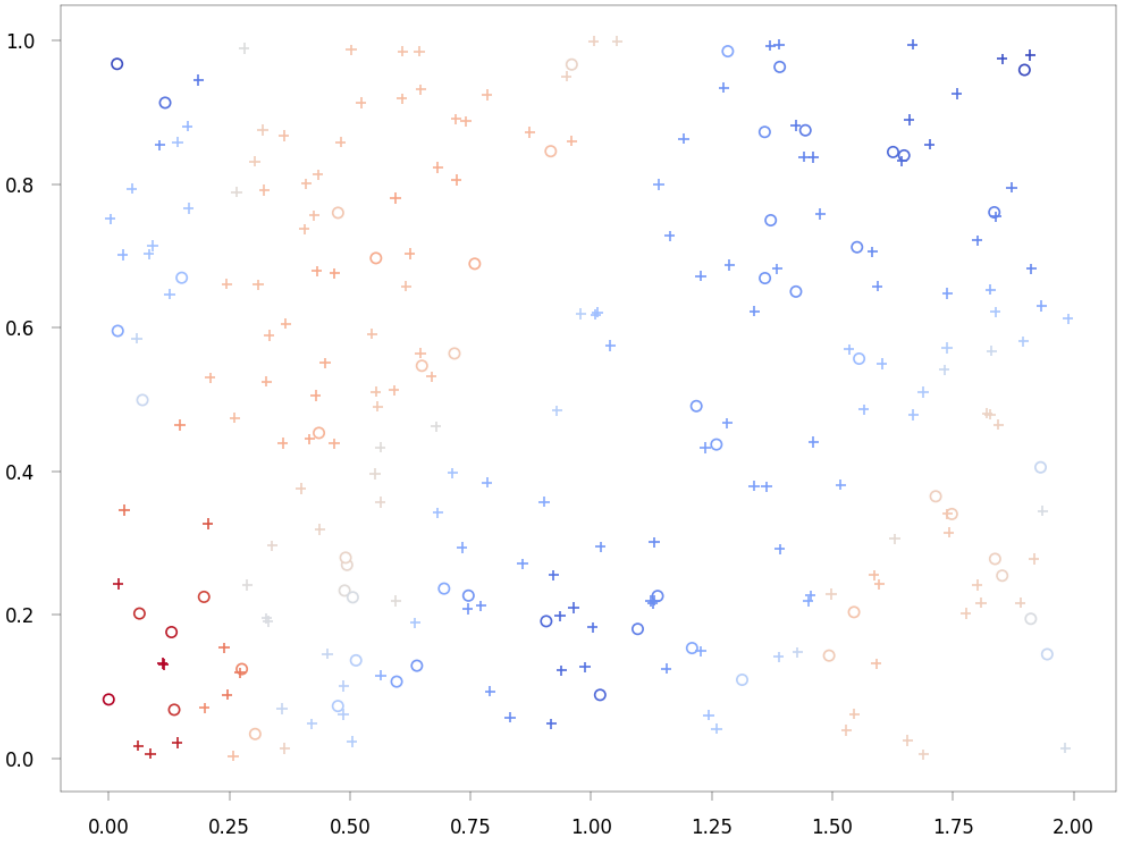


Figure 3.8: Interpolation: dots for training locations, + for validation points

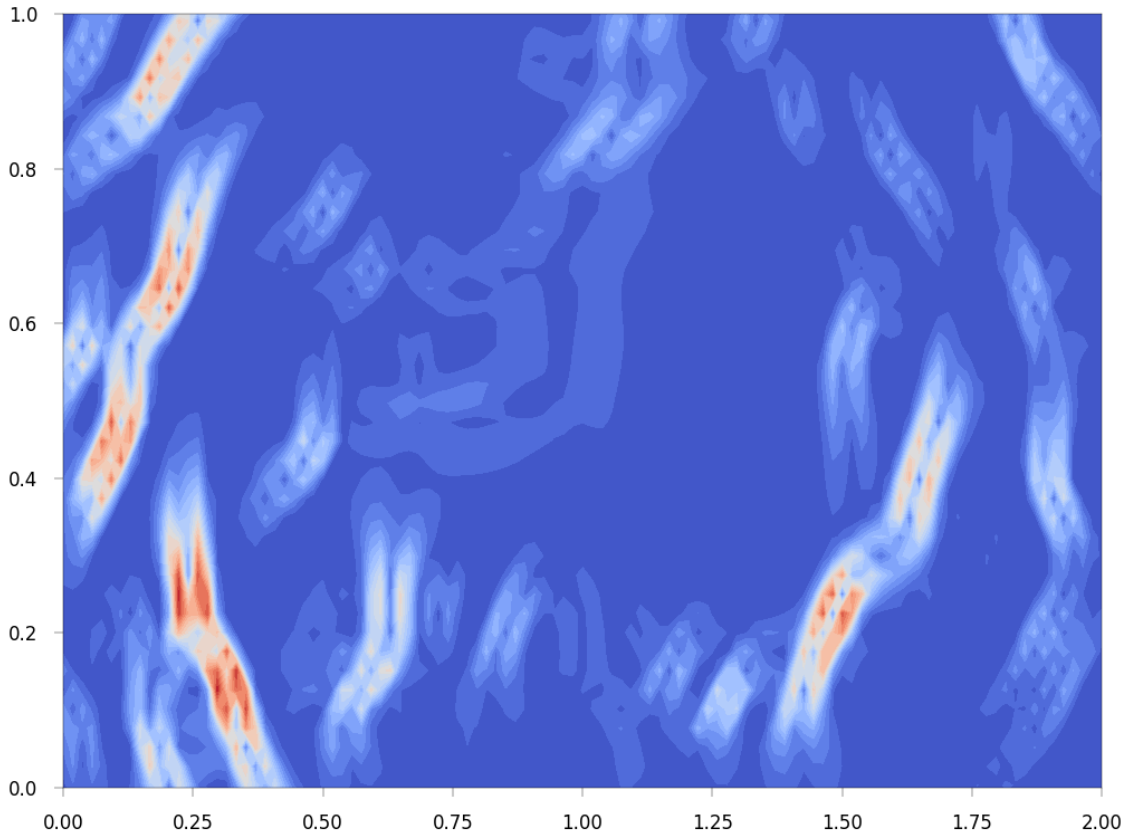


Figure 3.9: Interpolation of entire grid: second-order gradient representing smoothness

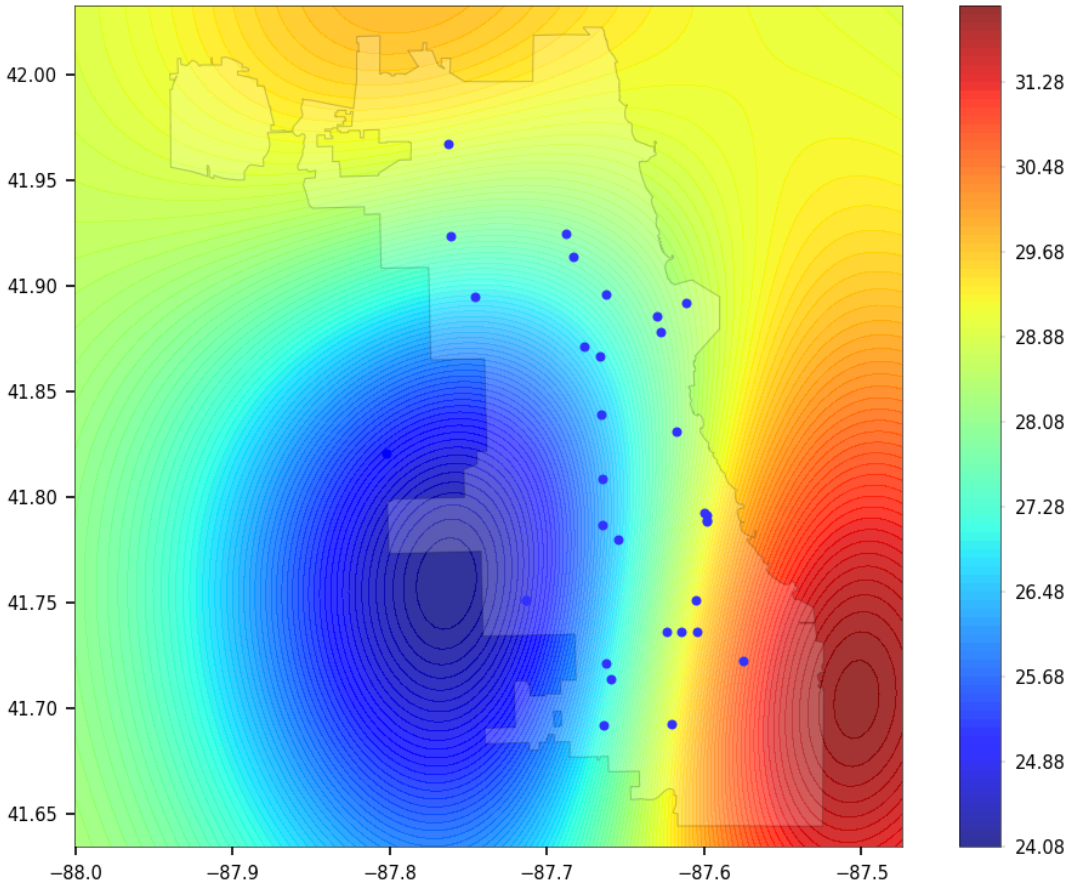


Figure 3.10: Kriging, temperature dataset; dots correspond to actual measurements

diagrams [Wiki] are much better rendered with my method, especially when few data points are available. It preserves local variations much better than kriging. Finally, despite offering exact interpolation, my method avoids overfitting, unlike polynomial regression. This is because I use some normalization in the interpolation formula. In short, kriging is a smoothing technique, while my method is best used for data reconstruction or synthetization.

There are many other potential topics to address. I listed below a few suggestions for the reader interested in further exploring this project.

- Play with the parameters $\alpha, \beta, \kappa, \delta$ to increase or decrease smoothness in the exact interpolation method, and see the impact on the error rate (measured on the validation set).
- Add noise to the observed values in the training set, and assess sensitivity of interpolated values to various levels of noise.
- Play with the parameters associated to the `gm` function, to produce many different sets of observed values. Compute the error (`error` in the code in section 3.2.2.2) and relative error, for each parameter set. What is the range of the relative error, depending on the size of the training set?
- Investigate other metrics to measure smoothness (step 5 in the project), for instance 2D generalizations of Hurst exponent [Wiki] used for time series.
- When is it useful to first transform the data, interpolate the transformed data, then apply the inverse transform? For instance, this is done for the Chicago temperature dataset: see chapter 9 in my book [7].
- How far outside the training set locations can you reasonably interpolate without losing too much accuracy? In this case, it is called extrapolation. Check the accuracy of interpolated values for locations in the validation set that are far away from any training set point.
- Compute confidence intervals for the interpolated values (validation set). In order to do so, generate 1000 training sets, each with the same number of points, but different locations. Or use the same training set each time, with a resampling technique such as bootstrapping [Wiki].

Regarding step 6, see how to create a data video in section 1.3.2 in this textbook. An example relevant to this project – an animated elevation map using agent-based modeling – can be found in chapter 14 in my book [7].

3.2.2.1 Kriging

The Chicago temperature dataset is discussed in chapter 9 in my book [7]. My code is inspired from a Jupyter notebook posted here by a team working on this project at the University of Chicago. My Python implementation `kriging_temperatures_chicago.py` listed below is also on GitHub, here.

```
# compare this with my spatial interpolation:
# https://github.com/VincentGranville/Statistical-Optimization/blob/main/interpol.py
# This kriging oversmooth the temperatures, compared to my method

import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import colors
from matplotlib import cm # color maps
import osmnx as ox
import pandas as pd
import glob
from pykrige.ok import OrdinaryKriging
from pykrige.kriging_tools import write_asc_grid
import pykrige.kriging_tools as kt
from matplotlib.colors import LinearSegmentedColormap

city = ox.geocode_to_gdf('Chicago, IL')
city.to_file("il-chicago.shp")

data = pd.read_csv(
    'sensors.csv',
    delim_whitespace=False, header=None,
    names=["Lat", "Lon", "Z"])

lons=np.array(data['Lon'])
lats=np.array(data['Lat'])
zdata=np.array(data['Z'])

import geopandas as gpd
Chicago_Boundary_Shapefile = 'il-chicago.shp'
boundary = gpd.read_file(Chicago_Boundary_Shapefile)

# get the boundary of Chicago
xmin, ymin, xmax, ymax = boundary.total_bounds

xmin = xmin-0.06
xmax = xmax+0.05
ymin = ymin-0.01
ymax = ymax+0.01
grid_lon = np.linspace(xmin, xmax, 100)
grid_lat = np.linspace(ymin, ymax, 100)

#------
# ordinary kriging

OK = OrdinaryKriging(lons, lats, zdata, variogram_model='gaussian', verbose=True,
    enable_plotting=False,nlags=20)
z1, ss1 = OK.execute('grid', grid_lon, grid_lat)
print (z1)

#-------
# plots

xintrp, yintrp = np.meshgrid(grid_lon, grid_lat)
plt.rcParams['axes.linewidth'] = 0.3
fig, ax = plt.subplots(figsize=(8,6))

contour = plt.contourf(xintrp, yintrp, z1,len(z1),cmap=plt.cm.jet,alpha = 0.8)
cbar = plt.colorbar(contour)
```

```
cbar.ax.tick_params(width=0.1)
cbar.ax.tick_params(length=2)
cbar.ax.tick_params(labelsize=7)

boundary.plot(ax=ax, color='white', alpha = 0.2, linewidth=0.5, edgecolor='black',
    zorder = 5)
npts = len(lons)

plt.scatter(lons, lats,marker='o',c='b',s=8)
plt.xticks(fontsize = 7)
plt.yticks(fontsize = 7)
plt.show()
```

3.2.2.2 Exact interpolation and smoothness evaluation

The instruction `(za,npt)=interpolate(xa,ya,npdata,0.5*delta)` produces interpolated values `za` for the the validation set locations `(xa,ya)`. The parameter `delta` controls the maximum distance allowed between a location, and a training set point used to interpolate the value at the location in question. The array `npt` stores the number of training set points used to compute each interpolated value. Also, the instruction `z,npt=interpolate(xg,yg,npdata,2.2*delta)` interpolates the entire grid.

The number of loops has been reduced to make the code run faster. In particular, the arguments `xa,ya` can be arrays; accordingly, the output `z` of the `interpolate` function can be an array or even a grid of interpolated values. Likewise, `npt` can be an array or grid, matching the shape of `z`. Finally, `error` measures the mean absolute error between exact and interpolated values on the validation set, while `average_smoothness` measures the smoothness of the grid, original or interpolated. The Python code, `interpol_smooth.py`, is also on GitHub, here.

```
import warnings
warnings.filterwarnings("ignore")

import numpy as np
import matplotlib as mpl
from matplotlib import pyplot as plt
from matplotlib import colors
from matplotlib import cm # color maps

n = 60      # number of observations in training set
ngroups = 3 # number of clusters in Gaussian mixture
seed = 13   # to initiate random number generator

#--- create locations for training set

width = 2
height = 1
np.random.seed(seed)
x = np.random.uniform(0, width, n)
y = np.random.uniform(0, height, n)


#--- create values z attached to these locations

weights = np.random.uniform(0.5, 0.9, ngroups)
sum = np.sum(weights)
weights = weights/sum

cx = np.random.uniform(0, width, ngroups)
cy = np.random.uniform(0, height, ngroups)
sx = np.random.uniform(0.3, 0.4, ngroups)
sy = np.random.uniform(0.3, 0.4, ngroups)
rho = np.random.uniform(-0.3, 0.5, ngroups)


def f(x, y, cx, cy, sx, sy, rho):
```

```
    # bivariate bell curve

    tx = ( (x - cx) / sx)**2
    ty = ( (y - cy) / sy)**2
    txy = rho * (x - cx) * (y - cy) / (sx * sy)
    z = np.exp(-(tx - 2*txy + ty) / (2*(1 - rho**2)) )
    z = z / (sx * sy * np.sqrt(1 - rho**2))
    return(z)

def gm(x, y, weights, cx, cy, sx, sy, rho):

    # mixture of gaussians

    n = len(x)
    ngroups = len(cx)
    z = np.zeros(n)
    for k in range(ngroups):
        z += weights[k] * f(x, y, cx[k], cy[k], sx[k], sy[k], rho[k])
    return(z)

z = gm(x, y, weights, cx, cy, sx, sy, rho)
npdata = np.column_stack((x, y, z))

print(npdata)

#--- model parameters

alpha = 1.0   # small alpha increases smoothing
beta = 2.0    # small beta increases smoothing
kappa = 2.0   # high kappa makes method close to kriging
eps  = 1.0e-8 # make it work if sample locations same as observed ones
delta = eps + 1.2 * max(width, height) # don't use faraway points for interpolation

#--- interpolation for validation set: create locations

n_valid = 200 # number of locations to be interpolated, in validation set
xa = np.random.uniform(0, width, n_valid)
ya = np.random.uniform(0, height, n_valid)

#--- interpolation for validation set

def w(x, y, x_k, y_k, alpha, beta):
    # distance function
    z = (abs(x - x_k)**beta + abs(y - y_k)**beta)**alpha
    return(z)


def interpolate(x, y, npdata, delta):

    # compute interpolated z at location (x, y) based on npdata (observations)
    # also returns npt, the number of data points used for each interpolated value
    # data points (x_k, y_k) with w[(x,y), (x_k,y_k)] >= delta are ignored
    # note: (x, y) can be a location or an array of locations

    if np.isscalar(x): # transform scalar to 1-cell array
        x = [x]
        y = [y]
    sum = np.zeros(len(x))
    sum_coeff = np.zeros(len(x))
    npt = np.zeros(len(x))

    for k in range(n):
        x_k = npdata[k, 0]
        y_k = npdata[k, 1]
        z_k = npdata[k, 2]
        coeff = 1
```

```
      for i in range(n):
         x_i = npdata[i, 0]
         y_i = npdata[i, 1]
         if i != k:
            numerator = w(x, y, x_i, y_i, alpha, beta)
            denominator = w(x_k, y_k, x_i, y_i, alpha, beta)
            coeff *= numerator / (eps + denominator)
      dist = w(x, y, x_k, y_k, alpha, beta)
      coeff = (eps + dist)**(-kappa) * coeff / (1 + coeff)
      coeff[dist > delta] = 0.0
      sum_coeff += coeff
      npt[dist < delta] += 1
      sum += z_k * coeff

   z = sum / sum_coeff
   return(z, npt)

(za, npt) = interpolate(xa, ya, npdata, 0.5*delta)

#--- create 2D grid with x_steps times y_steps locations, and interpolate entire grid

x_steps = 160
y_steps = 80
xb = np.linspace(min(npdata[:,0])-0.50, max(npdata[:,0])+0.50, x_steps)
yb = np.linspace(min(npdata[:,1])-0.50, max(npdata[:,1])+0.50, y_steps)
xc, yc = np.meshgrid(xb, yb)

zgrid = np.empty(shape=(x_steps,y_steps)) # for interpolated values at grid locations
xg = []
yg = []
gmap = {}
idx = 0
for h in range(len(xb)):
   for k in range(len(yb)):
      xg.append(xb[h])
      yg.append(yb[k])
      gmap[h, k] = idx
      idx += 1
z, npt = interpolate(xg, yg, npdata, 2.2*delta)

zgrid_true = np.empty(shape=(x_steps,y_steps))
xg = np.array(xg)
yg = np.array(yg)
z_true = gm(xg, yg, weights, cx, cy, sx, sy, rho) # exact values on the grid

for h in range(len(xb)):
   for k in range(len(yb)):
      idx = gmap[h, k]
      zgrid[h, k] = z[idx]
      zgrid_true[h, k] = z_true[idx]
zgridt = zgrid.transpose()
zgridt_true = zgrid_true.transpose()

#--- visualizations

nlevels = 20 # number of levels on contour plots

def set_plt_params():
   # initialize visualizations
   fig = plt.figure(figsize =(4, 3), dpi=200)
   ax = fig.gca()
   plt.setp(ax.spines.values(), linewidth=0.1)
   ax.xaxis.set_tick_params(width=0.1)
   ax.yaxis.set_tick_params(width=0.1)
   ax.xaxis.set_tick_params(length=2)
   ax.yaxis.set_tick_params(length=2)
```

```
    ax.tick_params(axis='x', labelsize=4)
    ax.tick_params(axis='y', labelsize=4)
    plt.rc('xtick', labelsize=4)
    plt.rc('ytick', labelsize=4)
    plt.rcParams['axes.linewidth'] = 0.1
    return(fig,ax)

# contour plot, interpolated values, full grid

(fig1, ax1) = set_plt_params()
cs1 = plt.contourf(xc, yc, zgridt, cmap='coolwarm',levels=nlevels,linewidths=0.1)
sc1 = plt.scatter(npdata[:,0], npdata[:,1], c=npdata[:,2], s=8, cmap=cm.coolwarm,
     edgecolors='black',linewidth=0.3,alpha=0.8)
cbar1 = plt.colorbar(sc1)
cbar1.ax.tick_params(width=0.1)
cbar1.ax.tick_params(length=2)
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()

# scatter plot: validation set (+) and training data (o)

(fig2, ax2) = set_plt_params()
my_cmap = mpl.colormaps['coolwarm'] # old version: cm.get_cmap('coolwarm')
my_norm = colors.Normalize()
ec_colors = my_cmap(my_norm(npdata[:,2]))
sc2a = plt.scatter(npdata[:,0], npdata[:,1], c='white', s=5, cmap=my_cmap,
    edgecolors=ec_colors,linewidth=0.4)
sc2b = plt.scatter(xa, ya, c=za, cmap=my_cmap, marker='+',s=5,linewidth=0.4)
plt.show()
plt.close()

#--- measuring quality of the fit on validation set
#   zd is true value, za is interpolated value

zd = gm(xa, ya, weights, cx, cy, sx, sy, rho)
error = np.average(abs(zd - za))
print("\nMean absolute error on validation set: %6.2f" %(error))
print("Mean value on validation set: %6.2f" %(np.average(zd)))

#--- plot of original function (true values)

(fig3, ax3) = set_plt_params()
cs3 = plt.contourf(xc, yc, zgridt_true, cmap='coolwarm',levels=nlevels,linewidths=0.1)
cbar1 = plt.colorbar(cs3)
cbar1.ax.tick_params(width=0.1)
cbar1.ax.tick_params(length=2)
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()

#--- compute smoothness of interpolated grid via double gradient
#   1/x_steps and 1/y_steps are x, y increments between 2 adjacent grid locations

h2 = x_steps**2
k2 = y_steps**2
dx, dy = np.gradient(zgrid) # zgrid_true for original function
zgrid_norm1 = np.sqrt(h2*dx*dx + k2*dy*dy)
dx, dy = np.gradient(zgrid_norm1)
zgrid_norm2 = np.sqrt(h2*dx*dx + k2*dy*dy)
zgridt_norm2 = zgrid_norm2.transpose()
average_smoothness = np.average(zgrid_norm2)
print("Average smoothness of interpolated grid: %6.3f" %(average_smoothness))

(fig4, ax4) = set_plt_params()
cs4 = plt.contourf(xc, yc, zgridt_norm2, cmap=my_cmap,levels=nlevels,linewidths=0.1)
```

```
plt.xlim(0, width)
plt.ylim(0, height)
plt.show()
```

Bibliography

[1] Ramiro Camino, Christian Hammerschmidt, and Radu State. Generating multi-categorical samples with generative adversarial networks. *Preprint*, pages 1–7, 2018. arXiv:1807.01202 [Link]. 55

[2] Fida Dankar et al. A multi-dimensional evaluation of synthetic data generators. *IEEE Access*, pages 11147–11158, 2022. [Link]. 54

[3] Vincent Granville. Feature clustering: A simple solution to many machine learning problems. *Preprint*, pages 1–6, 2023. MLTechniques.com [Link]. 43

[4] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 36

[5] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 43

[6] Vincent Granville. Smart grid search for faster hyperparameter tuning. *Preprint*, pages 1–8, 2023. MLTechniques.com [Link]. 41, 43

[7] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 14, 19, 20, 21, 23, 24, 30, 31, 33, 35, 36, 40, 41, 43, 44, 46, 54, 55

[8] Elisabeth Griesbauer. *Vine Copula Based Synthetic Data Generation for Classification*. 2022. Master Thesis, Technical University of Munich [Link]. 43

[9] Chang Su, Linglin Wei, and Xianzhong Xie. Churn prediction in telecommunications industry based on conditional Wasserstein GAN. *IEEE International Conference on High Performance Computing, Data, and Analytics*, pages 186–191, 2022. IEEE HiPC 2022 [Link]. 54

[10] Ruonan Yu, Songhua Liu, and Xinchao Wang. Dataset distillation: A comprehensive review. *Preprint*, pages 1–23, 2022. Submitted to IEEE PAMI [Link]. 41

Index