Vignette for geospaNN

1 Summary

`geospaNN` stands for **Geospa**tial **N**eural **N**etworks and is a python package that implements NN-GLS, a geographically-informed Graph Neural Network (GNN) architecture (Zhan and Datta 2024). This file provides a brief introduction on how to use `geospaNN` for geospatial data analysis. The vignette is divided into four parts:

- **Data preparation:** This section covers features that transform raw geospatial data into the input format for the Neural Network trainer. It also introduces a geospatial simulation function for conducting general statistical experiments.
- **Model training:** This section explains the main functions used to train the model. To streamline the common training procedure, a user-friendly trainer is introduced for NN-GLS.
- **Model evaluation:** This section visualizes the training process and spatial parameter estimation. It also provides functions for mean-function estimation and prediction at new location, which are the primary goals of the package and can support various geospatial applications.
- **Additional examples:** This section is independent with the main pipeline in the sections above and provides additional examples where `geospaNN` can be applied in different scenarios.

In addition to this vignette, `geospaNN`'s official website (https://wentaozhan1998.github.io/geospaNN-doc/) offers an installation guide, comprehensive documentation, and several statistical experiments. The demo codes for this vignette is available at (https://wentaozhan1998.github.io/geospaNN-doc/Examples/).

2 Main Content

We will start by loading the `geospaNN` and other dependencys for this vignette:

```
import torch
import geospaNN
import numpy as np
import time
import pandas as pd
import seaborn as sns
import random

import matplotlib
import matplotlib.pyplot as plt

path = '../data/Output/'
```

2.1 Spatial Mixed Effect Model (SPMM) and Goal

In this vignette, we focus on the spatial mixed effect model, formulated as:

$$Y(s) = m\big(X(s)\big) + w(s) + \epsilon(s),$$

where:

- $X(s)$ are the observed covariates at location s,
- $Y(s)$ is the observed outcome,
- $\epsilon(s)$ is the i.i.d. random Gaussian noise following $N(0, \tau\sigma^2)$, also known as the nugget in spatial literature, where σ^2 denotes the variance of the spatial effect, and τ quantifies the proportion of non-spatial variance relative to the spatial variance.
- $w(s)$ represents a stochastic process accounting for spatial correlation, modeled as a mean-zero exponential Gaussian Process (GP) with the following covariance structure:

$$\operatorname{cov}\big(w(s_1), w(s_2)\big) = C(s_1, s_2|\theta) = \sigma^2 \exp\left(-\phi\|s_1 - s_2\|\right).$$

  Here, θ is introduced as a general function for (σ^2, ϕ, τ). Under this setting, `geospaNN` aims to estimate the non-linear mean function $m(\cdot)$ by using the Neural Network family (estimation task), as well as predict the response $y(s_{new})$ at new locations s_{new} by combining the mean estimation $\hat{m}(\cdot)$ and the kriging prediction $\hat{w}(s_{new})$. Nearest Neighbor Gaussian Process (NNGP) Datta et al. (2016) is used as an approximate process to the full GP, ensuring the scalability of `geospaNN`. This enables analysis on geospatial datasets with up to 1 million observations.

2.2 Simulation and Data Preprocessing

2.2.1 Simulation

Simulation of geospatial data is a key feature of `geospaNN`, enabling the generation of geospatial datasets for various experiments and facilitating methodology development with flexibility and efficiency. Simulating spatially correlated data typically requires $O(n^3)$ running time due to the need to compute the Cholesky factor of an $n \times n$ Gaussian process covariance matrix. However, NNGP provides efficient precision matrix computation and Cholesky factorization, reducing the time complexity to $O(n)$ for large sample sizes.

In this section, we simulate data from the model described earlier. Specifically, we use the function `funXY`:

$$m(x) = 10\sin(2\pi x).$$

The simulation setup includes:

- The spatial effect $w(s)$, generated as an exponential Gaussian process with spatial variance $\sigma^2 = 1$ (`sigma`) and spatial correlation decay $\phi = 3\sqrt{2}/20$ (`phi`).
- The i.i.d. random noise $\epsilon(s)$, with variance $\eta^2 = \tau \cdot \sigma^2$, where $\tau = 0.01$ (`tau`).

Additional required arguments include:

- Sample size (`n`): The desired number of locations.
- Number of covariates (`p`): The number of predictors.
- Number of neighbors (`nn`): Used for NNGP approximation.
- Range of spatial locations (`range`): Specifies the 2D domain $[\texttt{range}]^2$ from which spatial locations will be sampled.

The function returns the following in tensor formats: the coordinates `X`, response `Y`, coordinates `coord`, covariance matrix `cov` in sparse format, and random effect $w(s) + \epsilon(s)$ `corerr`.

```
def f1(X): return 10 * np.sin(2*np.pi * X)

p = 1;
funXY = f1
```

```
n = 1000
nn = 20
batch_size = 50

sigma = 1
phi = 0.3
tau = 0.01
theta = torch.tensor([sigma, phi / np.sqrt(2), tau])

X, Y, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
```

2.2.2 Practical Simulation Approach

In practice, the covariate X itself is often spatially correlated (denoted as $X(s)$). A practical shortcut for simulating in this setting involves:

1. Using `geospaNN.Simulation` to generate one correlated spatial process `corerr` as $X(s)$.
2. Generating another `corerr` as $w(s) + \epsilon(s)$.
3. Assembling Y as $m\big(X(s)\big) + w(s) + \epsilon(s)$.

The process is illustrated in the following code chunks, and figure 1 visualizes the simulated data.

```
#1
_, _, _, _, X = geospaNN.Simulation(n, p, nn, funXY, torch.tensor([1, 5, 0.01]), range=[0, 1])
X = X.reshape(-1,p)
X = (X - X.min())/(X.max() - X.min())
#2
_, _, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
#3
Y = funXY(X).reshape(-1) + corerr
```

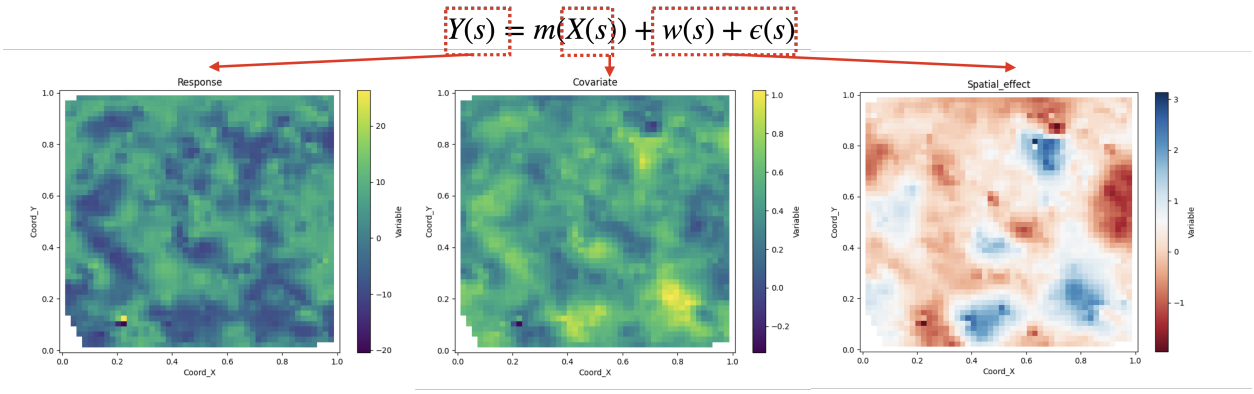


Figure 1: Simulated Data Visualization

2.2.3 Data Preprocessing

Covariates X $(n \times p)$, response Y $(n \times 1)$, and coordinates s $(n \times 2)$ form the fundamental elements of a geospatial dataset. However, to prepare the dataset for a GNN module, a directed graph must be constructed

using the `geospaNN.make_graph` function. This function creates a `DataLoader` object for efficient data loading and management. In addition to `X`, `Y`, and `coord`, it requires:

- The number of nearest neighbors (`nn`), since the function constructs a k-nearest-neighbor graph by default, with k specified by `nn`.
- An $n \times k$ index tensor `Ind_list`, which allows customization of the graph using. The i-th row contains the indices of nodes connected to s_i, with `-1` indicating no connection. If no index tensor is specified, a k-nearest-neighbor graph mentioned above will be constructed by default.

```
data = geospaNN.make_graph(X, Y, coord, nn, Ind_list = None)
```

Additionally, splitting of the dataset into training, testing, and validation sets is simplified using the `geospaNN.split_data` function. Like `geospaNN.make_graph`, this function takes covariates X, response Y, coordinates s, and neighbor size as inputs. Users can customize the split proportions with `val_proportion` and `test_proportion`, both defaulting to 0.2. The output includes three `DataLoader` objects required for the training process.

```
data_train, data_val, data_test = geospaNN.split_data(X, Y,coord,neighbor_size=nn,
test_proportion=0.2,val_proportion=0.2)
```

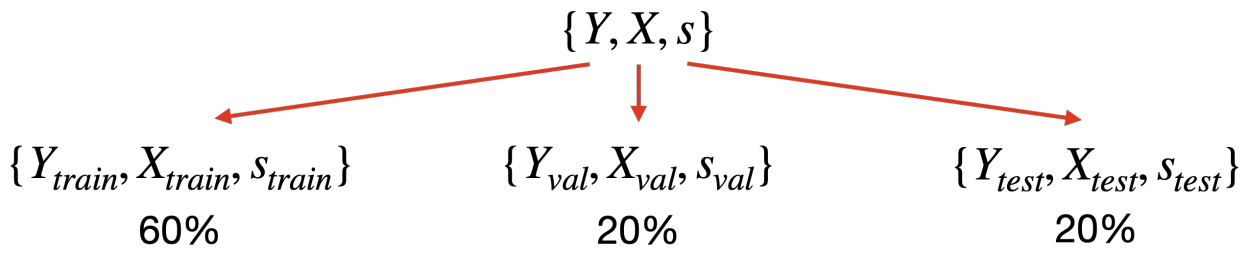


Figure 2: Data Splitting Visualization

2.3 Model Training

After preprocessing the data, `geospaNN` fits the model in two steps:

1. **Initialize the spatial parameters**.
2. **Fit NN-GLS**.

2.3.1 Parameter Initialization

Initializing the spatial parameters θ is necessary for training NN-GLS since L-BFGS-based likelihood maximum-likelihood estimation is employed to iteratively update θ. A reasonable initial guess is crucial to a successful training because the GLS-style loss function is sensitive to these spatial parameters.

Unless in special cases where initial value is available, users are encouraged to follow our default pipeline, where θ is initialized as the maximum likelihood estimator from NN-based residual `theta0` (can be obtained without any spatial parameters). In this example, we use a 3-layer multi-layer perceptron as the NN's architecture, with 100, 50, and 20 nodes for each layer and a scalar as the output.

```
mlp_nn = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
trainer_nn = geospaNN.nn_train(mlp_nn, lr=0.01, min_delta=0.001)
training_log = trainer_nn.train(data_train, data_val, data_test, seed = 2025)
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                               data_train.pos, neighbor_size=20)
```

2.3.2 Fit NN-GLS

With `theta0` being the initial value, the model is fitted as following:

```
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 100),
    torch.nn.ReLU(),
    torch.nn.Linear(100, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1),
)
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls,
                       theta=torch.tensor(theta0))
trainer_nngls = geospaNN.nngls_train(model, lr=0.1, min_delta=0.001)
training_log = trainer_nngls.train(data_train, data_val, data_test, epoch_num= 200,
                                   Update_init=10, Update_step=2, seed = 2025)
```

The code chunk consists of 4 steps:

1. **Step 1**: Define the non-spatial architecture. Here we defined a 3-layer multi-layer perceptron (MLP) through `torch.nn.Sequential`).
2. **Step 2**: Define the NN-GLS model with the MLP defined in step 1 and an initialized exponential Gaussian covariance structure specified by `theta0` using `geospaNN.nngls`, which requires:
   - Number of covariates (`p`).
   - Number of nearest neighbors (`neighbor_size`).
   - Dimension of spatial coordinates (`coord_dimensions`).
   - Nonspatial architecture (`mlp_nngls`).
   - Spatial parameters (`theta`).
3. **Step 3**: Define a trainer for NNGLS through `geospaNN.nngls_train`, which requires:
   - Learning rate (`lr`): we recommend values in the range of $[0.01, 0.1]$ for our default ADMM optimizer.
   - Minimum value of validation loss drop (`min_delta`): any decrease of validation loss from last epoch larger than this value will be recorded as "significant gain." If no significant gain happens in the last few epochs, early stopping will be triggered.

4. **Step 4**: Run the trainer with the dataloaders, which requires:
    - Dataloaders for training, validation, and testing (`data_train`, `data_val`, `data_test`).
    - Maximum number of epochs (`epoch_num`).
    - Initial epoch for spatial parameter estimation (`Update_init`).
    - Number of epochs between two spatial parameter updates (`Update_step`).

2.3.3 Iterative Parameter Update

We note that in training NN-GLS, we isolate the updating of spatial parameters θ from that of the other weights parameters in the neural network architecture and update them iteratively as is demonstrated in figure 3. We control the frequency and starting point of updating θ to improve the efficiency and stability of the training. In general, the properties of a SPMM are primarily governed by its spatial structure (i.e., θ). If θ is updated while the non-spatial estimates remain unstable and biased, the newly estimated θ will be substantially influenced by these inaccuracies. This can lead to a feedback loop in which θ progressively deviates from its true value, ultimately resulting in non-convergence.

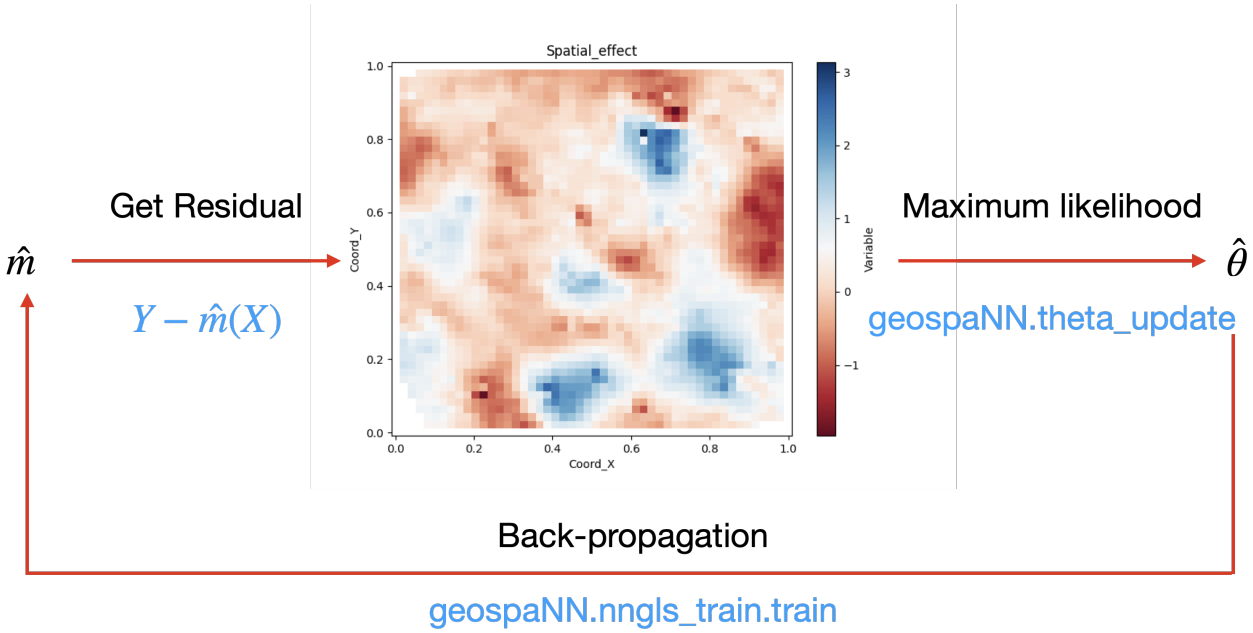


Figure 3: Iterative Parameter Update

2.4 Result Output

Our pipeline returns the outputs in the following aspects:

- The performance of training.
- The model-related performance, including prediction at new locations and mean effect estimation.
- Visualization tools for the functions mentioned above.

2.4.1 Performance of Training

To evaluate how `geospaNN` performed in the training process, use `geospaNN.plot_log` which requires:

- Optional true spatial parameters (`theta`).
- Path to save the figure (`path`).

```
geospaNN.plot_log(training_log, theta, path, save = True)
```

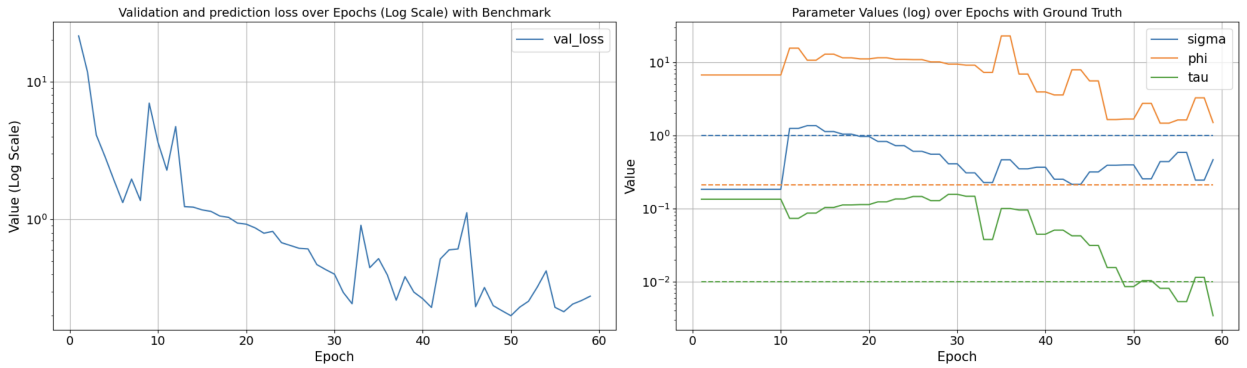


Figure 4: Training Log

2.4.2 Mean Function Estimation

Mean function estimation corresponds to estimating the $m(x)$ term in SPMM model, which represents the non-spatial relationship between Y and covariates X. To obtain the mean function estimation for covariates `X`, use the following method:

```
estimate = model.estimate(X)
```

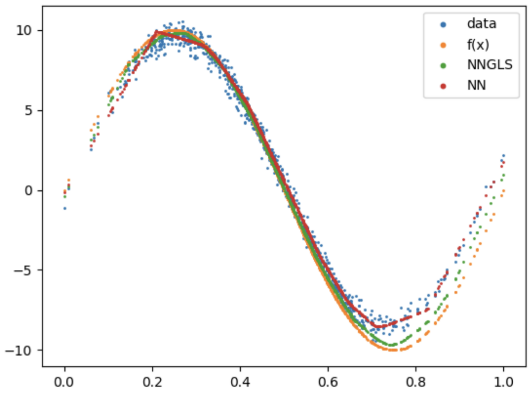


Figure 5: Mean Function Estimation

Note that for the NN-GLS model, the mean function is estimated by the non-spatial architecture `mlp_nngls`, or can be equivalently called by `model.mlp`. Thus, `model.estimate(X)` is equivalent to `model.mlp(X)` and `mlp_nngls(X)`. Additional code to produce the figure used the `matplotlib` library and is shown in Appendix chunk 1.

2.4.3 Prediction and Intervals

To predict, use `geospaNN.predict()` as follows:

```
[test_predict, test_PI_U, test_PI_L] = model.predict(data_train, data_test, PI = True)
```

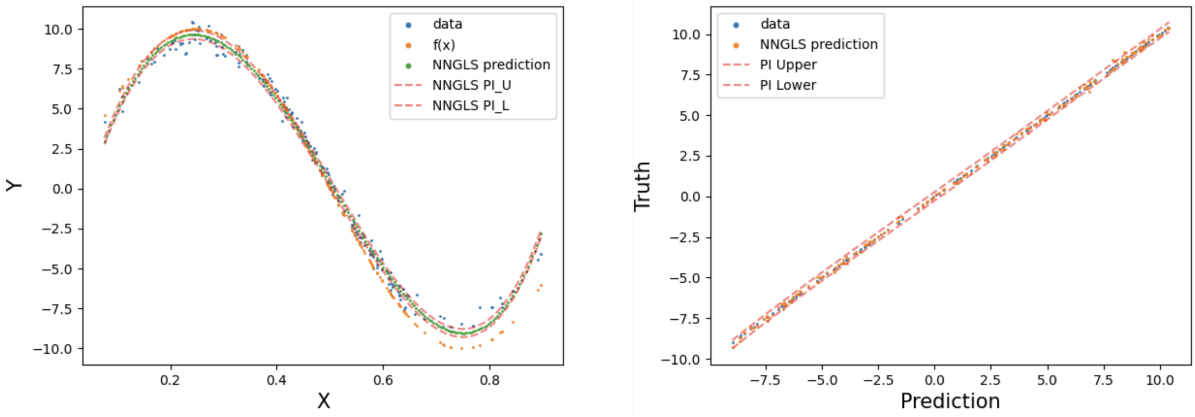


Figure 6: Prediction

NNGLS provides prediction and 95% prediction intervals by kriging. For new location(s) s_{new} and covariates $X(s_{new})$, prediction is obtained by evaluating the mean function estimate at the new covariate value:

$$\hat{Y}(s_{new}) = \hat{m}\big(X(s_{new})\big) + \widehat{w(s_{new}) + \epsilon}(s_{new}).$$

The first term is computed by evaluating the mean function estimate at the new covariate value $X(s_{new})$. The second term is assumed a spatial process and incorporates uncertainty. In our model, conditioning on the training set, the spatial effect on new locations follows a Gaussian distribution whose mean and variance are obtained through kriging. Nearest neighbor kriging (NNGP approximation) is utilized here to guarantee the scalability of prediction. Additional code to produce the figure used the `matplotlib` library and is shown in Appendix chunks 2 and 3.

2.4.4 Partial Dependency Plot

When the covariates are multi-dimensional, `geospaNN` offers plots of partial dependence functions summarizing the marginal contribution of each covariate to the mean function. To illustrate this, we consider a simulation scenario where the mean is specified as a 5-dimensional Friedman's function and run the same pipeline, the PDP can be generated by using `geospaNN.visualize.plot_PDP_list`.

```
geospaNN.visualize.plot_PDP_list([funXY, mlp_nngls, mlp_nn],
                                 ['Friedmans function', 'NNGLS', 'NN'],
                                 X, split = True)
```

Partial Dependency Plot (PDP) is a commonly used visualization tool in the machine learning community to illustrate the marginal effect of a single covariate on the response for a high-dimensional model. Given a multi-dimensional function $m(X)$ and one covariate X_i, its PDP is generated by numerically integrating out the other covariates via:

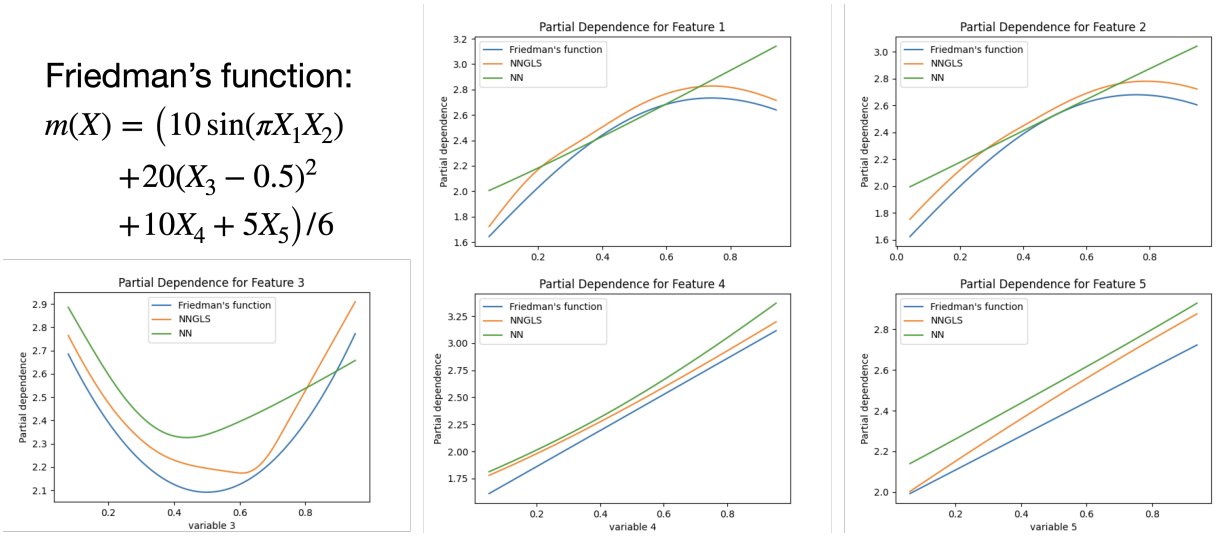


Figure 7: PDP

$$PD(m, X_i) = \int m(X_1, \cdots, X_p) P(X_{-i}) dX_{-i}.$$

Note that PDPs are only needed when the covariate dimension is greater than one. So the PDP function in `geospaNN` only works for multui-dimensional covariates and will not work for the 1-D example in this vignette. A full PDP script is available at https://abhirupdatta.github.io/geospatial_stats_ML_short_course_2024/lec_code/lec4_ibc/PDP.ipynb for more details.

2.5 Additional Examples

2.5.1 Spatial Linear Mixed Model `geospaNN` also provides efficient solution to the spatial linear mixed model (SPLMM), which is a special case of SPMM, where the mean function is assumed to be linear:

$$Y(s) = X(s)\beta + w(s) + \epsilon(s).$$

In `geospaNN`, similar to that of NN-GLS, NNGP approximation is also used to simplify the computation and keep the linear complexity ($O(n)$) of solving SPLMM. To apply SPLMM, use the function `geospaNN.linear_gls` on the dataset constructed by `geospaNN.make_graph` with the same arguments introduced in section 2.2.3:

```
model_linear = geospaNN.linear_gls(data_train)

estimate = model_linear.estimate(X)
[test_predict, test_PI_U, test_PI_L] = model_linear.predict(data_train, data_test, PI = True)
#### To get estimated covariance of linear coefficients, call:
model_linear.var
```

`model_linear` is in the same class as the NN-GLS `model`, both estimation and prediction (in figure 7) can be obtained using `model_linear.estimate()` and `model_linear.predict()`. Additionally, the function estimates the covariance matrix of linear coefficients β, which is contained in the additional attribute `model_linear.var`. SPLMM naturally extends the linear mixed model and is the mostly widely used model for data with spatial structure. The implementation of SPLMM here is equivalent to the R-packages BRISC (Saha and Datta 2018) and should be an optimal choice for the python users if efficient solution is wanted for large geospatial dataset. Additional code to produce the figure is provided in Appendix chunk 4.

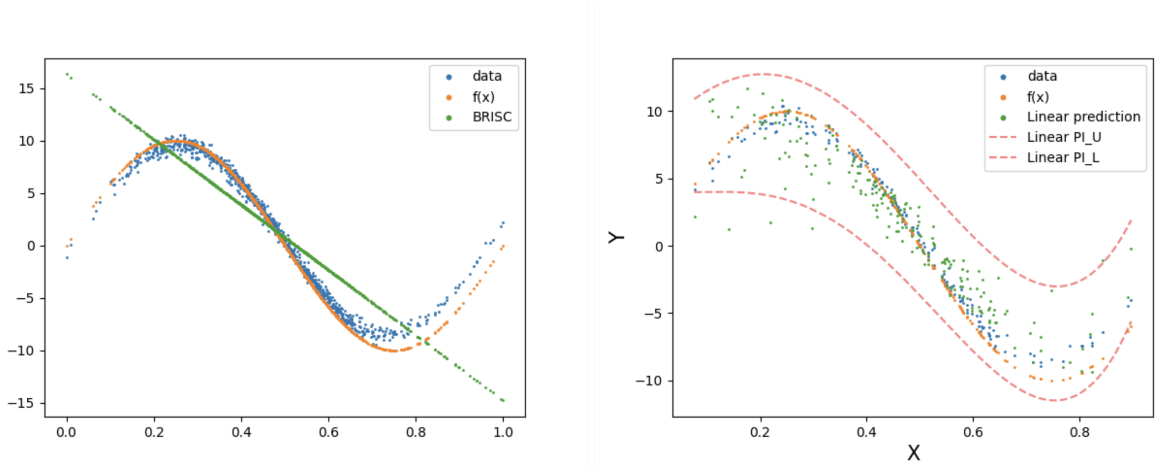


Figure 8: Estimation and Prediction of SPLMM

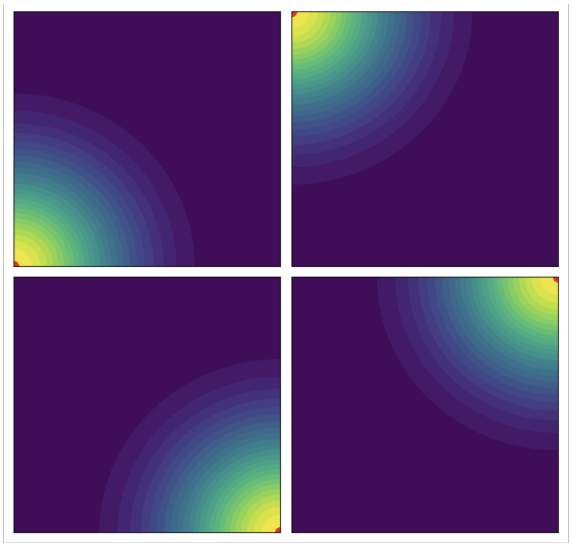
2.5.2 NN-GLS with added spatial features An alternative strategy to solve spatial mixed model is to add spatial co-ordinates or some transformations (such distances or basis functions) as additional covariates. These added-spatial-features models incorporate all spatial information into the mean function. As a result, they are not able to separate non-spatial and spatial effect and can only be used for prediction. In `geospaNN`, we provide easy method for the added-spatial-features style neural networks, some of which have been proposed in literature but not formally implemented in any software.

To simply add spatial coordinates to the covariates X, use concatenation:

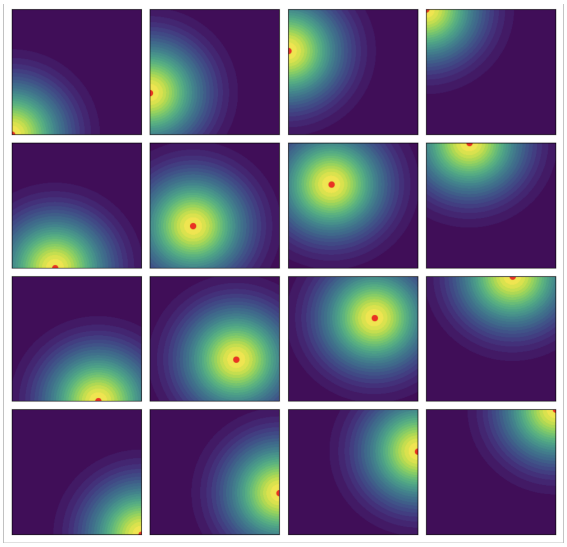
```
K = 2

data_add_train, data_add_val, data_add_test =
      geospaNN.split_data(torch.concat([X, coord], axis = 1),
      Y, coord, neighbor_size=nn,test_proportion=0.2)
```

Where `K` is the additional dimension brought by the spatial features. We also implemented 2-D radial basis functions using Wendland kernel as a more complex type of spatial feature. This spline-based approach was taken in a recent work DeepKriging (Chen et al. 2020) and is closely related to kriging and its associated variants such as fixed rank kriging. `geospaNN.coord_basis` can be used to generate the basis functions, where the user need to specify the resolutions for each set of knots by a list of integers. For example, `4 ** 2` in `num_basis` means the knots for the second of basis functions are on the 4×4 grid within a unit square (see the figure below for an illustration of $2x2$ and $4x4$ knots).



Wendland RBFs at 2×2 Knots



Wendland RBFs at 4×4 Knots

```
K, coord_basis = geospaNN.coord_basis(coord, num_basis = [2 ** 2, 4 ** 2, 6 ** 2])

data_add_train, data_add_val, data_add_test =
      geospaNN.split_data(torch.concat([X, coord_basis], axis = 1),
      Y, coord, neighbor_size=nn,test_proportion=0.2)
```

No matter which of the added-spatial-features approach is taken, the same code for training the non-spatial neural networks (section 2.3.1) can be used for added-spatial-features neural networks:

```
mlp_nn_add = torch.nn.Sequential(
    torch.nn.Linear(p+K, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
nn_add_model = geospaNN.nn_train(mlp_nn_add, lr=0.01, min_delta=0.001)
training_log = nn_add_model.train(data_add_train, data_add_val, data_add_test, seed = 2025)
```

and the prediction can be obtained by the mean function estimation of the model:

```
predict_nn_add = mlp_nn_add(data_add_test.x).reshape(-1)
```

In example provided for this section, the MSE of prediction on test set for the first approach, i.e. adding coordinates as covariates, is 2.84, while for the second approach using basis functions is 1.70. For comparison, the MSE for NN-GLS is 0.45, which illustrates the advantage of Gaussian process-based modeling. Additional code to produce the figures is provided in Appendix chunk 5.

2.5.3 Application on time series analysis The applications of NN-GLS are not restricted to geospatial data with correctly specified dependency structure. It was shown in Zhan and Datta (2024) that NN-GLS works well in scenarios where the covariance structure and data generating process are misspecified. To demonstrate how user can easily generate dependent data and apply NN-GLS on it with `geospaNN`, we

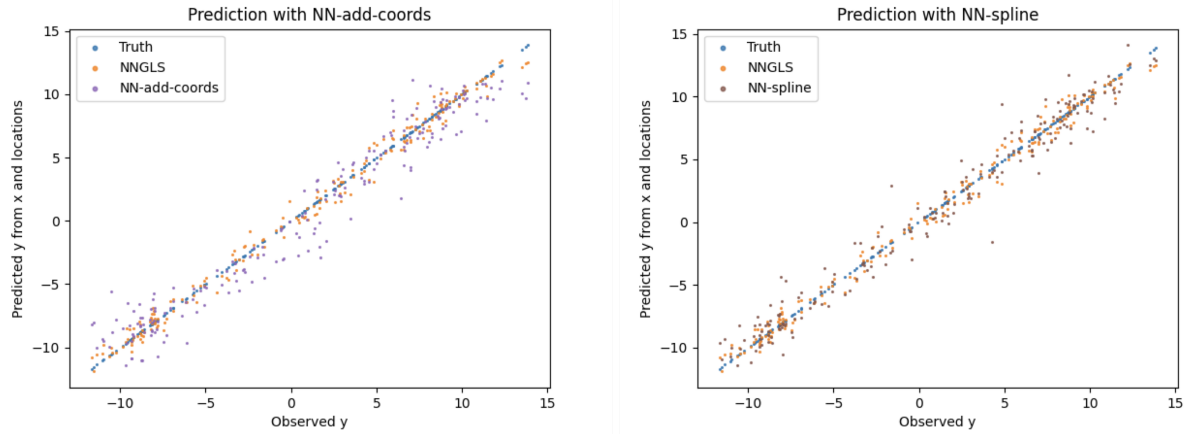


Figure 9: Prediction of added-spatial-features approaches vs NN-GLS

provide a time series example in this section. In this example, the mean function is still $m(x) = 10\sin(2\pi x)$ and the noise is generated from an auto regressive (AR1) process, one of the most popular models used to account for temporal effect in time series analysis.

$$w(t) = \rho w(t-1) + \epsilon(t)$$

Given the arguments and data generated in section 2.2.1, the following steps are taken to generate the time series data:

1. Specifying parameters for the AR(1) process, which include:
   - AR(1) coefficient `rho`.
   - Standard deviation of noise `sigma_AR`.
   - Initial value `x_0`
2. Simulate white noise ϵ (`epsilon`).
3. Initialize and generate the AR(1) process `corerr`. Define the time points by `coord` to fit into the previous training process.
4. Follow the other steps in section 2.3 to train NN-GLS.

```
#1
rho = np.sqrt(0.7)
sigma_AR = 5
x0 = 0
#2
np.random.seed(2024)
epsilon = np.random.normal(0, sigma_AR, n)
#3
corerr = np.zeros(n)
corerr[0] = x0
for t in range(1, n):
    corerr[t] = rho * corerr[t-1] + epsilon[t]
coord = torch.zeros((n,2))
coord[:,0] = torch.tensor(range(n))/100
```

The AR(1) process (random effect) with the coefficient $\rho^2 = 0.7$. The estimation performance of NN-GLS are illustrated with the code provided in Appendix chunk 6. The MSE of mean-function estimations are 0.556 and 0.958 for NN-GLS and NN respectively.

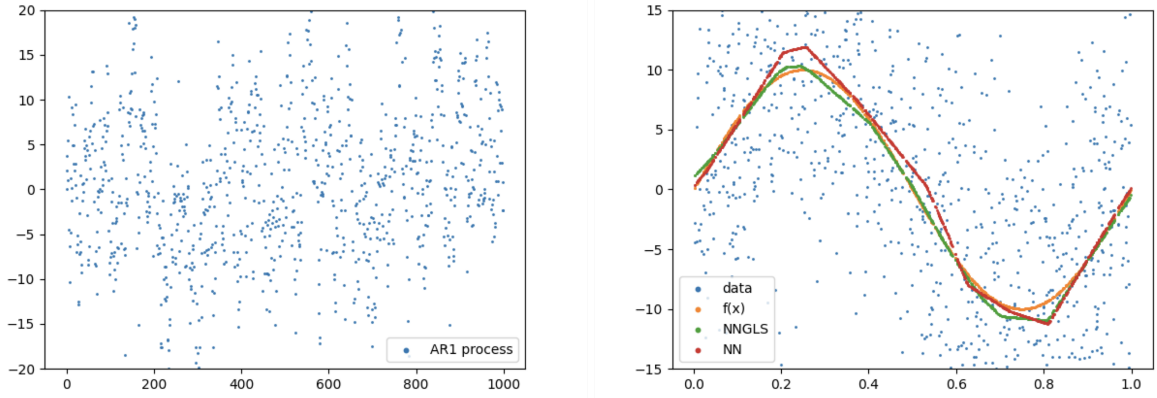


Figure 10: AR(1) process and the estimation performance

Appendix

Chunk 1

```
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='NNGLS')
plt.scatter(X.detach().numpy(), mlp_nn(X).detach().numpy(), s=1, label='NN')
lgnd = plt.legend()
```

Chunk 2

```
x_np = data_test.x.detach().numpy().reshape(-1)
x_smooth = np.linspace(x_np.min(), x_np.max(), 200)  # Create finer x-points
degree = 4
U_fit = np.polyfit(x_np, test_PI_U, degree)
L_fit = np.polyfit(x_np, test_PI_L, degree)
Pred_fit = np.polyfit(x_np, test_predict, degree)

# Evaluate the polynomial on a smooth grid
y_smooth_U = np.polyval(U_fit, x_smooth)
y_smooth_L = np.polyval(L_fit, x_smooth)
y_smooth = np.polyval(Pred_fit, x_smooth)

plt.clf()
plt.scatter(data_test.x.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.x.detach().numpy(), funXY(data_test.x.detach().numpy()), s=1, label='f(x)')
plt.scatter(data_test.x.detach().numpy(), test_predict.detach().numpy(), s=1, label='NNGLS prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='NNGLS PI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='NNGLS PI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
```

```
plt.ylabel("Y", fontsize=15)
lgnd = plt.legend()
```

Chunk 3

```
plt.clf()
plt.scatter(data_test.x.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.x.detach().numpy(), funXY(data_test.x.detach().numpy()), s=1, label='f(x)')
plt.scatter(x_smooth, y_smooth, s=1, label='NNGLS prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='NNGLS PI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='NNGLS PI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
plt.ylabel("Y", fontsize=15)
lgnd = plt.legend()
```

Chunk 4

```
estimate = model_linear.estimate(X)
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='BRISC')
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])

[test_predict, test_PI_U, test_PI_L] = model_linear.predict(data_train, data_test, PI = True)
x_np = data_test.x.detach().numpy().reshape(-1)
x_smooth = np.linspace(x_np.min(), x_np.max(), 200)  # Create finer x-points
degree = 4
U_fit = np.polyfit(x_np, test_PI_U, degree)
L_fit = np.polyfit(x_np, test_PI_L, degree)
Pred_fit = np.polyfit(x_np, test_predict, degree)

# Evaluate the polynomial on a smooth grid
y_smooth_U = np.polyval(U_fit, x_smooth)
y_smooth_L = np.polyval(L_fit, x_smooth)
y_smooth = np.polyval(Pred_fit, x_smooth)

plt.clf()
plt.scatter(data_test.x.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.x.detach().numpy(), funXY(data_test.x.detach().numpy()), s=1, label='f(x)')
plt.scatter(data_test.x.detach().numpy(), test_predict.detach().numpy(), s=1, label='Linear prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='Linear PI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='Linear PI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
plt.ylabel("Y", fontsize=15)
lgnd = plt.legend()
for handle in lgnd.legend_handles[:3]:
    handle.set_sizes([10.0])
```

Chunk 5

```
plt.clf()
plt.scatter(data_test.y.detach().numpy(), data_test.y.detach().numpy(), s=1.5, alpha = 0.8, label='Trutl
plt.scatter(data_test.y.detach().numpy(), predict_nngls, s=1.5, alpha = 0.8, label='NNGLS')
plt.scatter(data_test.y.detach().numpy(), predict_nn_add, s=1.5, alpha = 0.8, label=label)
lgnd = plt.legend(fontsize=10)
plt.xlabel('Observed y', fontsize=10)
plt.ylabel('Predicted y from x and locations', fontsize=10)
plt.title('Prediction with NN-add-coords')

for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
```

Chunk 6

```
plt.clf()
plt.scatter(range(n), corerr, s=1, label='AR1 process')
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])

estimate = model.estimate(X)
plt.clf()
plt.scatter(X.detach().numpy(), Y.detach().numpy(), s=1, label='data')
plt.scatter(X.detach().numpy(), funXY(X.detach().numpy()), s=1, label='f(x)')
plt.scatter(X.detach().numpy(), estimate, s=1, label='NNGLS')
plt.scatter(X.detach().numpy(), mlp_nn(X).detach().numpy(), s=1, label='NN')
lgnd = plt.legend()
for handle in lgnd.legend_handles:
    handle.set_sizes([10.0])
```


Chen, Wanfang, Yuxiao Li, Brian J Reich, and Ying Sun. 2020. "Deepkriging: Spatially Dependent Deep Neural Networks for Spatial Prediction." *arXiv Preprint arXiv:2007.11972*.

Datta, Abhirup, Sudipto Banerjee, Andrew O Finley, and Alan E Gelfand. 2016. "On Nearest-Neighbor Gaussian Process Models for Massive Spatial Data." *Wiley Interdisciplinary Reviews: Computational Statistics* 8 (5): 162–71.

Saha, Arkajyoti, and Abhirup Datta. 2018. "BRISC: Bootstrap for Rapid Inference on Spatial Covariances." *Stat* 7 (1): e184.

Zhan, Wentao, and Abhirup Datta. 2024. "Neural Networks for Geospatial Data." *Journal of the American Statistical Association*, no. just-accepted: 1–21.