Vignette for GeospaNN

Wentao Zhan

1 Summary

GeospaNN stands for **Geospa**tial **N**eural **N**etworks, is a package implements NN-GLS, a geographically-informed Graph Neural Network (GNN) architecture. This file provides a brief introduction on how to use geospaNN for geospatial data analysis. The vignette is devided into three parts:

- Data preparation: This part mainly contains features that transform the raw geospatial data into the input format for Neural Network trainer. A geospatial simulation function is also introduced to conduct general statistical experiments.
- Modeling training: This part contains the main functions to train the model. To wrap up the common training procedure, a user-friendly trainer is introduced for NN-GLS.
- Model evaluation: This part visualizes the training process as well as the spatial parameter estimation. Additionally, it provides functions for mean-effect estimation and new-location prediction, which are the main goals of the package and can lead to multiple geospatial applications.

Besides this vignette, GeospaNN's official website is live, including an installation guide, a comprehensive documentation, and several statistical experiments. The demo code for this vignette is available here.

2 Main content

2.1 Spatial mixed effect model and goal

In this vignette, we focus on spatial mixed effet model, which is formulated as:

$$Y(s) = m\big(X(s)\big) + w(s) + \epsilon(s), \tag{1}$$

where $X(s)$ are the spatial covariates at location s, $Y(s)$ is the observed variable, ϵ is the i.i.d random Gaussian noise follows $N(0, \tau\sigma^2)$, which is also called the nugget in spatial literature. $w(s)$ represents a stochastic process accounting for the spatial correlation, which is further assumed to be an mean-zero exponential Gaussian Process (GP) with the following covariance structure:

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

θ is introduced as a general notation for (σ^2, ϕ, τ).

Under this setting, geospaNN aims to estimate the non-linear mean effect $m(\cdot)$ by $\widehat{m}(\cdot)$ falling in 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 $\widehat{m}(\cdot$ and the kriging prediction $\widehat{w}(s_{new})$. Nearest Neighbor Gaussian Process (NNGP) (Datta et al., 2016), as an approximate process to the full GP, is used to guarantee the scalability of geospaNN, enabling the analysis on geospatial data set with up to 1 million observations.

2.2 Simulation and data preprocessing

Simulation

Simulation is a key function in geospaNN which generates desired geospatial datasets for various experiments and facilities methodology development with flexibility and efficiency. However, simulating spatially correlated data usually requires $O(n^3)$ running time by obtaining the Cholesky's factor of a $n \times n$ covariance matrix and multiplying it to i.i.d random effect. Fortunately, NNGP not only provides efficient precision matrix computation, but also efficient Cholesky's factorization. In geospaNN, for large sample size, we use NNGP to obtain an $O(n)$ time complexity.

Here, we simulate a data from model 1, where the function (`funXY`):

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

the spatial effect $w(s)$ is generated as a exponential Gaussian process with spatial variance $\sigma^2 = 1$ (`sigma`)and spatial correlation decay $\phi = 3\sqrt{2}/20$ (`phi`); and $\epsilon(s)$ is the i.i.d random noise with variance $\eta^2 = \tau * \sigma^2 = 0.01$ (`tau`). The other required parameters includes:

- The desired sample size (`n`).
- The number of covariates (`p`).
- Number of neighbors used for NNGP approximation (`nn`).
- Range of spatial locations (`range`). The locations are sampled from the 2-D domain $[\texttt{range}]^2$

The coordinates `X`, response `Y`, coordinates `coord`, covariance matrix `cov`, and random effect $w(s) + \epsilon(s)$ `corerr` are returned in tensor formats.

```
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])
```

In practice, usually the covariate X it self is spatially correlated (denoted as $X(s)$), a short-cut to simulating in this setting is to first use the `geospaNN.Simulation` to simulate one correlated spatial process `corerr` as $X(s)$, and another `corerr` as $w(s) + \epsilon(s)$. Y can be then assembled as $m\big(X(s)\big) + w(s) + \epsilon(s)$. The process is illustrated in the next few code chunks and figure 1 visualizes the simulated data.

```
torch.manual_seed(2025)
_, _, _, _, 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())
```

```
torch.manual_seed(2025)
_, _, coord, cov, corerr = geospaNN.Simulation(n, p, nn, funXY, theta, range=[0, 1])
```

```
Y = funXY(X).reshape(-1) + corerr
```

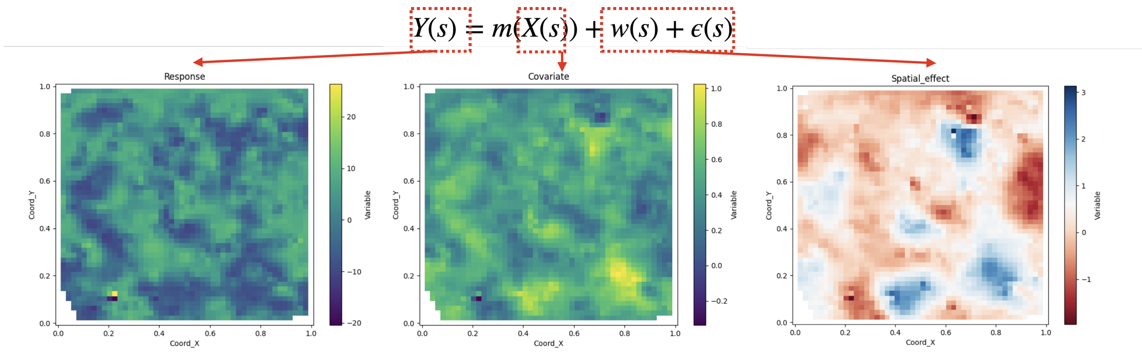


Figure 1: Simulated data

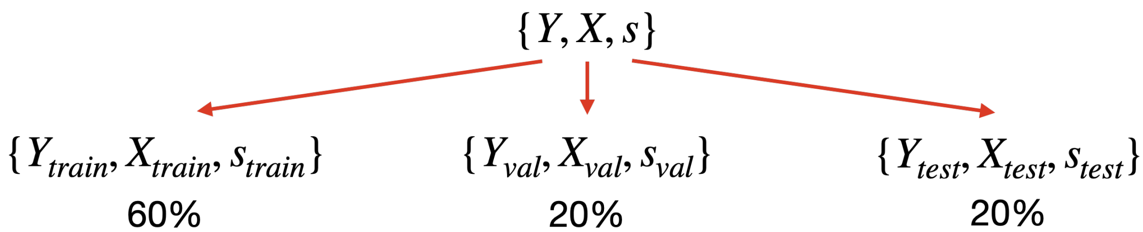
Data preprocessing

Covariates $X(n \times p)$, response $Y(n \times 1)$, coordinates $s(n \times 2)$ are the fundamental elements in geospatial dataset. However, to wrap the dataset into a recognizable format to GNN module, a directed graph has to be constructed beforehand through function `geospaNN.make_graph`. This function creates a `DataLoader` object, which is used for efficient data loading and management. By default, a k-nearest-neighbor graph will be constructed, with k specified by `nn`. Users are also allowed to customize the graph using a $n \times k$ index tensor `Ind_list`, where the ith row contains the index list for nodes connected to s_i, and -1 will stand for non-connection.

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

Besides a directed graph, training-validation-testing split is also easily achieved by `geospaNN.split_data`. Similar to `geospaNN.make_graph`, covariates X, response Y, coordinates s, neighbor size, are the inputs. `val_proportion` and `test_proportion` can be customized for flexible split proportions and are both set to 0.2 as default. The output are three dataloaders that are required by the training process.

```
torch.manual_seed(2024)
np.random.seed(0)
data_train, data_val, data_test = geospaNN.split_data(X, Y, coord, neighbor_size=nn, test_proportion=0.2)
```



2.3 Model training

After preprocessing the data, geospaNN fits the model with two steps,

- Initialize the spatial parameters.
- Fit NN-GLS.

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).

Fit NN-GLS

With `theta0` being the initial value, the model is fitted as following: The code chunk consists of 4 steps:

```
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)
```

```
torch.manual_seed(2024)
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)
```

- Step 1: Define the non-spatial architecture. Here we defined a 3-layer multi-layer perceptron (MLP) through `torch.nn.Sequential`).
- Step 2: Define the NN-GLS model with the MLP defined in step 1 and a 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`).
- 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.

- 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`).

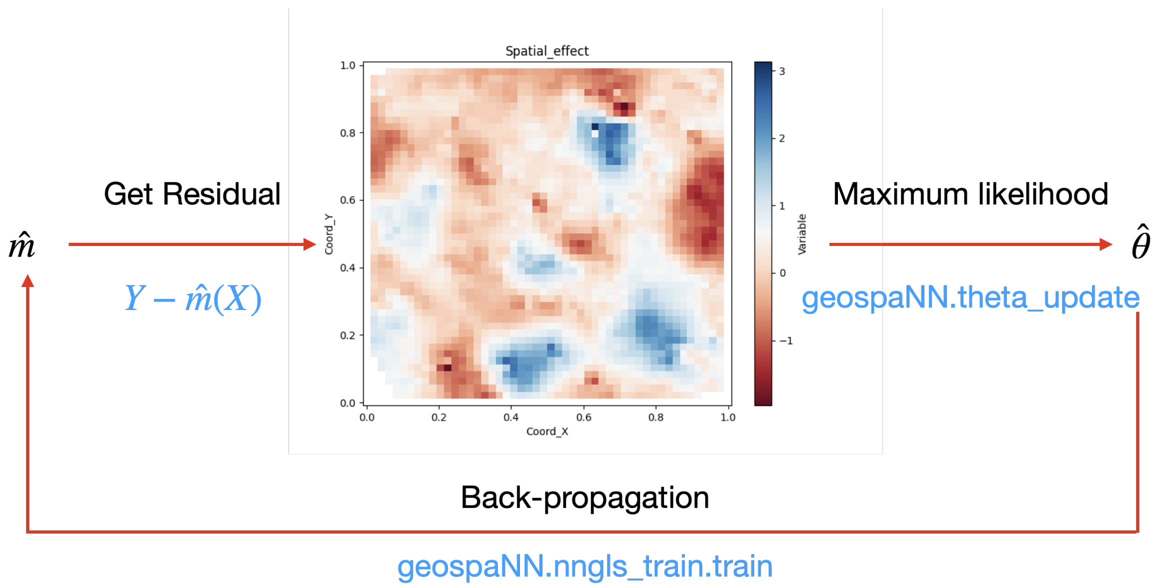


Figure 2: Iterative parameter update

We note that in training NN-GLS, we isolate the spatial paramaters θ's update from the other weights parameters in the non-spatial architecture and update them iteratively as is demonstrated in figure 2. We control the frequency and starting point of updating θ to improve the efficiency and stability of the training. In general cases, the property of a SPMM is dominated by its spatial structure (i.e. θ). If θ is updated when the non-spatial estimation is still unstable and biased, the new θ will be significantly affected, a vicious cycle will form as θ deviating further from the truth and cause non-convergence.

2.4 Result output

Our pipeline returns the outputs in the following aspects:

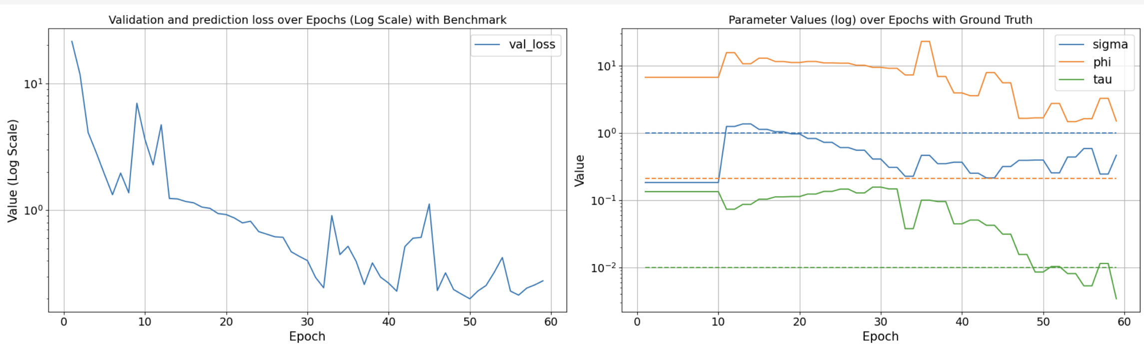
- The performance of training.
- The model-related performance, including prediction at new locations and mean effect estimation.
- Visualization tools for the functions mentioned above.

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`).

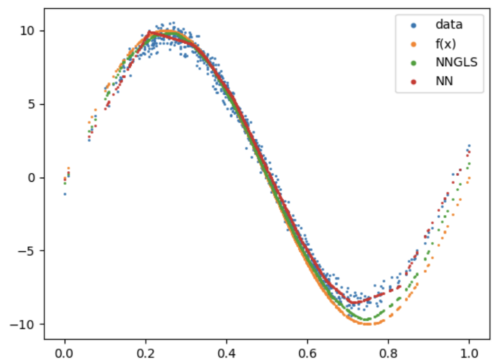
```
geospaNN.plot_log(training_log, theta, path)
```



Mean function estimation

Mean function estimation corresponds to the $m(x)$ term in model 1, which represents the non-spatial relationship between Y and covariates X. To obtain the mean function estimation for covariates `X`, use the following method: Note that for NN-GLS model, the

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



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 `matplotlib` library and is shown in chunk 3.

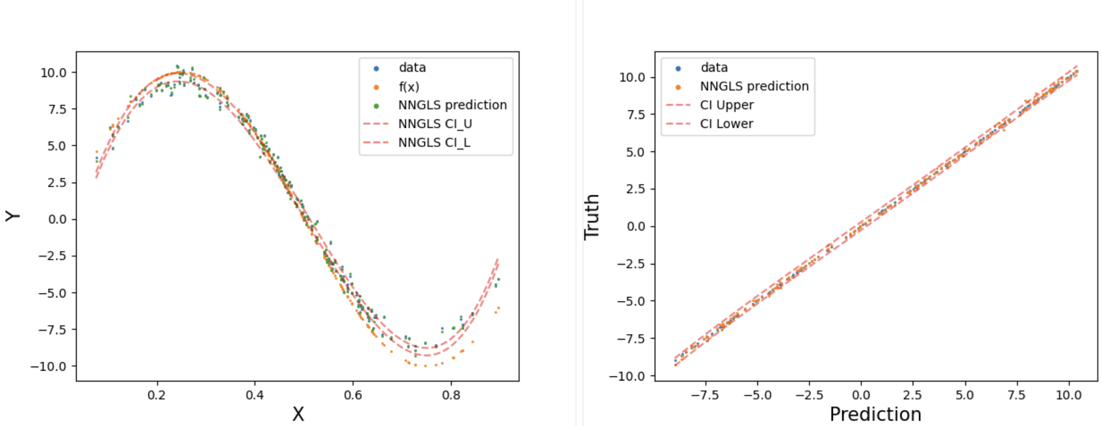
Prediction and intervals

To predict, use `geospaNN.predict()` as follows. NNGLS provides prediction and 95% confidence intervals by kriging. For new location(s) s_{new} and covariates $X(s_{new})$, prediction is obtained as

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

The first term is deterministic and computed as mean function estimation. 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

```python
[test_predict, test_CI_U, test_CI_L] = model.predict(data_train, data_test, CI = True)
```



approximation) is utilized here to guarantee the scalability of prediction. Additional code to produce the figure used `matplotlib` library and is shown in chunk 4.

Partial Dependency Plot

If we switch the 1-dimensional sine function used through this vignette to the 5-dimensional Friedman's function and run the same pipeline, the PDP can be generated by using `geospaNN.visualize.plot_PDP_list`. Partial Dependency Plot (PDP) is a commonly used visualization tool in machine learning community to illustrate the marginal effect of a single covariate on the response for 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

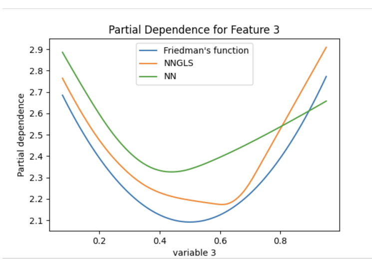
$$PD(m, X_i) = \int m(X_1, \cdots, X_p)P(X_{-i})dX_{-i}.$$

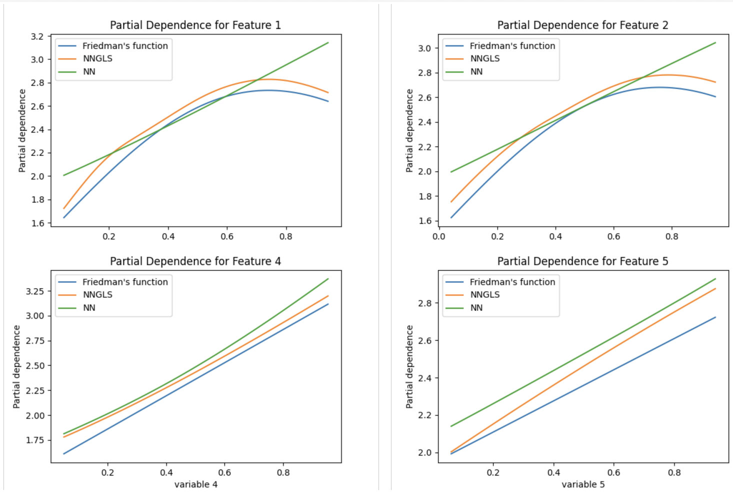
The code chunk above just provides an illustration of the usage, a full script is available here for download.

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

Friedman's function:

$$m(X) = \big(10\sin(\pi X_1 X_2) \\ +20(X_3 - 0.5)^2 \\ +10X_4 + 5X_5\big)/6$$





Appendix

```
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])
plt.savefig(path + 'Estimation.png')
```

Figure 3: Code to visualize the estimation result

```
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_CI_U, degree)
L_fit = np.polyfit(x_np, test_CI_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 CI_U', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='NNGLS CI_L', color = 'red', alpha = 0.5)
plt.xlabel("X", fontsize=15)
plt.ylabel("Y", fontsize=15)
```

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

plt.clf()
plt.scatter(data_test.y.detach().numpy(), data_test.y.detach().numpy(), s=1, label='data')
plt.scatter(data_test.y.detach().numpy(), test_predict.detach().numpy(), s=1, label='NNGLS prediction')
plt.plot(x_smooth, y_smooth_U, linestyle='--', label='CI Upper', color = 'red', alpha = 0.5)
plt.plot(x_smooth, y_smooth_L, linestyle='--', label='CI Lower', color = 'red', alpha = 0.5)
plt.xlabel("Prediction", fontsize=15)
plt.ylabel("Truth", fontsize=15)
```

Figure 4: Code to visualize the prediction result

References

Datta, A., Banerjee, S., Finley, A. O. and Gelfand, A. E. (2016), 'On nearest-neighbor Gaussian process models for massive spatial data', *Wiley Interdisciplinary Reviews: Computational Statistics* **8**(5), 162–171.