STATISTICAL AND MACHINE LEARNING FOR BIG GEOSPATIAL DATA: Part IV b

Wentao Zhan
Johns Hopkins University
Department of Biostatistics

Overview of Part IV b

Introduction to geospaNN

Basic features of geospaNN

Simulation examples of geospaNN

- General Architecture design
- NNGLS handles complex interaction
- NNGLS vs added-spatial-features approaches

Real data application of geospaNN

GeospaNN

Stands for: Geospatial Neural Networks.

GeospaNN   Search

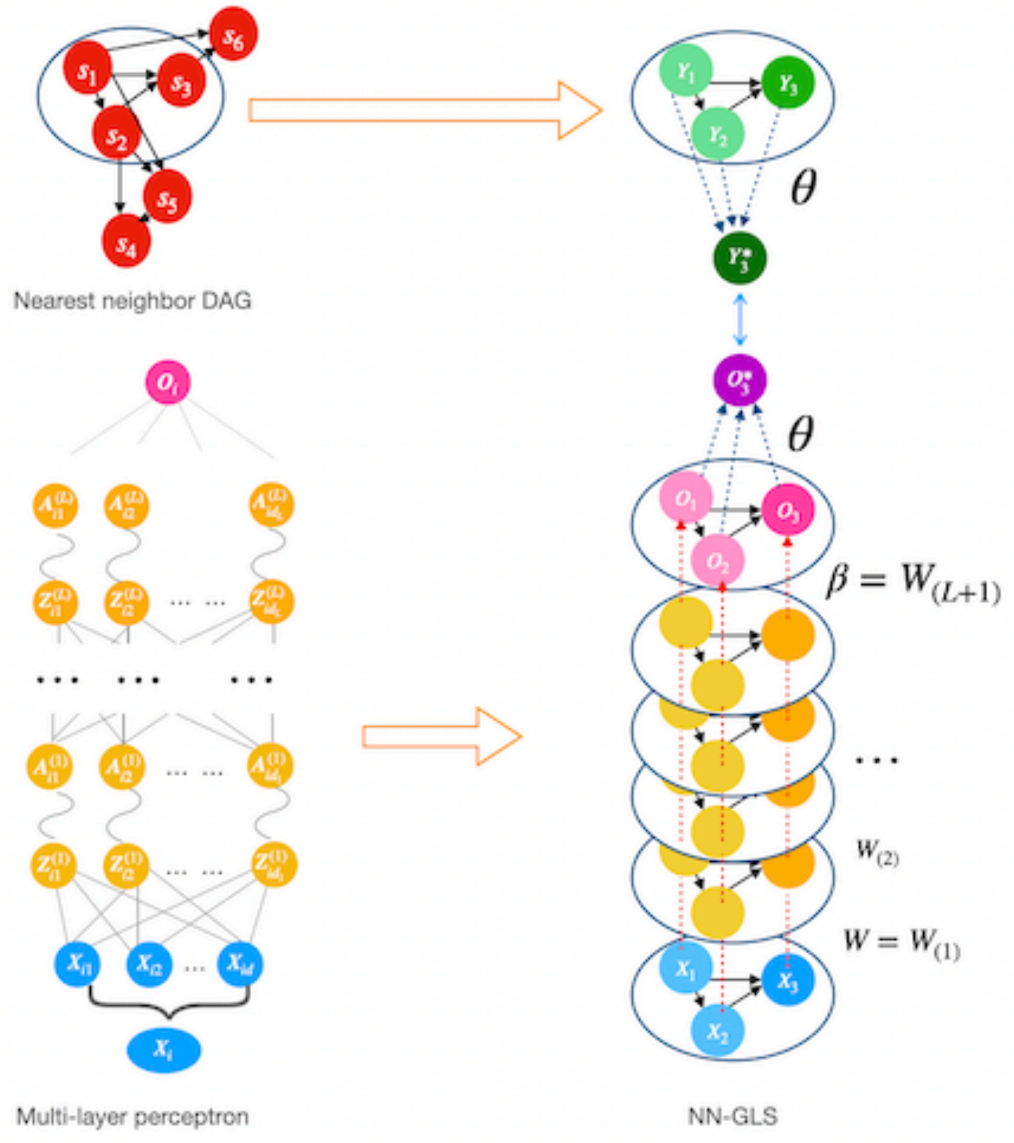
GeospaNN
Home
Overview
How to start
Examples
Documentation

GeospaNN - Neural networks for geospatial data

Authors: Wentao Zhan (wzhan3@jhu.edu), Abhirup Datta (abhidatta@jhu.edu)

A package based on the paper: Neural networks for geospatial data

GeospaNN is a formal implementation of NN-GLS, the Neural Networks for geospatial data proposed in Zhan et.al (2023), that explicitly accounts for spatial correlation in the data. The package is developed using PyTorch and under the framework of PyG library. NN-GLS is a geographically-informed Graph Neural Network (GNN) for analyzing large and irregular geospatial data, that combines multi-layer perceptrons, Gaussian processes, and generalized least squares (GLS) loss. NN-GLS offers both regression function estimation and spatial prediction, and can scale up to sample sizes of hundreds of thousands. Users are welcome to provide any helpful suggestions and comments.



Pypi: https://pypi.org/project/geospaNN/

Outline

1. **Basic features**

2. Simulation

3. Real data example

Start point: what we have?

Point process:

- Data: $(Y_i, X_i, s_i) \; : i = 1, \ldots, n$
  - Y_i : scalar response
  - X_i : d-dimensional covariate
  - s_i : location

air pressure	relative humidity	U-wind	V-wind	PM 2.5	longitude	latitude
0.887664	0.774197	0.868530	0.781498	5.020834	0.980311	0.906268
0.882153	0.751742	0.864206	0.770715	3.837500	0.983093	0.889762
0.928359	0.714189	0.697080	0.813224	2.041666	0.974238	0.814722
0.954218	0.690767	0.625266	0.868161	3.669444	0.976951	0.798275
0.893599	0.685830	0.688808	0.842395	1.020833	0.945193	0.800337
...	...	...	...	...	...	...

Simulate data: input

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

p = 1;
funXY = f1

n = 1000
nn = 20
range = [0,1]

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

$$Y_i(s) = m\left(X_i(s)\right) + w(s) + \epsilon(s)$$

- "p": Dimension of the input.
- "funXY": 1-D function $m : X \in R^p \to Y \in R$
  - $Y = m(X) = 10\sin(2\pi X),\ X \in R$
  - $Y = m(X) = \frac{1}{6}\left(10\sin(\pi X_1 X_2) + 20(X_3 - 0.5)^2 + 20X_4 + 5X_5\right),\ X \in [0,1]^5$

Simulate data: geospaNN.Simulation

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

p = 1;
funXY = f1

n = 1000
nn = 20
range = [0,1]
```

+

```
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=range)
```

$$Y_i(s) = m(X_i(s)) + w(s) + \epsilon(s)$$

- “n”: Number of spatial locations.
- “nn”: Number of nearest neighbors used for NNGP approximation. 20 recommended.
- “theta”: Spatial parameters in $Cov(s, t) = \sigma^2(\exp(-\phi\,|\,s - t\,|\,) + \tau^2 I(s = t))$
- “range”: Spatial coordinates are sampled randomly from $[0,1]^2$.

Simulate data: geospaNN.Simulation

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

p = 1;
funXY = f1

n = 1000
nn = 20
range = [0,1]

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=range)
```

$$Y_i(s) = m\left(X_i(s)\right) + w(s) + \epsilon(s)$$

"X": X(s), non-spatial covariates sampled from [0,1]

"Y": Y(s), response

"coord": s, spatial coordinates

"cov": covariance matrix

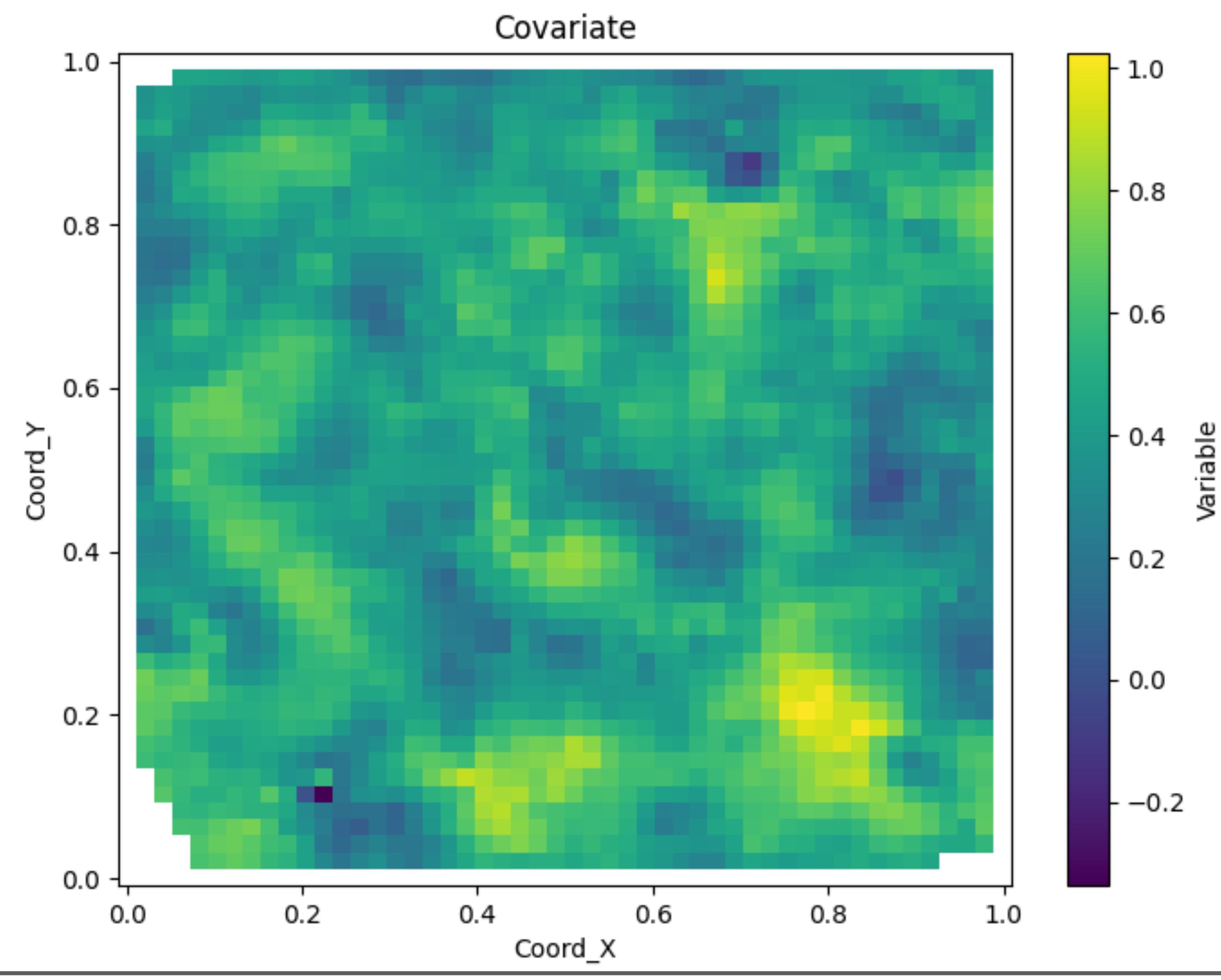
"corerr": $w(s) + \epsilon(s)$, correlated effect (error) term

Simulate data: geospaNN.Simulation

What if we want $X(s)$ to also have spatial distribution?

1: Simulate $X(s)$ as a spatial term.

```
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())
```

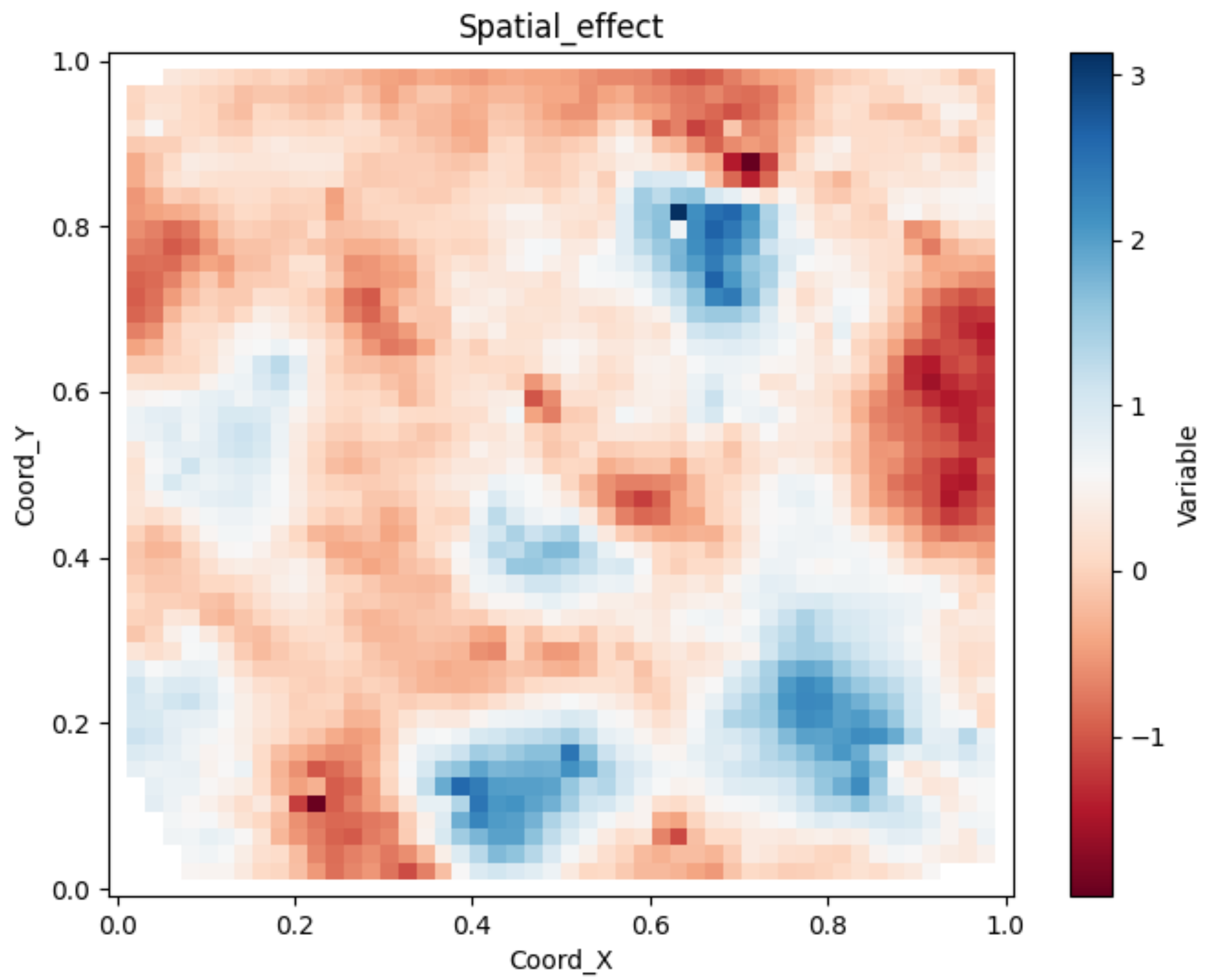


Simulate data: geospaNN.Simulation

What if we want $X(s)$ to also have spatial distribution?

2: Simulate spatial coordinates and true spatial effect.

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

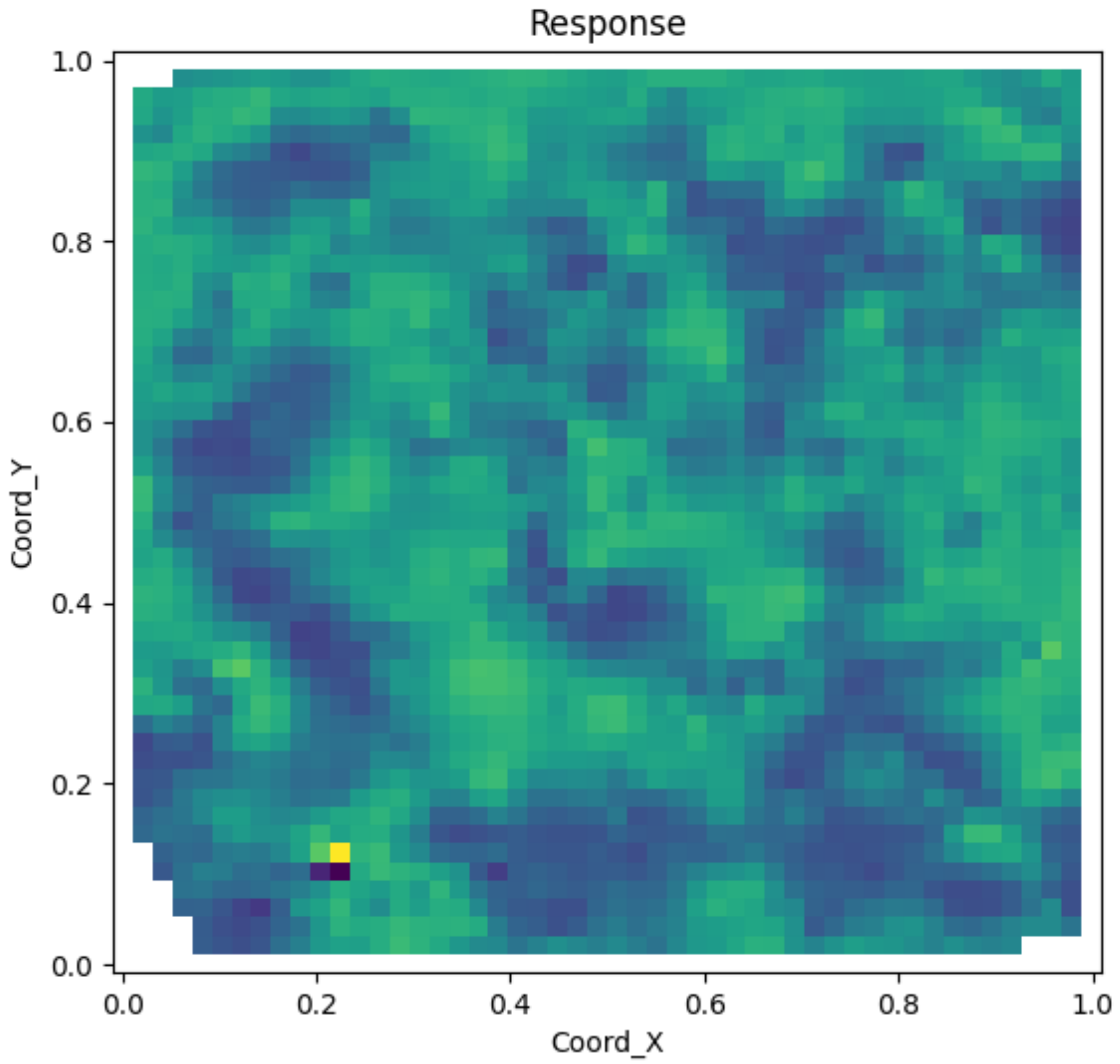


Simulate data: geospaNN.Simulation

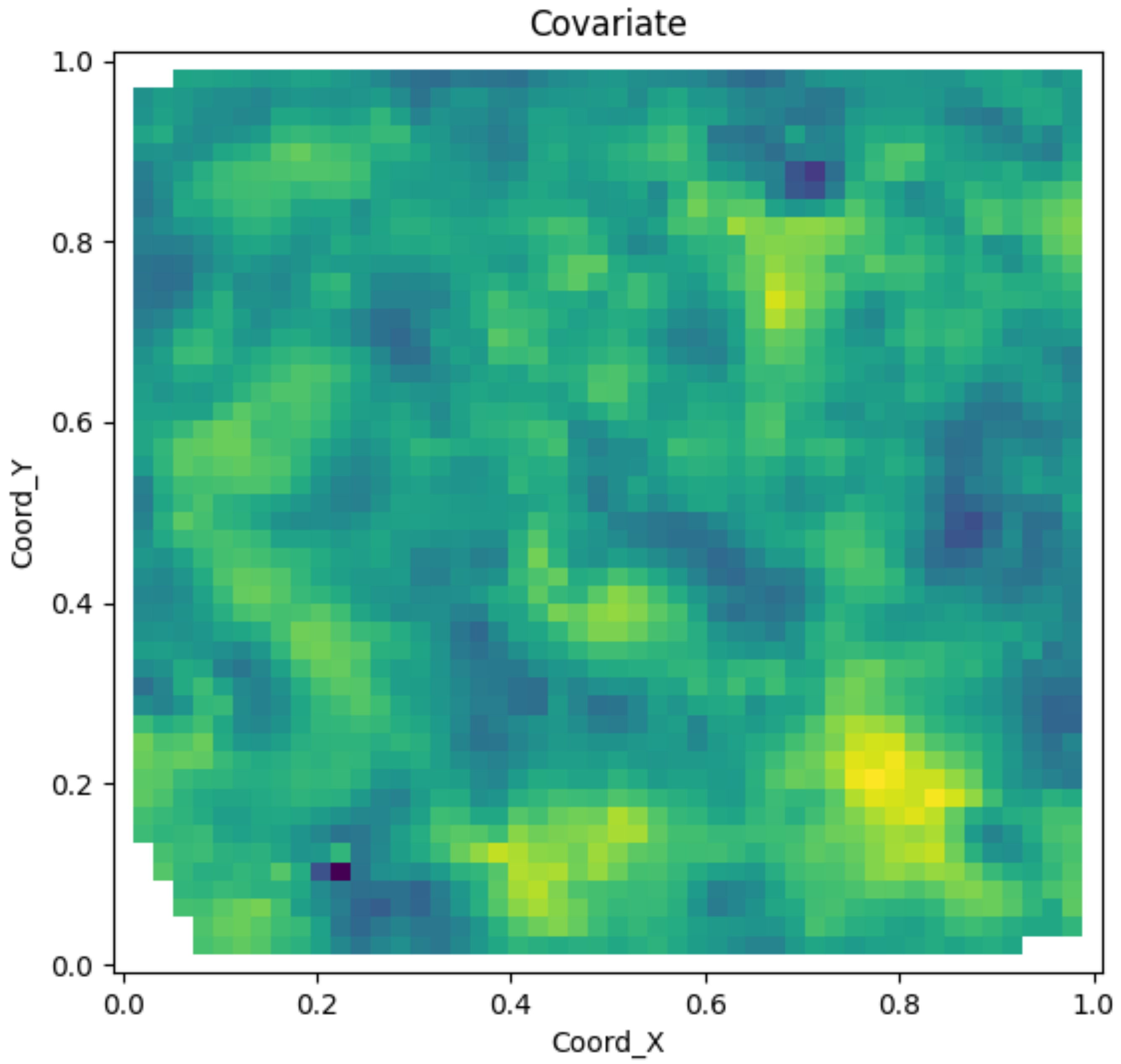
What if we want $X(s)$ to also have spatial distribution?

3: Compose the response.

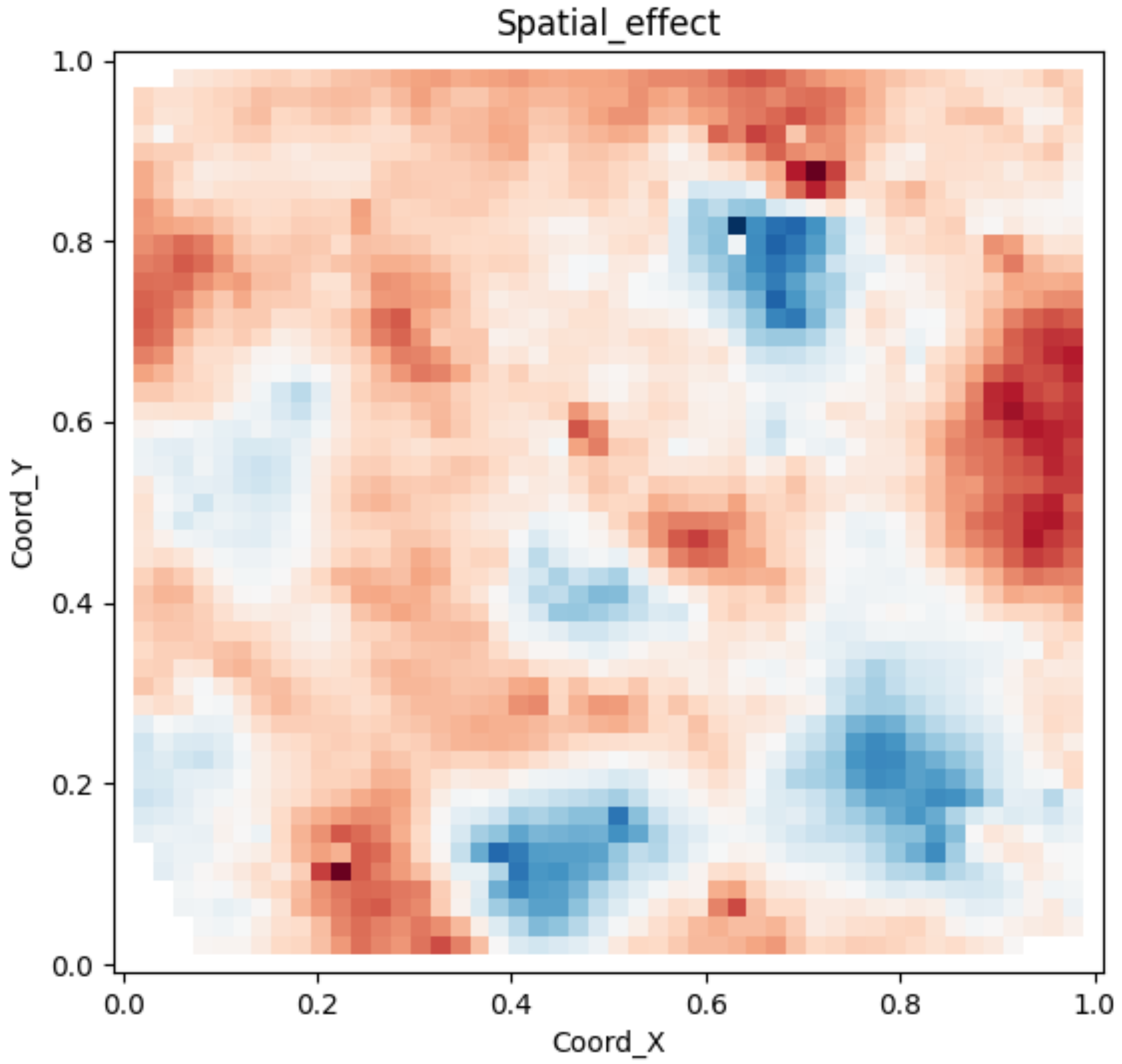
```
Y = funXY(X).reshape(-1) + corerr
```



=



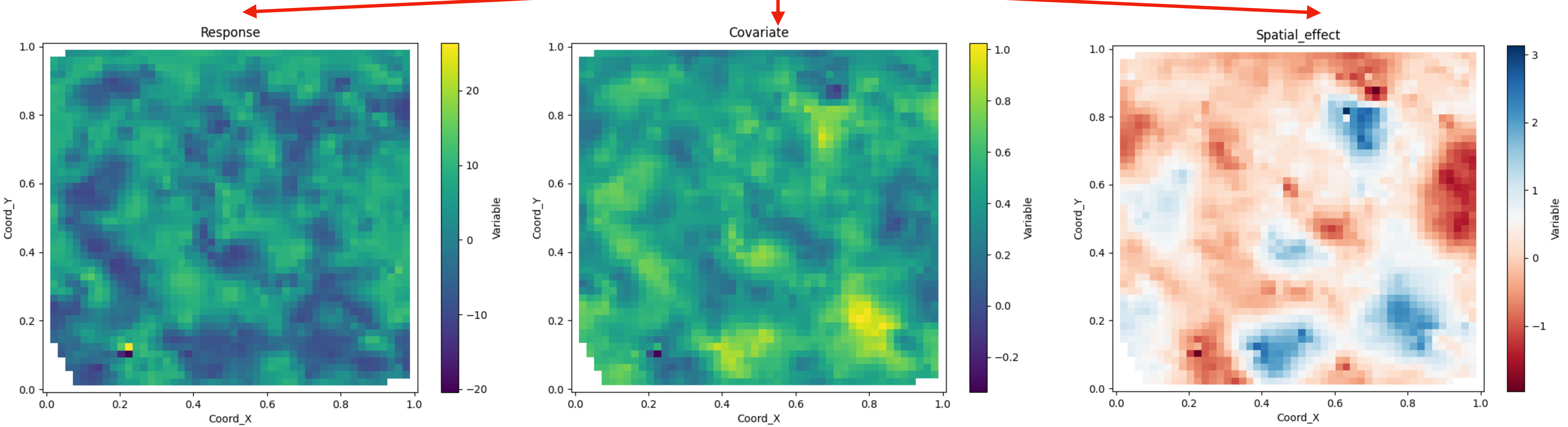
+



Visualize data: geospaNN.spatial_plot_surface

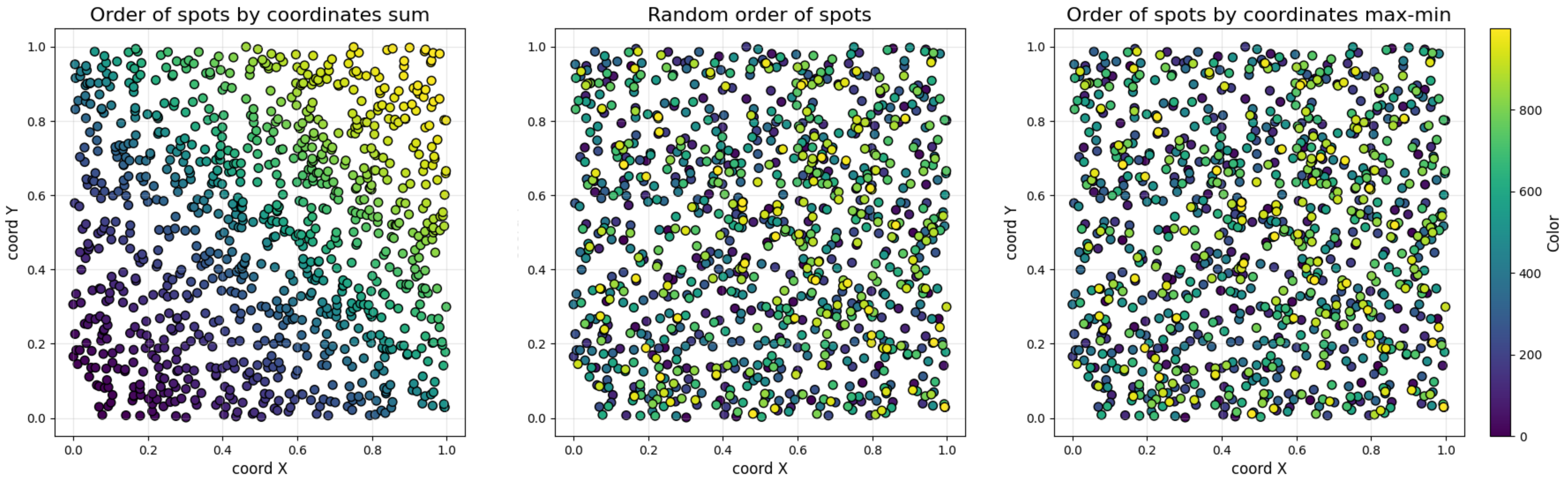
```
dict = {"Covariate": X, "Response": Y, "Spatial_effect":corerr}

for index, (name, variable) in enumerate(dict.items()):
    geospaNN.spatial_plot_surface(variable.detach().numpy().reshape(-1), coord.detach().numpy(),
                                  grid_resolution = 50, method = "CloughTocher",
                                  title = name, save_path = path, file_name = name + ".png")
```

$$\boxed{Y(s)} = m(\boxed{X(s)}) + \boxed{w(s) + \epsilon(s)}$$



Spatial ordering

```
X, Y, coord, _ = geospaNN.spatial_order(X, Y, coord, method='max-min')
```



Spatial data loader: geospaNN.make_graph

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

Data loader provides an efficient way to iterate over a dataset.

geospaNN.make_graph generates data loader object including:

- X, Y, s in spatial modeling.
- Edge index and attributions connecting nearest neighbors.
- Allow users to pass predefined $n \times k$, k neighbor index matrix.
- Objects can be called by data.x, data.y, data.pos, ……
- Key function.

Training-testing split: geospaNN.split_data

```
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)
```

geospaNN.split_data generates three data objects.

It's a wrapper for:

$$\{Y, X, s\}$$

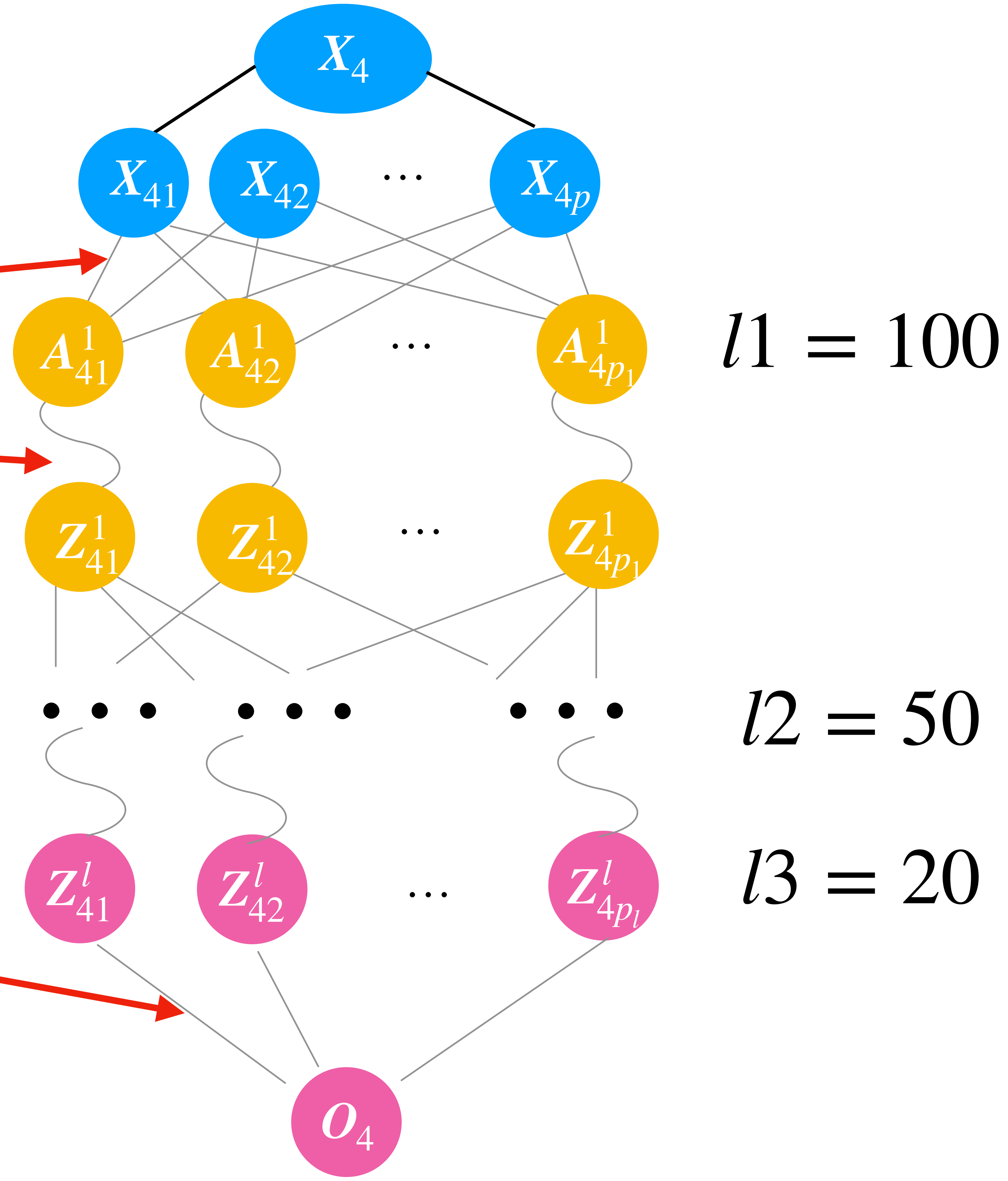
$$\{Y_{train}, X_{train}, s_{train}\} \qquad \{Y_{val}, X_{val}, s_{val}\} \qquad \{Y_{test}, X_{test}, s_{test}\}$$

60% 20% 20%

```
data_train = make_graph(X_train, Y_train, coord_train, neighbor_size)
data_val = make_graph(X_val, Y_val, coord_val, neighbor_size)
data_test = make_graph(X_test, Y_test, coord_test, neighbor_size)
```

Model training: ordinary neural networks

```
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),
)
```



Model training: ordinary neural networks

```
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)
```

- “mlp_nn”: multi-layer perceptron (mlp) architecture
- “nn_model”: object for training process and hyper parameters.
    - “lr”: learning rate.
    - “min_delta”: cutoff for “significant update”.
- “nn_model.train”: A wrapper for the common training loop.

Model training: NN-GLS

```
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)
```

$$Cov\left(w(s_1), w(s_2)\right) = C(s_1, s_2 \,|\, \theta) = \sigma^2\left(\exp(-\phi\,|\,s_1 - s_2\,|\,) + \tau^2 I(s_1 = s_2)\right)$$

- Spatial parameters are initialized and updated in training.
  - “Update_init” is the initial epoch starting updating.
  - “Update_step” is the gap of epochs between updates.
- Everything else than θ are the same to NN.

How θ get updated?

```
torch.manual_seed(2025)
_, _, coord_simp, _, corerr_simp = geospaNN.Simulation(n, p, nn, funXY,
                                                      torch.tensor([1,1.5,0.01]), range=[0, 10])
theta_hat = geospaNN.theta_update(corerr_simp, coord_simp, neighbor_size=20)
```

```
Theta estimated as
[0.88942914 1.74742522 0.01030131]
```

- geospaNN.theta_update currently call the **BRISC** R-package (Saha, & Datta, 2018) for likelihood-based parameter estimation.

How θ get updated?

```
torch.manual_seed(2025)
_, _, coord_simp, _, corerr_simp = geospaNN.Simulation(n, p, nn, funXY,
                                              torch.tensor([1,1.5,0.01]), range=[0, 10])
theta_hat = geospaNN.theta_update(corerr_simp, coord_simp, neighbor_size=20)
```

```
Theta estimated as
[0.88942914 1.74742522 0.01030131]
```

- geospaNN.linear_GLS, as wrapper of BRISC_estimation() in R, can be called to solve β and θ in SPLMM:

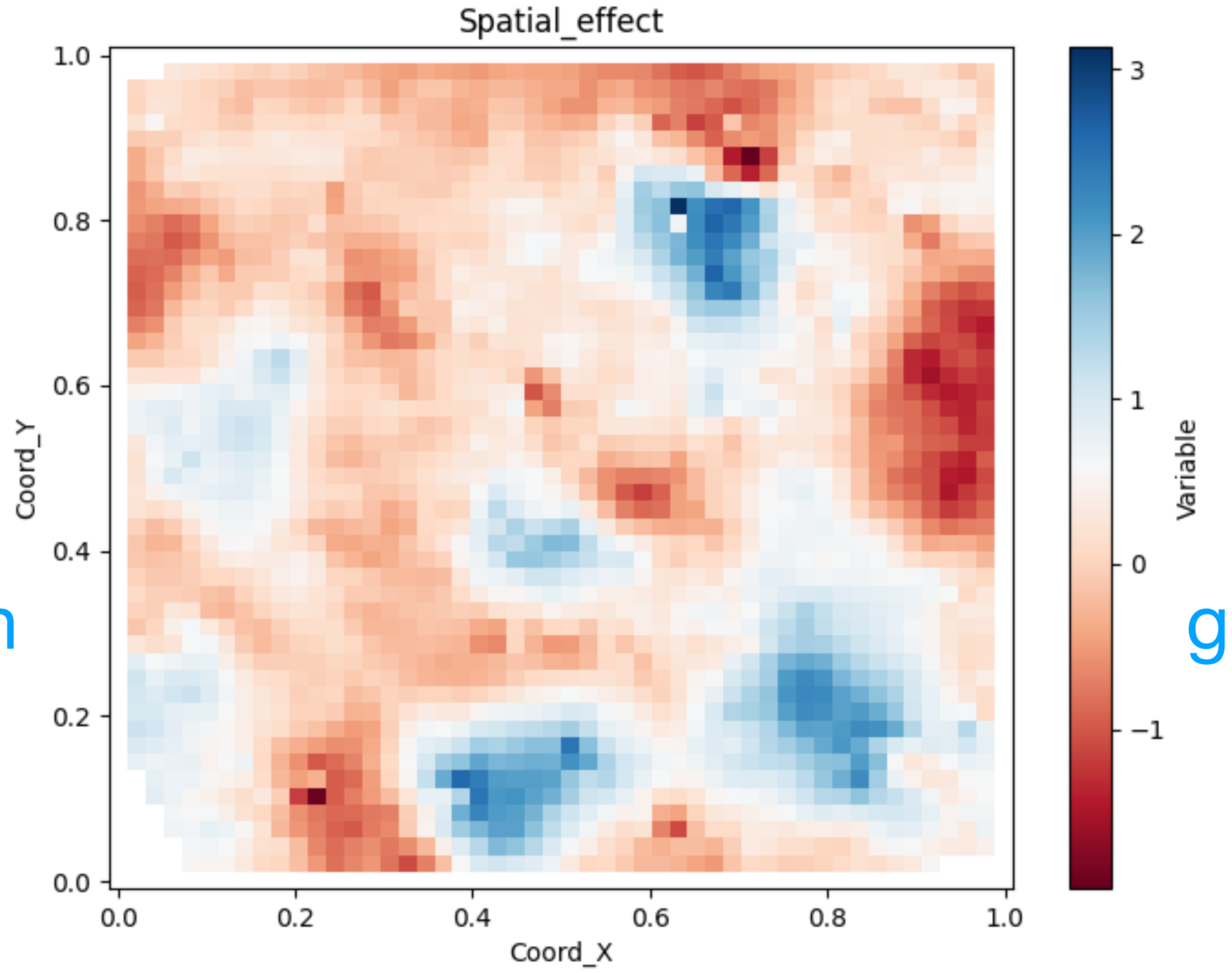
$$Y_i(s) = \boxed{X_i(s)\beta} + w(s) + \epsilon(s), w(s) \sim C(\cdot, \cdot \,|\, \theta)$$

$$Y_i(s) = \boxed{m\left(X_i(s)\right)} + w(s) + \epsilon(s)$$

How θ get updated?

$\theta \xrightarrow{\text{Simulation}}$

geospaNN.Simulation

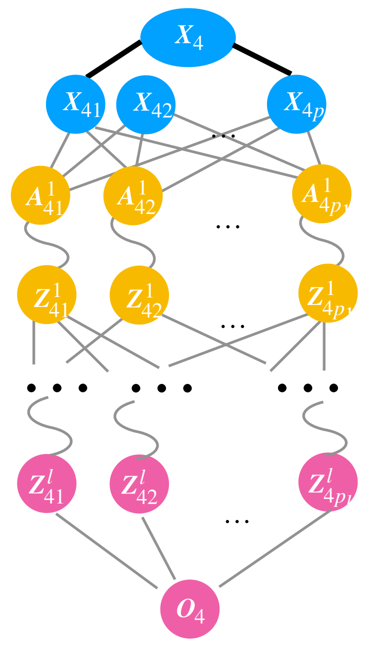


$\xrightarrow{\text{Maximum likelihood}} \hat{\theta}$

geospaNN.theta_update

How θ get updated in NN-GLS?

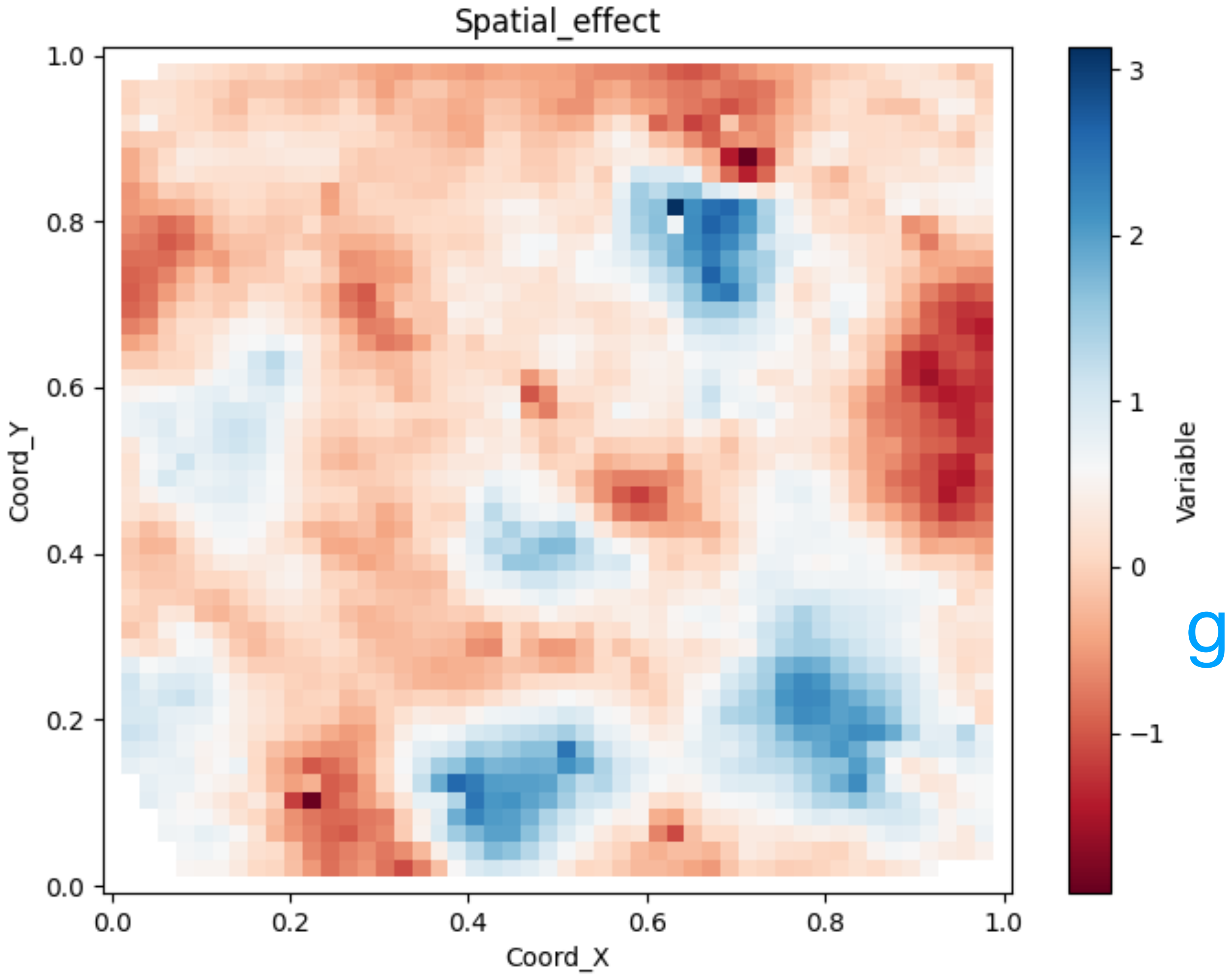
```
theta0 = geospaNN.theta_update(mlp_nn(data_train.x).squeeze() - data_train.y,
                               data_train.pos, neighbor_size=20)
```



$\hat{f}$

Get Residual

$Y - \hat{m}(X)$



Maximum likelihood

$\hat{\theta}$

geospaNN.theta_update

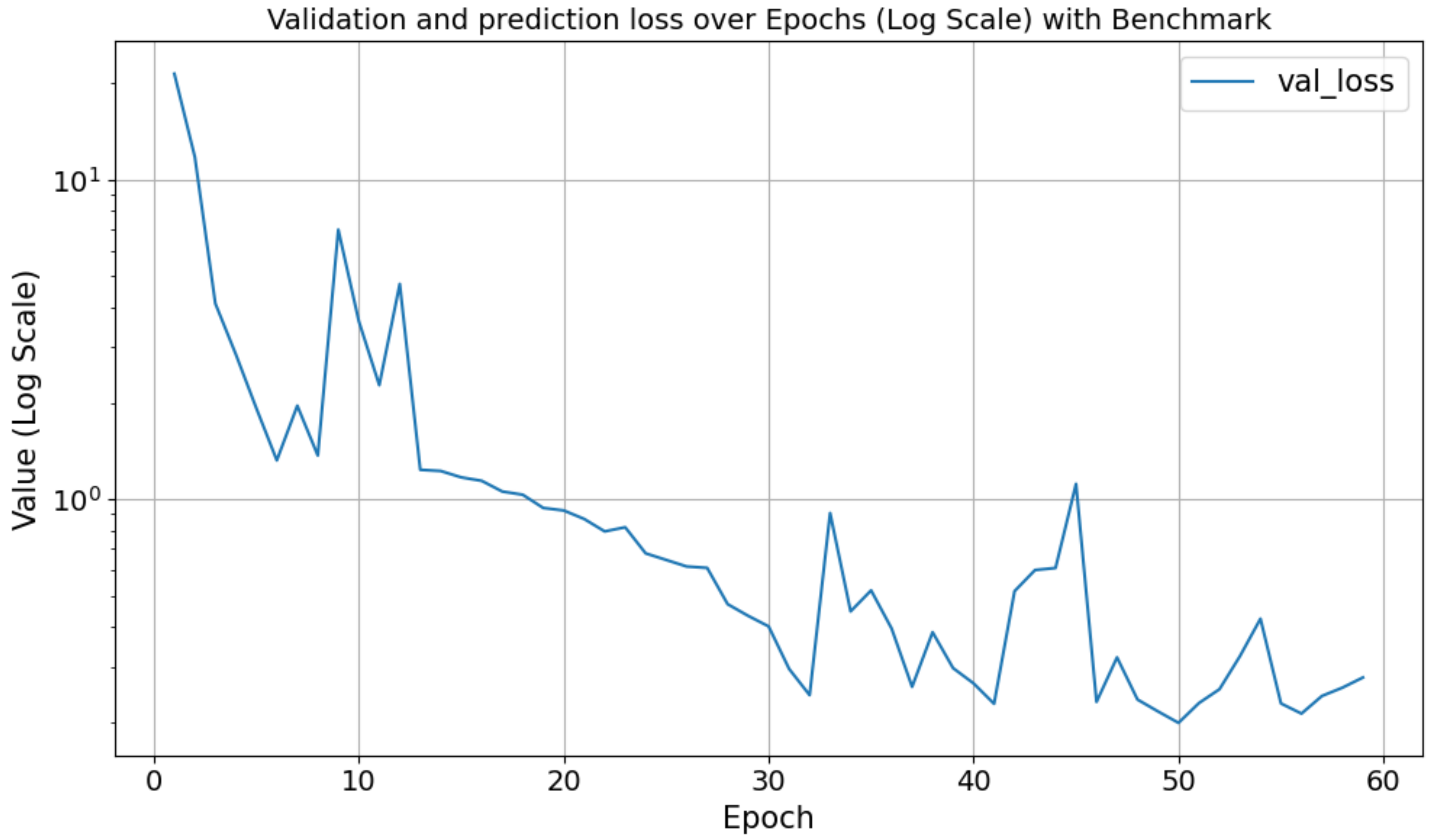
Back-propagation

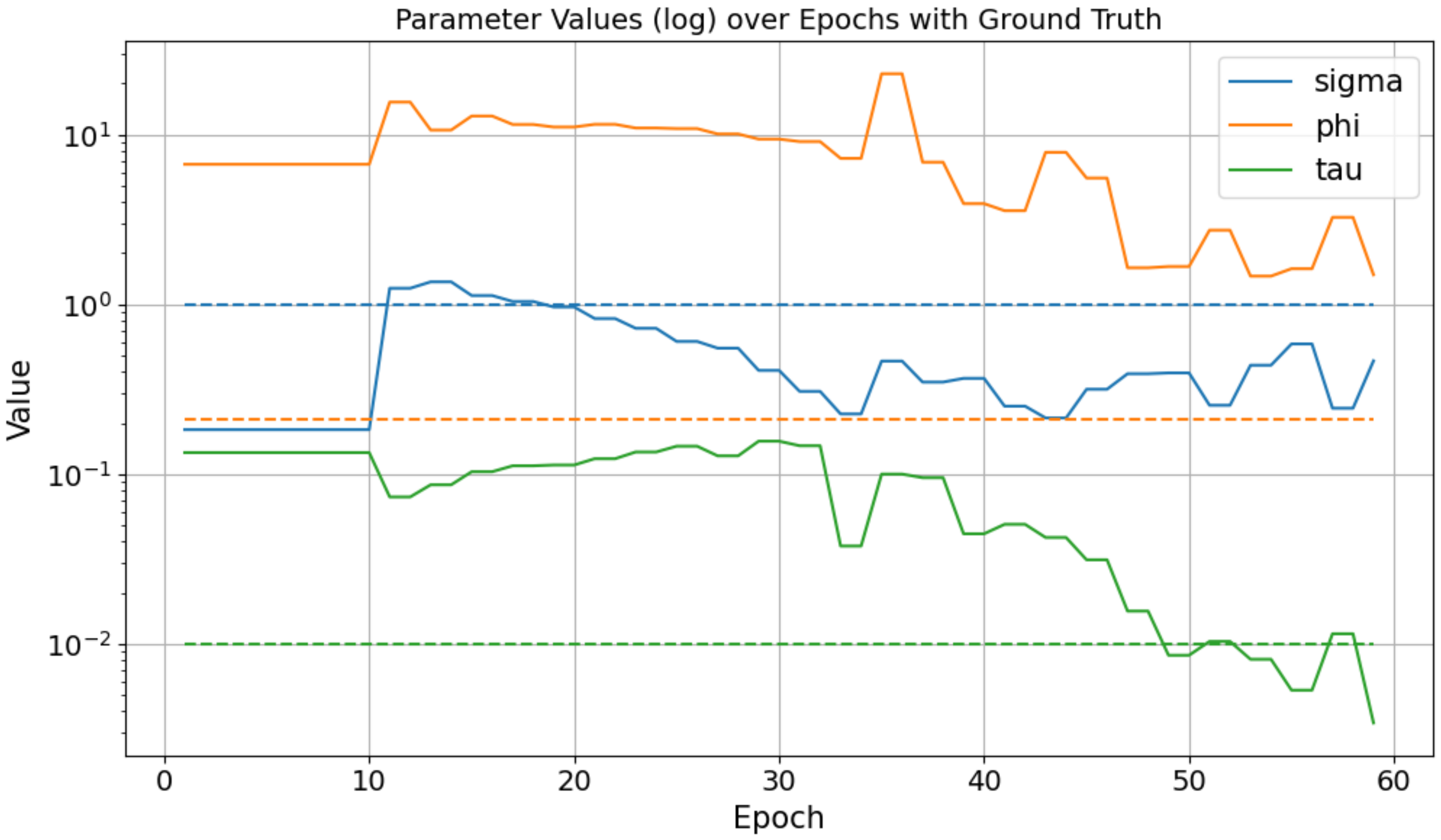
geospaNN.nngls_train.train

Training output

Training curves of validation loss and spatial parameters.

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



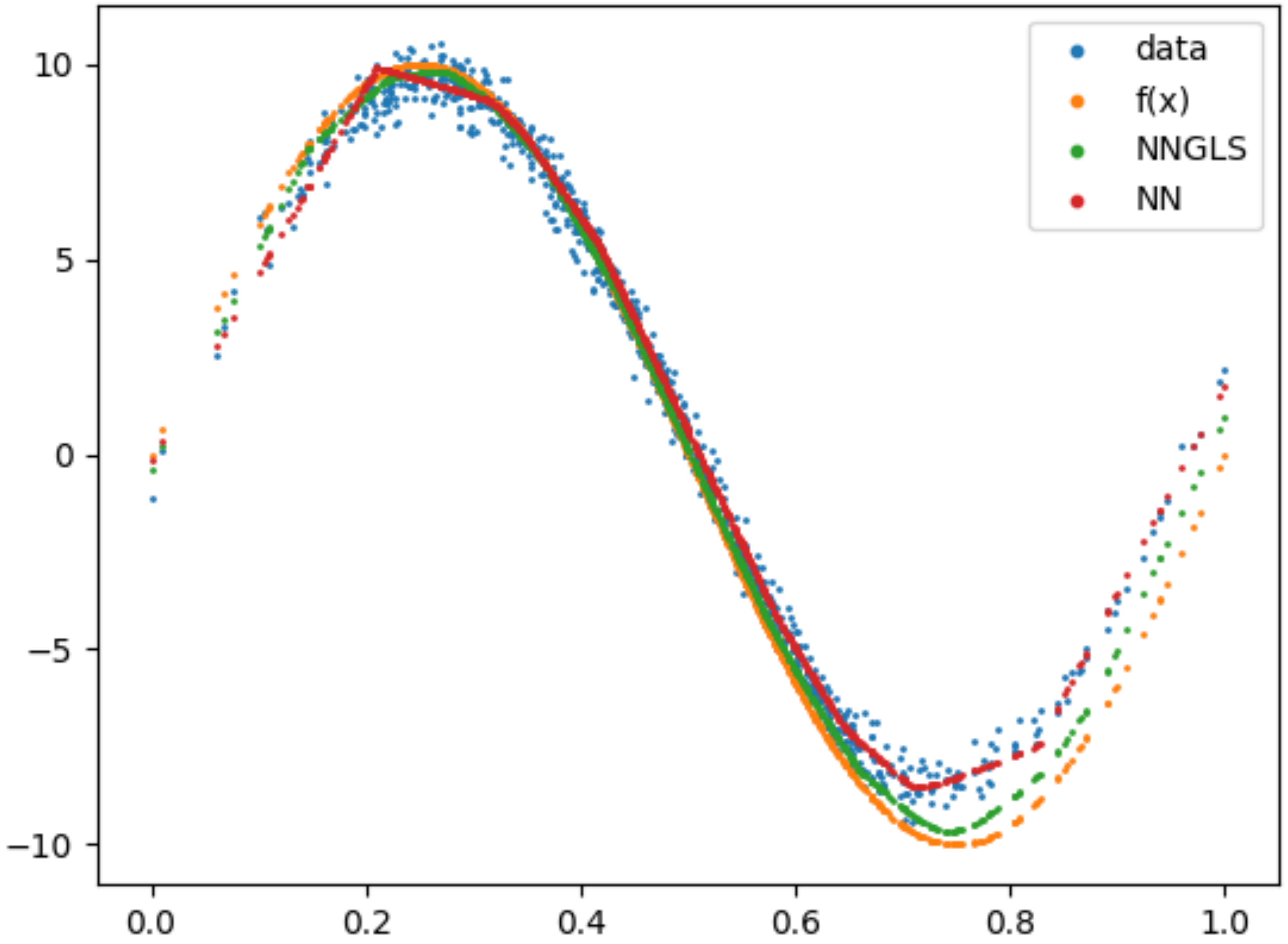


Estimation: geospaNN.nngls.estimate

```python
model = geospaNN.nngls(p=p, neighbor_size=nn, coord_dimensions=2, mlp=mlp_nngls,
                       theta=torch.tensor(theta0))
```

Equivalent to “model.mlp(X)”.

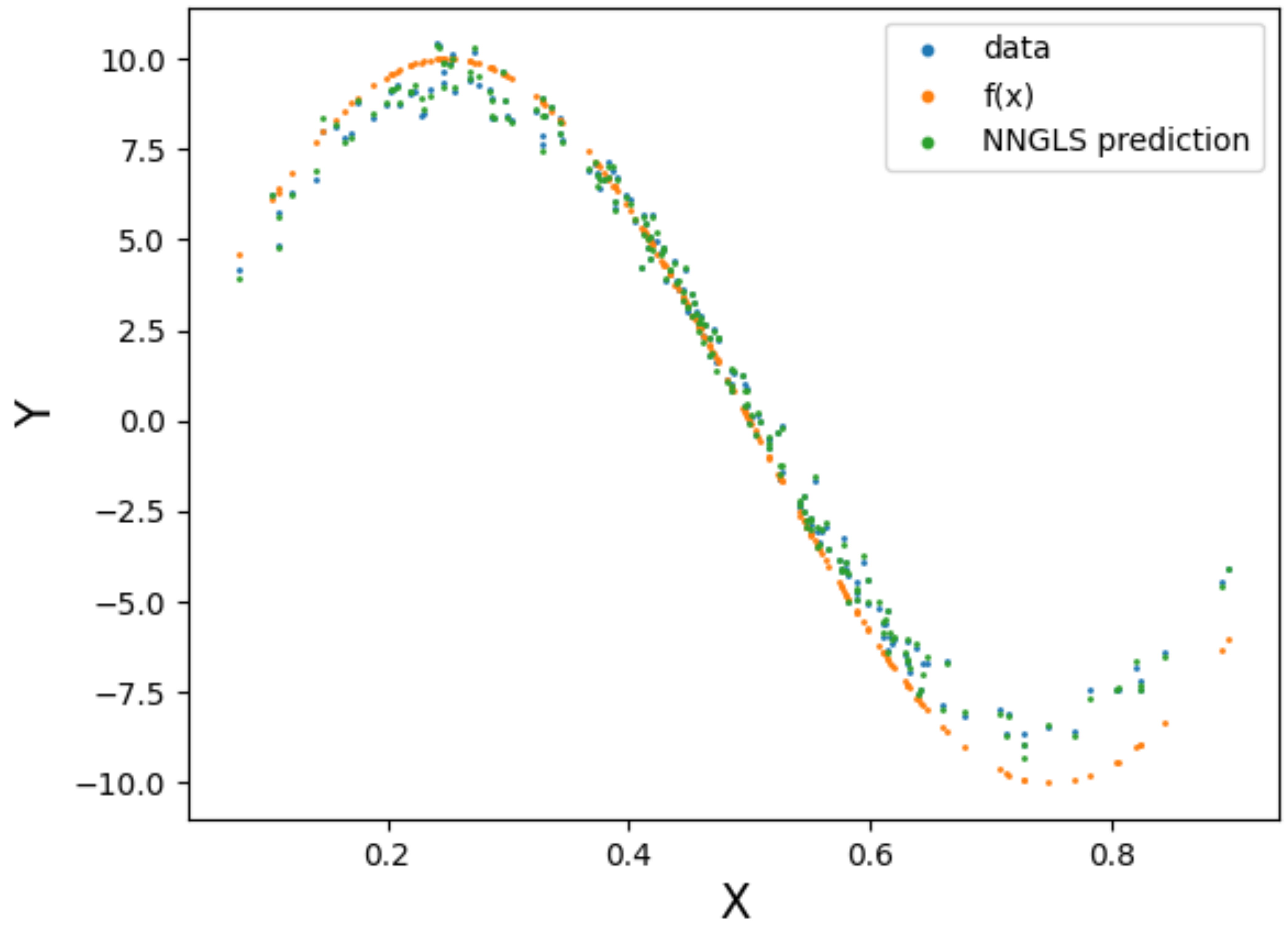
```python
estimate = model.estimate(X)
```

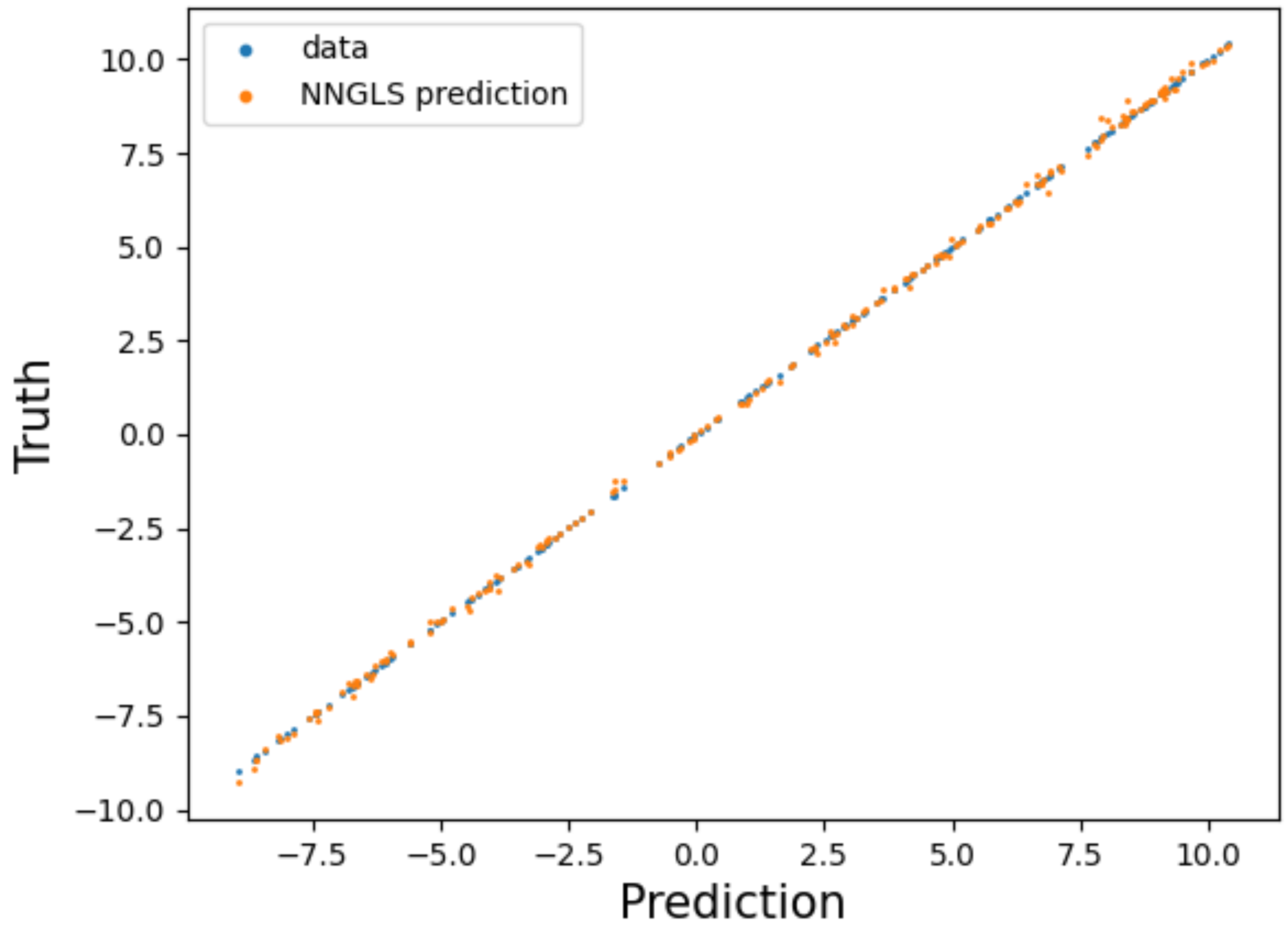


Prediction: geospaNN.nngls.predict

Efficient prediction through nearest neighbor kriging:

```
test_predict = model.predict(data_train, data_test)
```

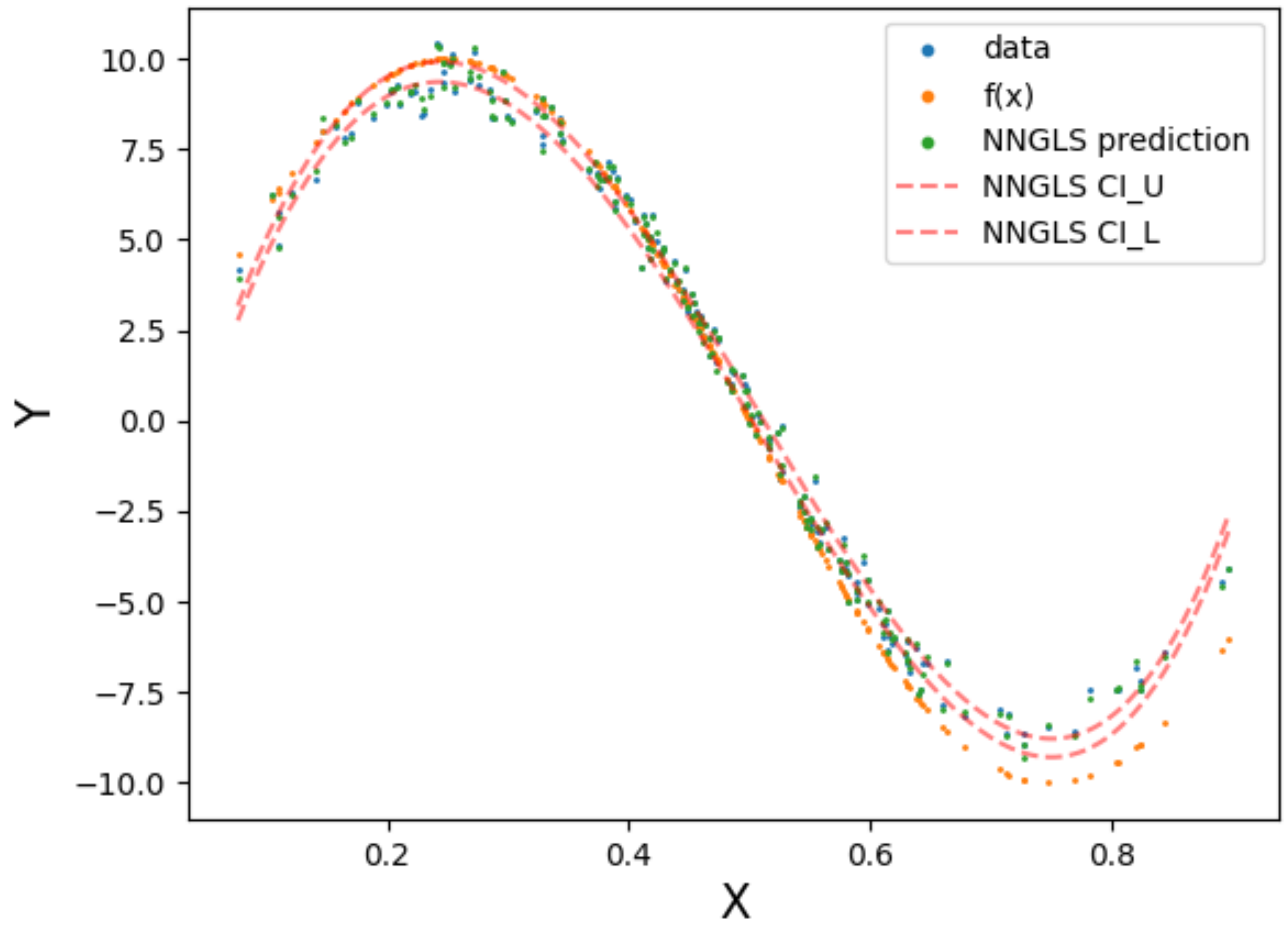


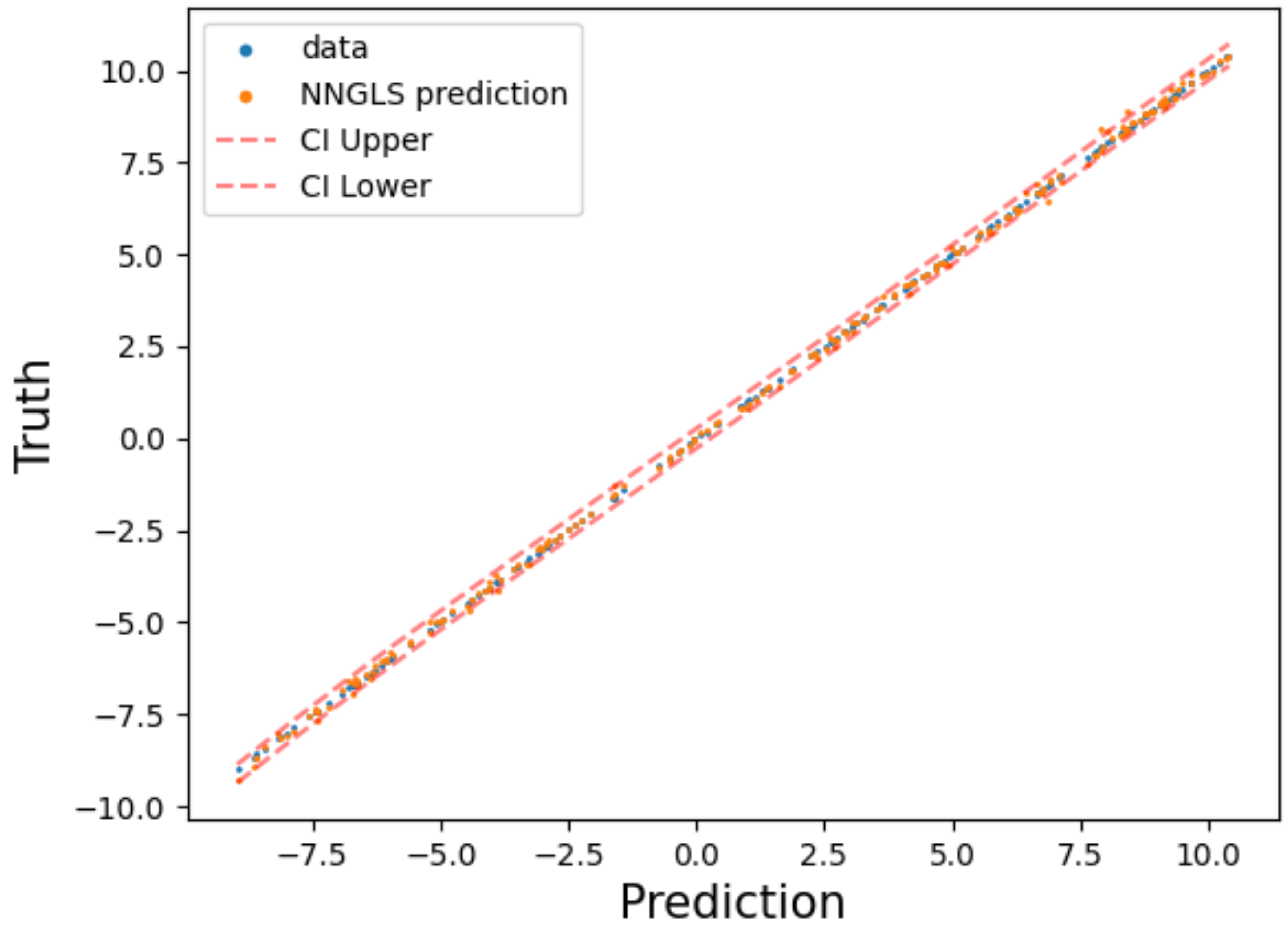


Prediction: geospaNN.nngls.predict

Efficient prediction through nearest neighbor kriging:

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

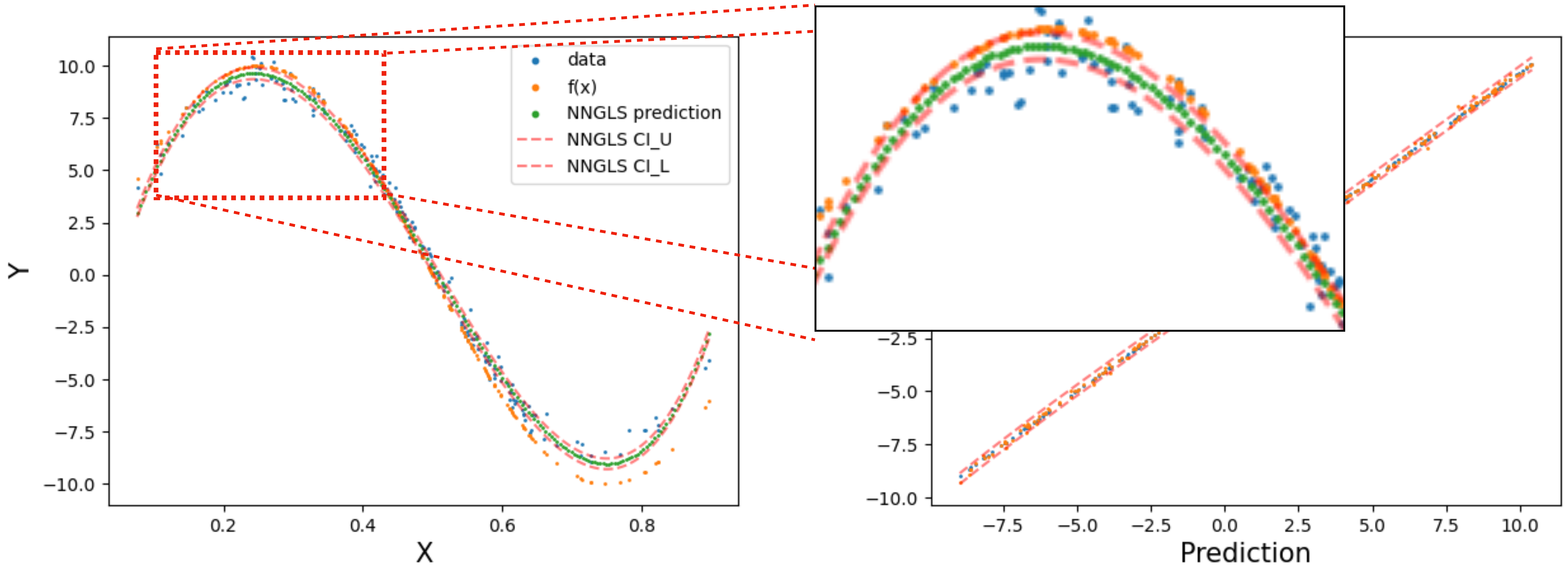




Prediction: geospaNN.nngls.predict

Efficient prediction through nearest neighbor kriging:

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



PDP (Partial Dependency Plot)

Partial Dependence plot shows the dependence between the target function $m(X_1, \cdots, X_p)$ and a set of individual features X_i.

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

Example: Friedman's function:

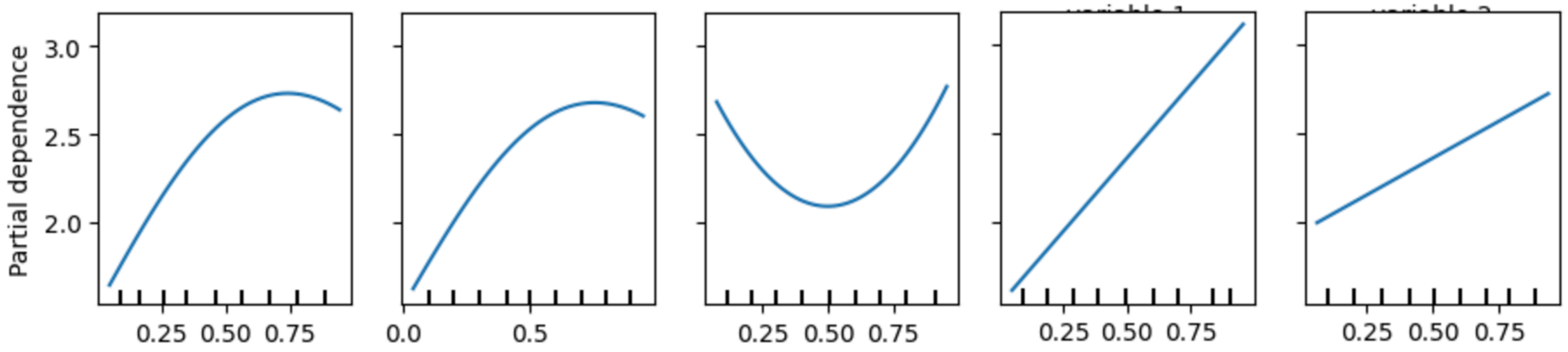
$$m(X) = \left(10 \sin(\pi X_1 X_2) + 20(X_3 - 0.5)^2 + 10X_4 + 5X_5\right)/6$$

PDP: geospaNN.plot_PDP

Example: Friedman's function:

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

```
def f5(X): return (10 * np.sin(np.pi * X[:, 0] * X[:, 1]) + 20 * (X[:, 2] - 0.5) ** 2 +
                   10 * X[:, 3] + 5 * X[:, 4]) / 6

PDP_truth = geospaNN.visualize.plot_PDP(funXY, X, names = ["PDP"], save_path = path, save = True)
```

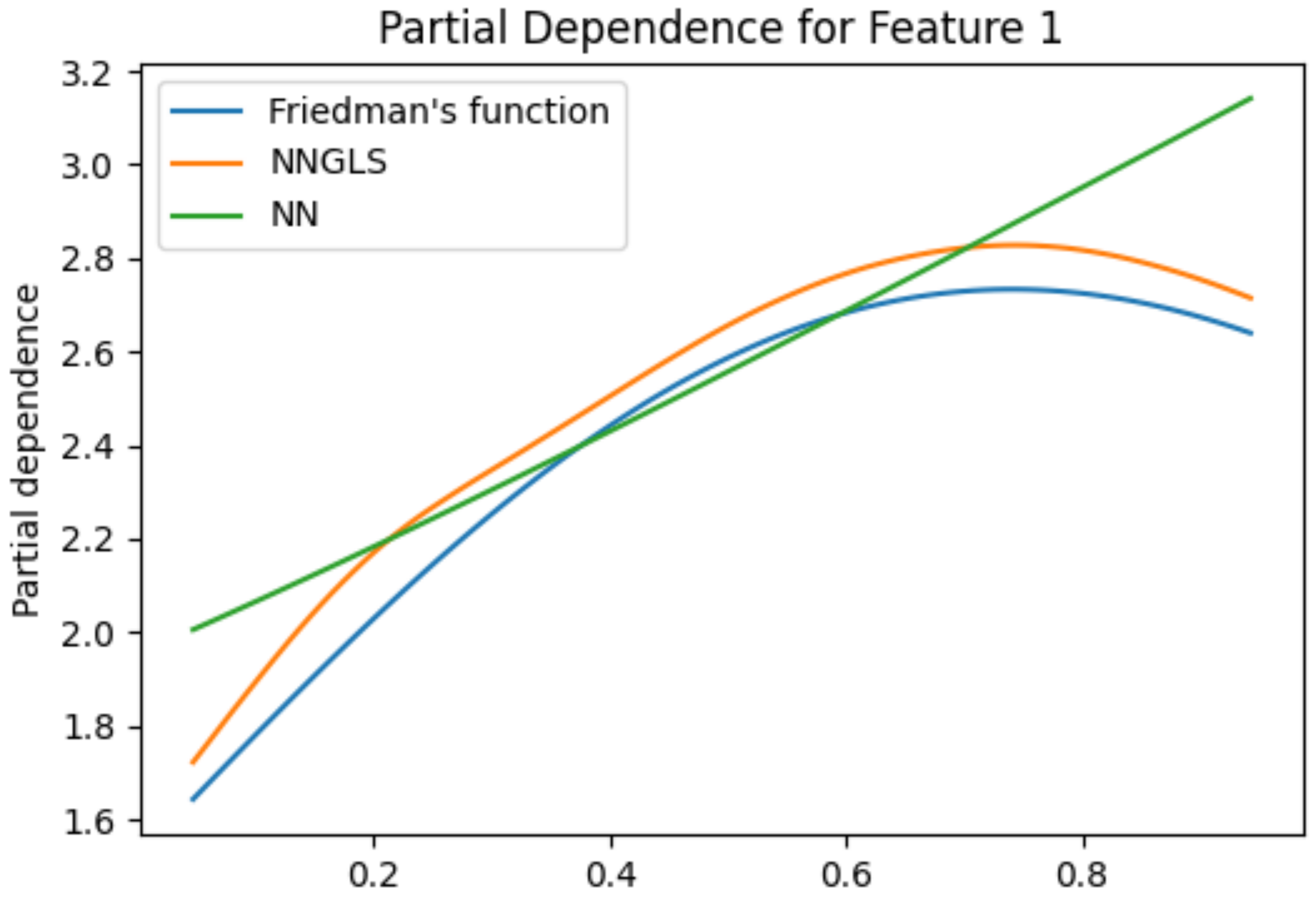


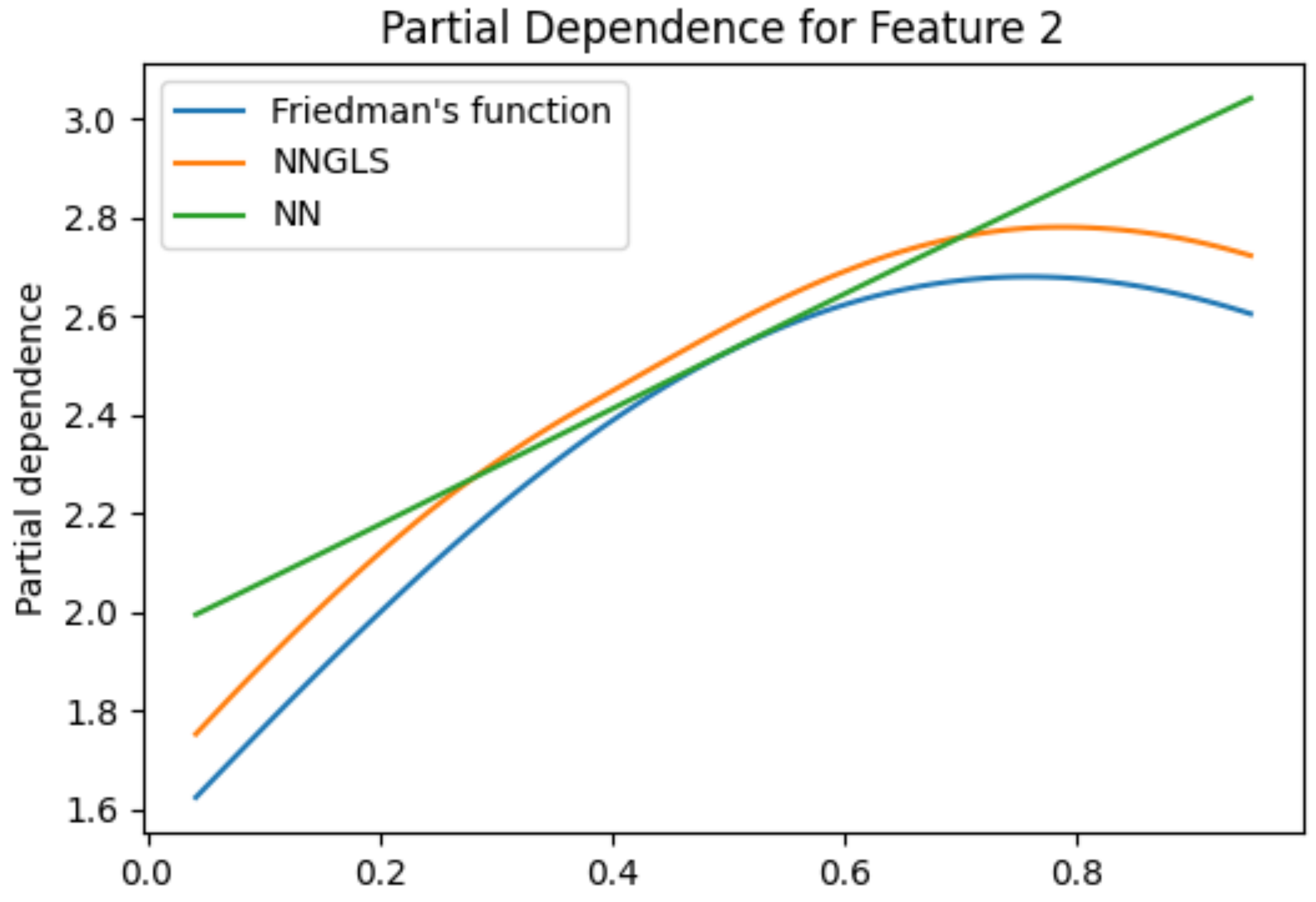
PDP: geospaNN.plot_PDP_list

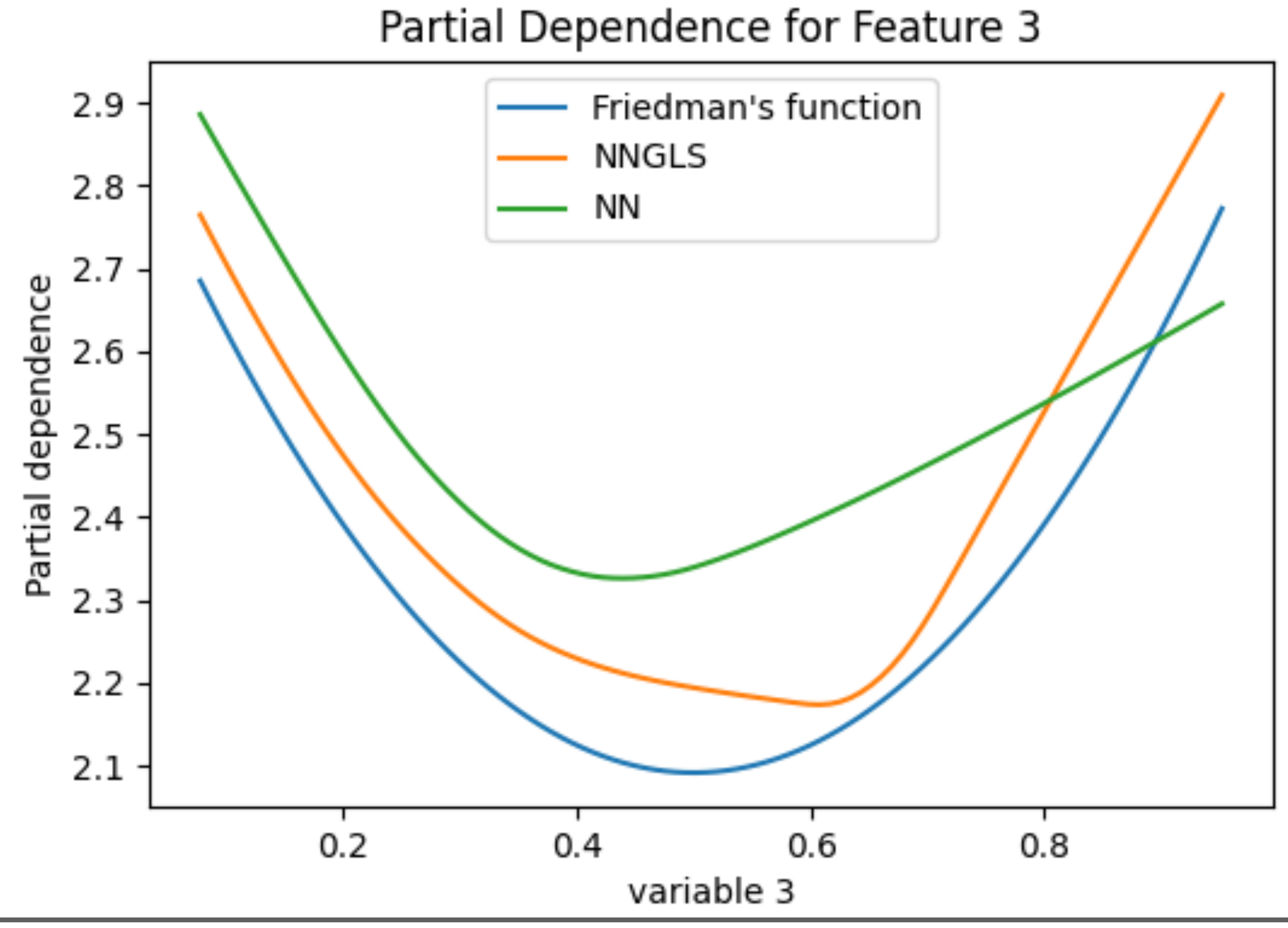
```python
geospaNN.visualize.plot_PDP_list([funXY, mlp_nngls, mlp_nn], ['Friedmans function', 'NNGLS', 'NN'], X, split = True)
```

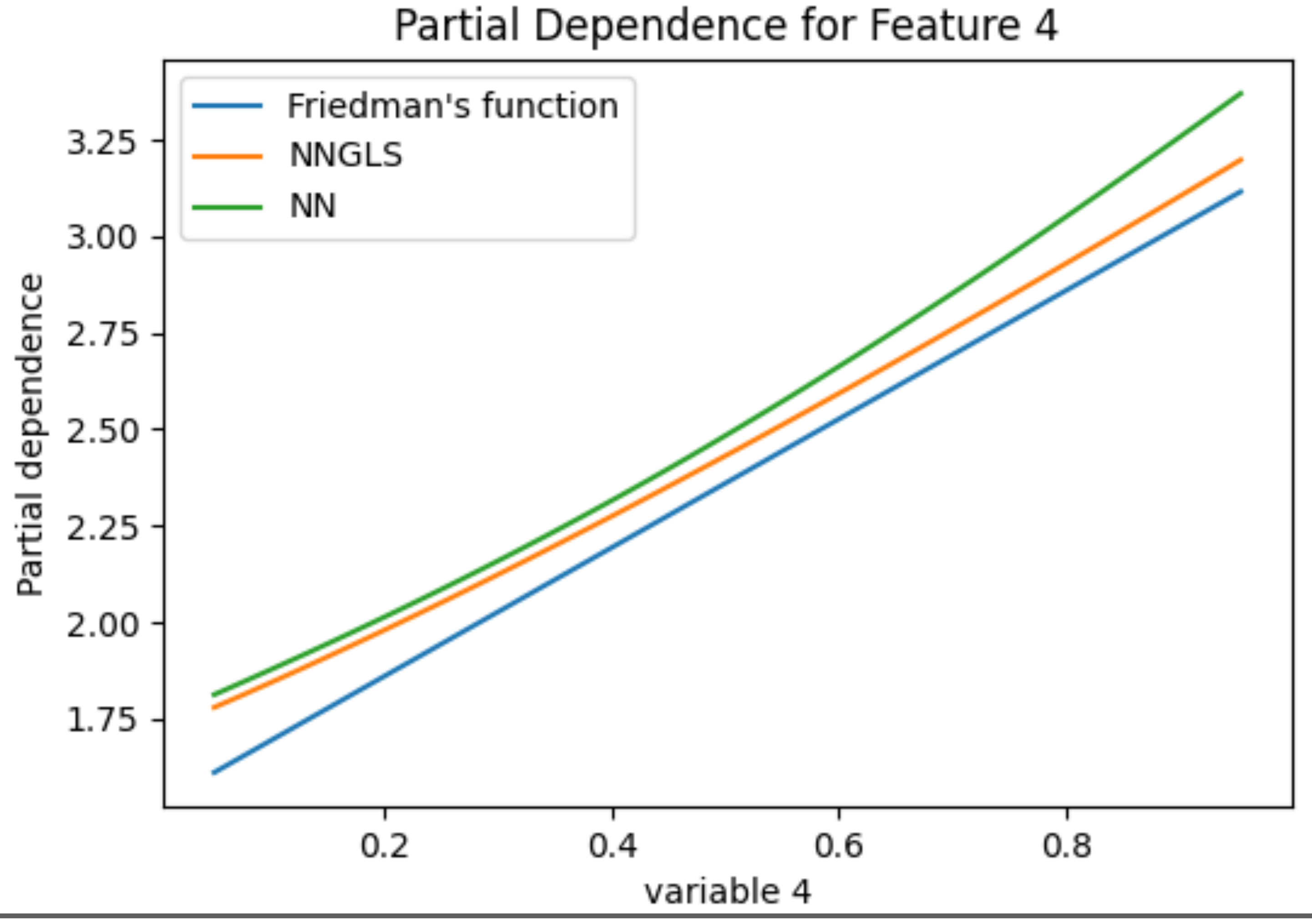
Friedman's function:

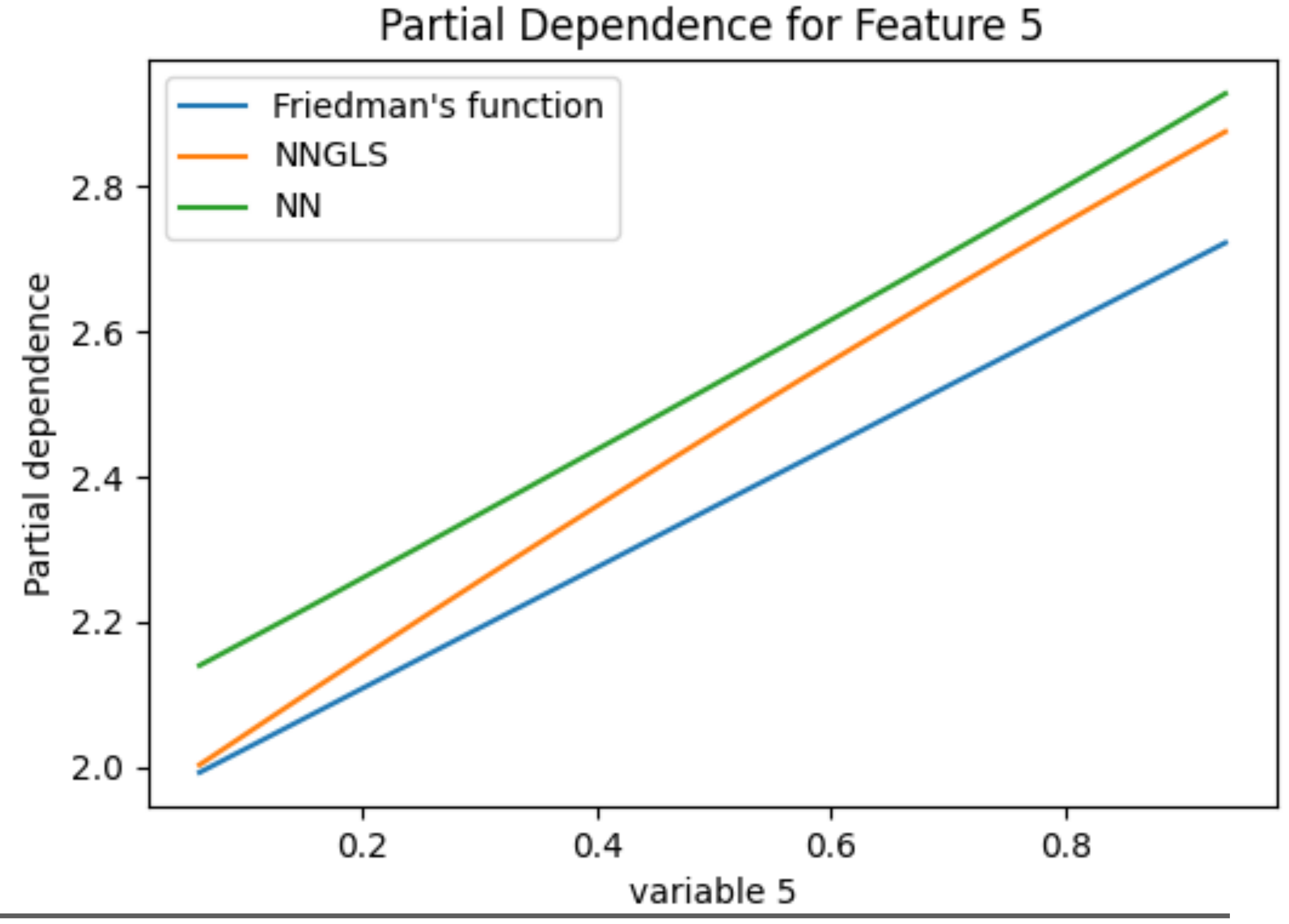
$$m(X) = \Big(10\sin(\pi X_1 X_2) + 20(X_3 - 0.5)^2 + 10X_4 + 5X_5\Big)/6$$











Outline

1. Basic functions

2. **Simulation examples**

   A. **General Architecture design**

   B. NNGLS handles complex interaction

   C. NNGLS vs add-covariate approaches

3. Real data example

Architecture design

```python
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),
)
```

```python
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 5),
    torch.nn.ReLU(),
    torch.nn.Linear(5, 1)
)
```

```python
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 5),
    torch.nn.ReLU(),
    torch.nn.Linear(5, 2),
    torch.nn.ReLU(),
    torch.nn.Linear(2, 1)
)
```

```python
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
```

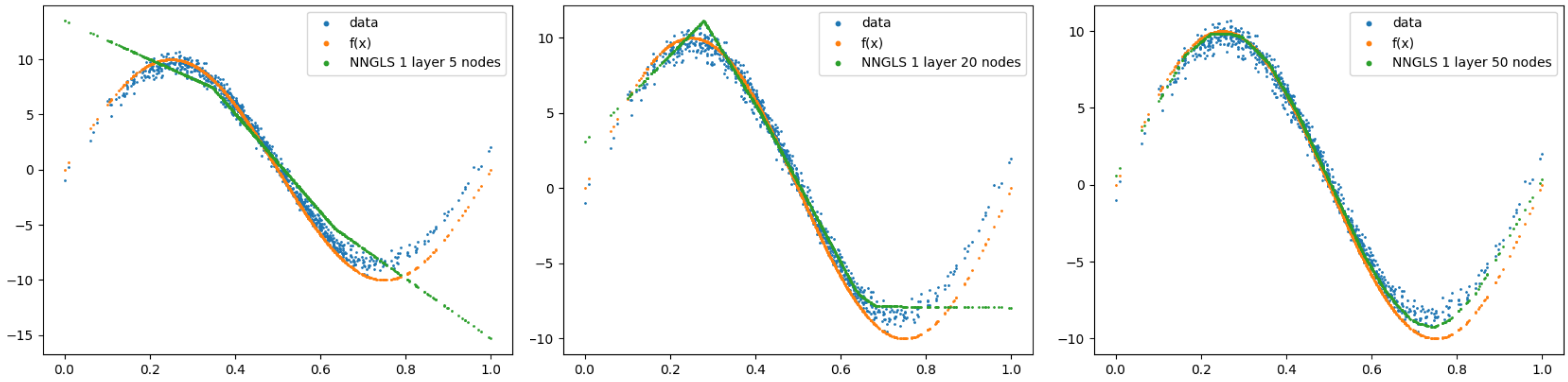
```python
mlp_nngls = torch.nn.Sequential(
    torch.nn.Linear(p, 50),
    torch.nn.ReLU(),
    torch.nn.Linear(50, 20),
    torch.nn.ReLU(),
    torch.nn.Linear(20, 1)
)
```

```python
model = geospaNN.nngls(p=p, neighbor_s
                       theta=torch.ter
nngls_model = geospaNN.nngls_train(mod
training_log = nngls_model.train(data_train, data_val, data_test, Update_ini
```

How to choose among architectures?

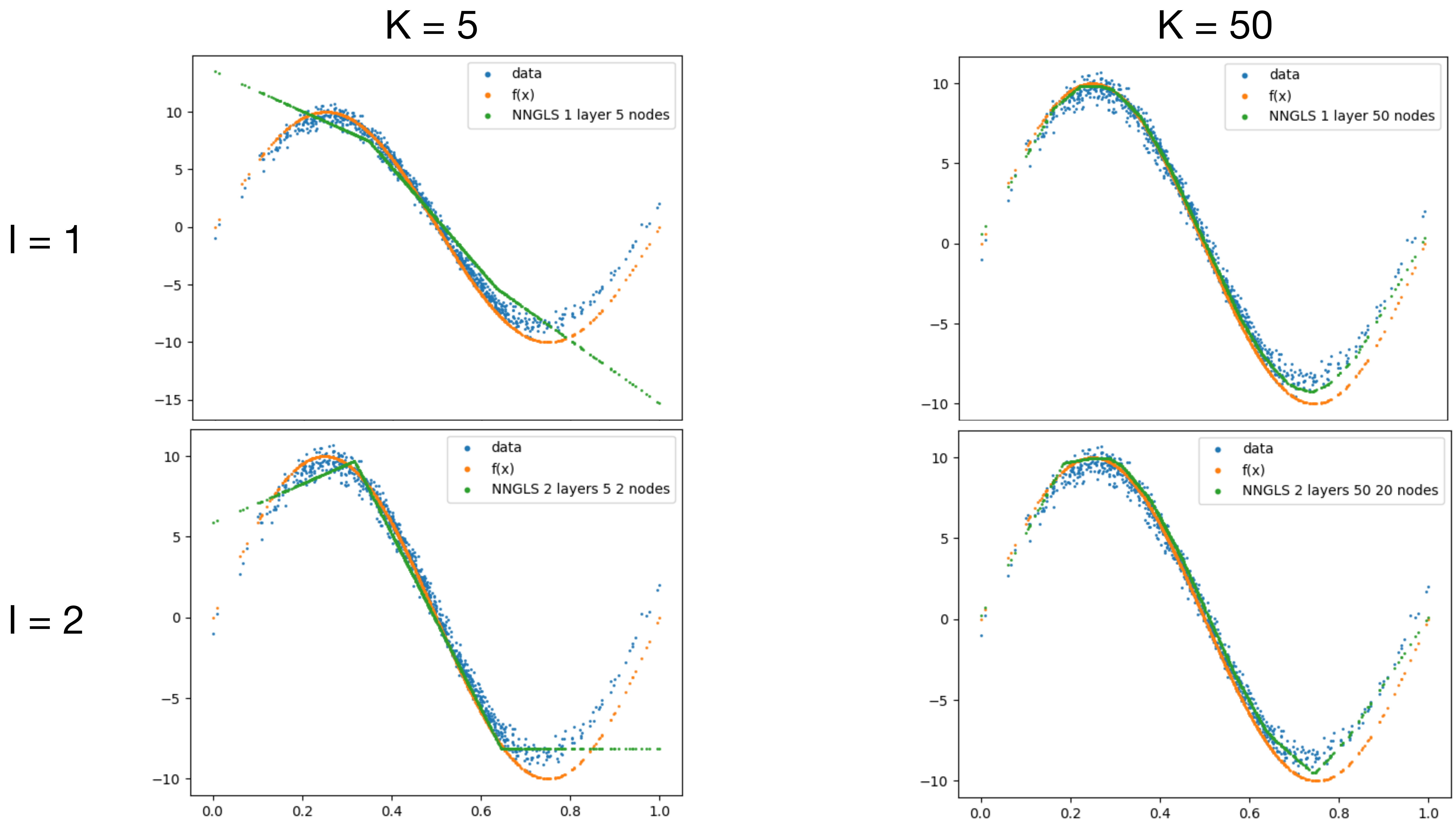
Architecture design: width

K = 5 K = 20 K = 50



Width of a layer

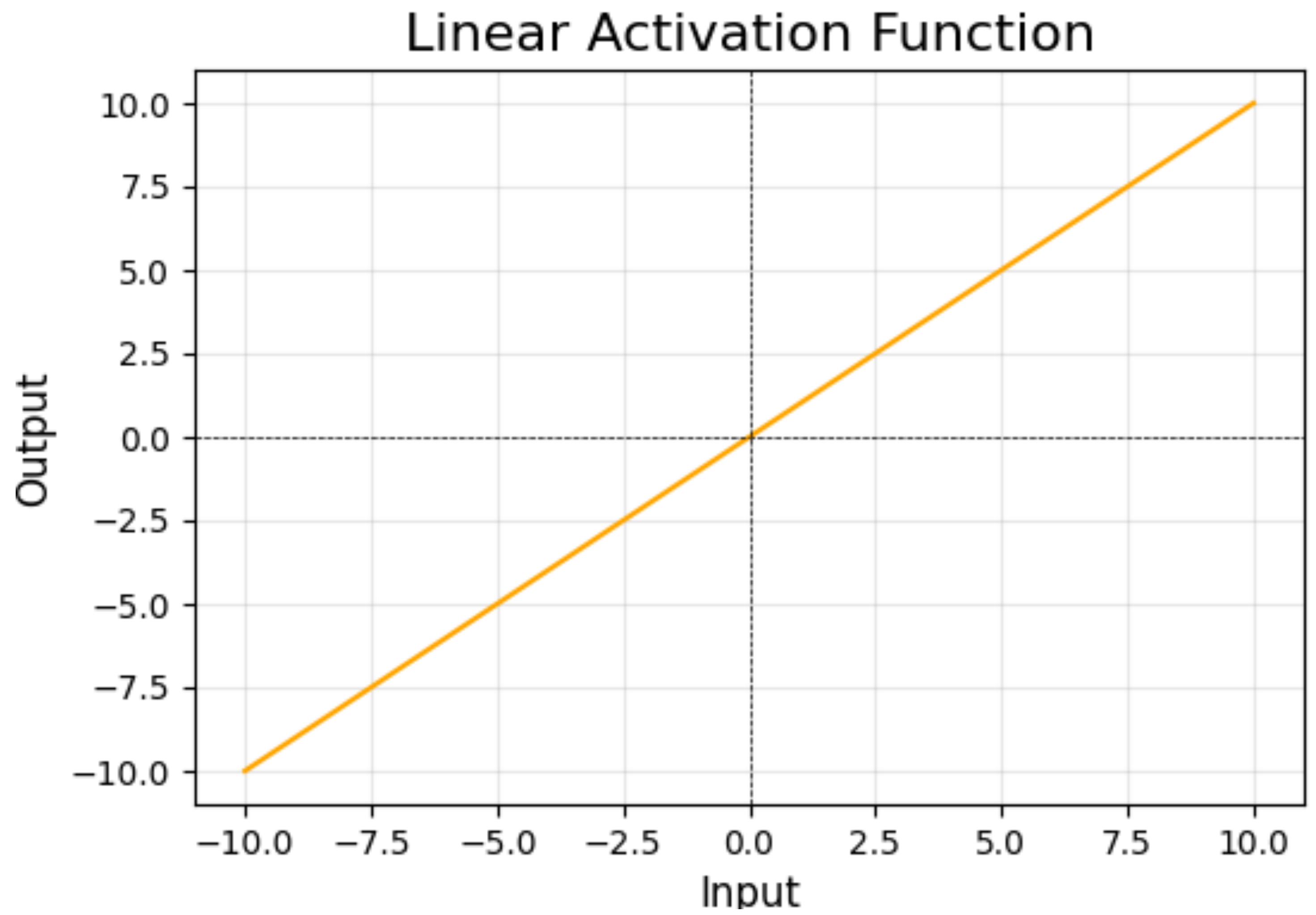
Architecture design: width & depth



Architecture design: Activation functions

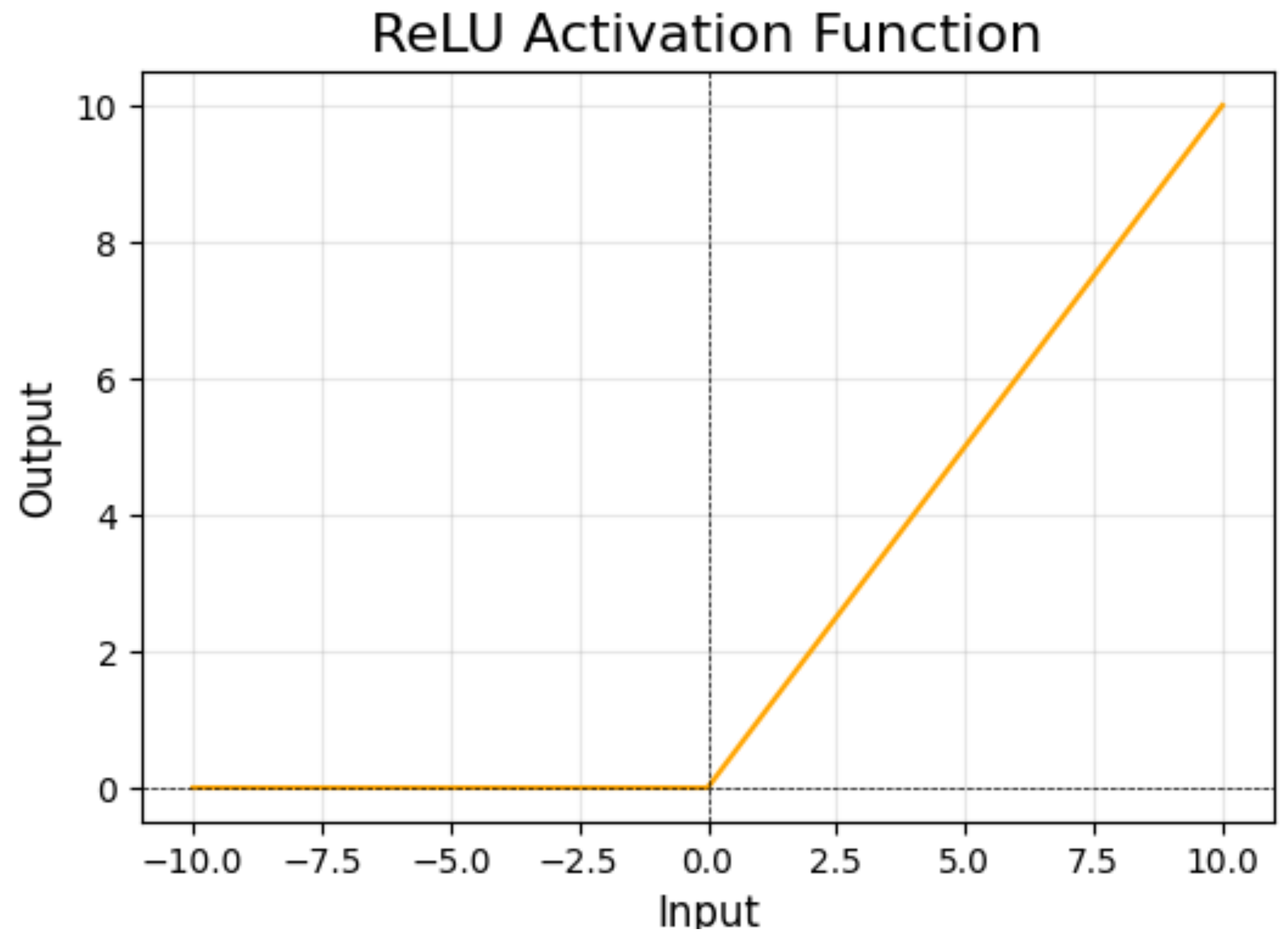
Linear:

$$Y = X$$



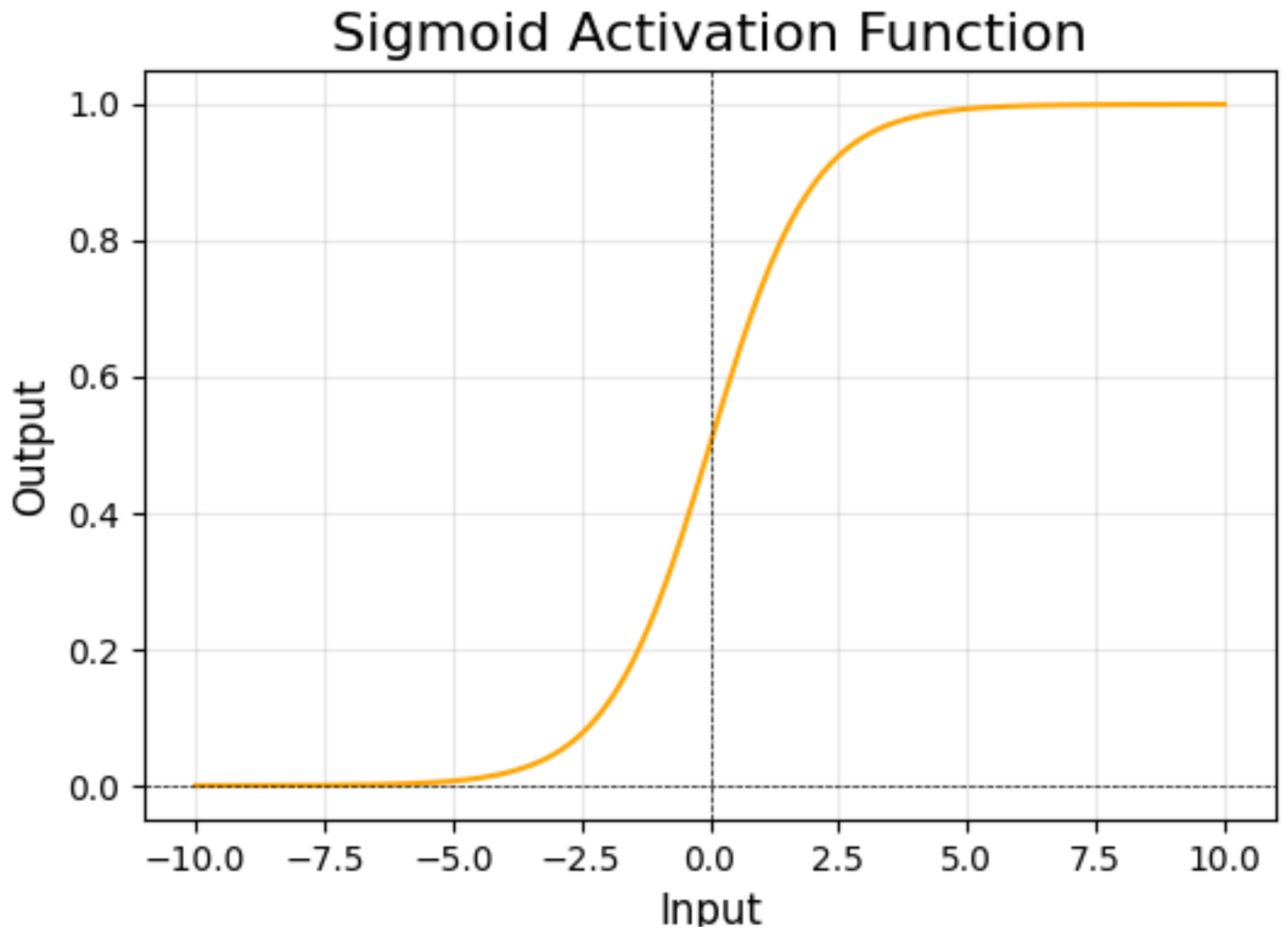
ReLu:

$$Y = X \cdot I(X > 0)$$



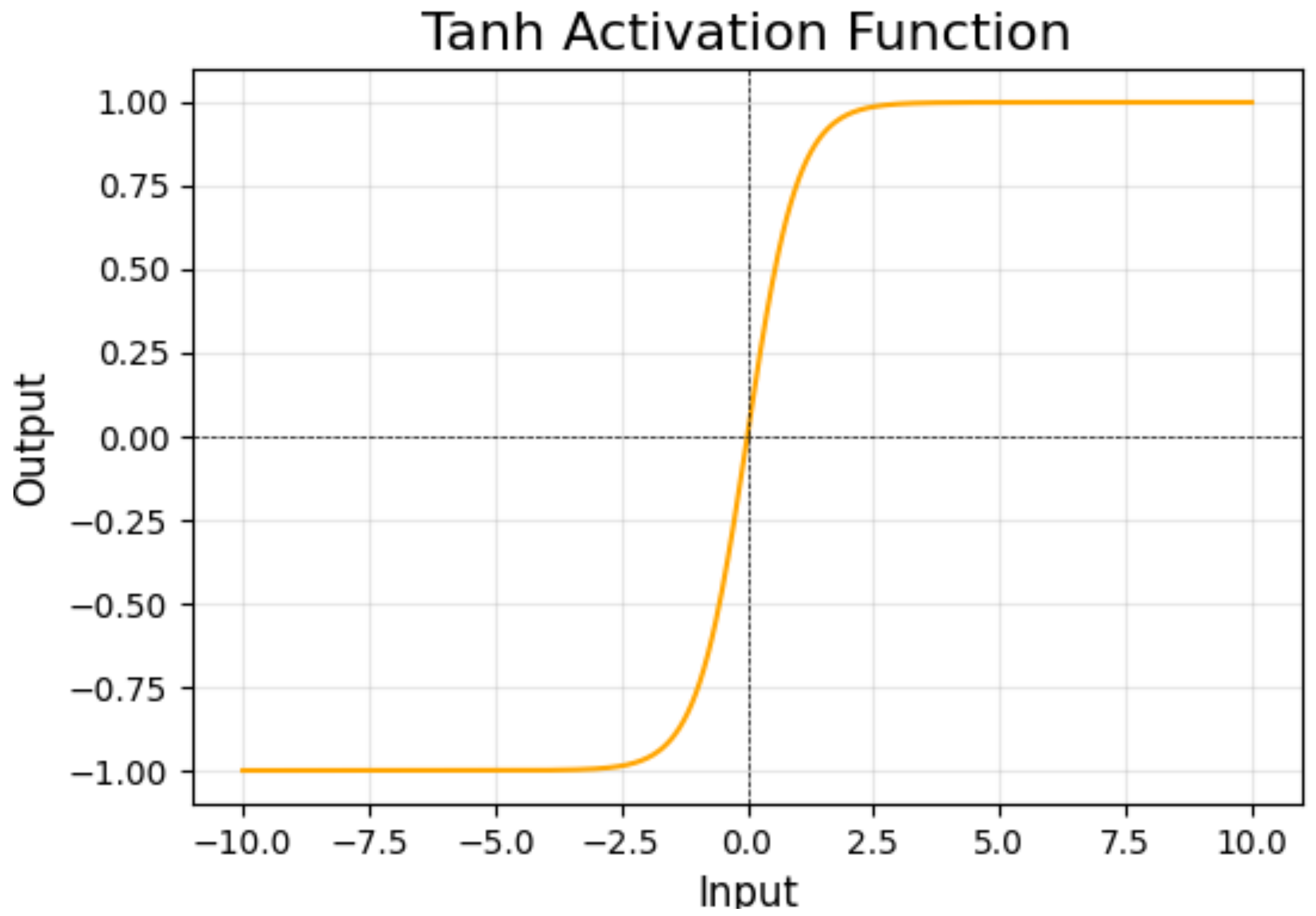
Sigmoid:

$$Y = \frac{1}{1 + e^{-x}}$$



Tanh:

$$Y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Architecture design

Simple | Architecture | Complex

Easier training, lower power | | Finer tuning, deeper structure

- Get a rough sense of the target function.
- Increase the number of width of the layers gradually.
- Choose non-linear activation functions properly (ReLU recommended)
- Try until no significant improvement is gained from increasing complexity.

Outline

1. Basic functions

2. Simulation examples

   A. General Architecture design

   B. NNGLS handles complex interaction

   C. NNGLS vs add-covariate approaches

3. Real data example

Compare with GAM

GAM (generalized additive models) is a common non-linear estimator.

$$m(X) = b_0 + \sum_{k=1}^{p} b_k(X_k)$$

Where $b_k()$'s are usually basis functions (for example B splines).

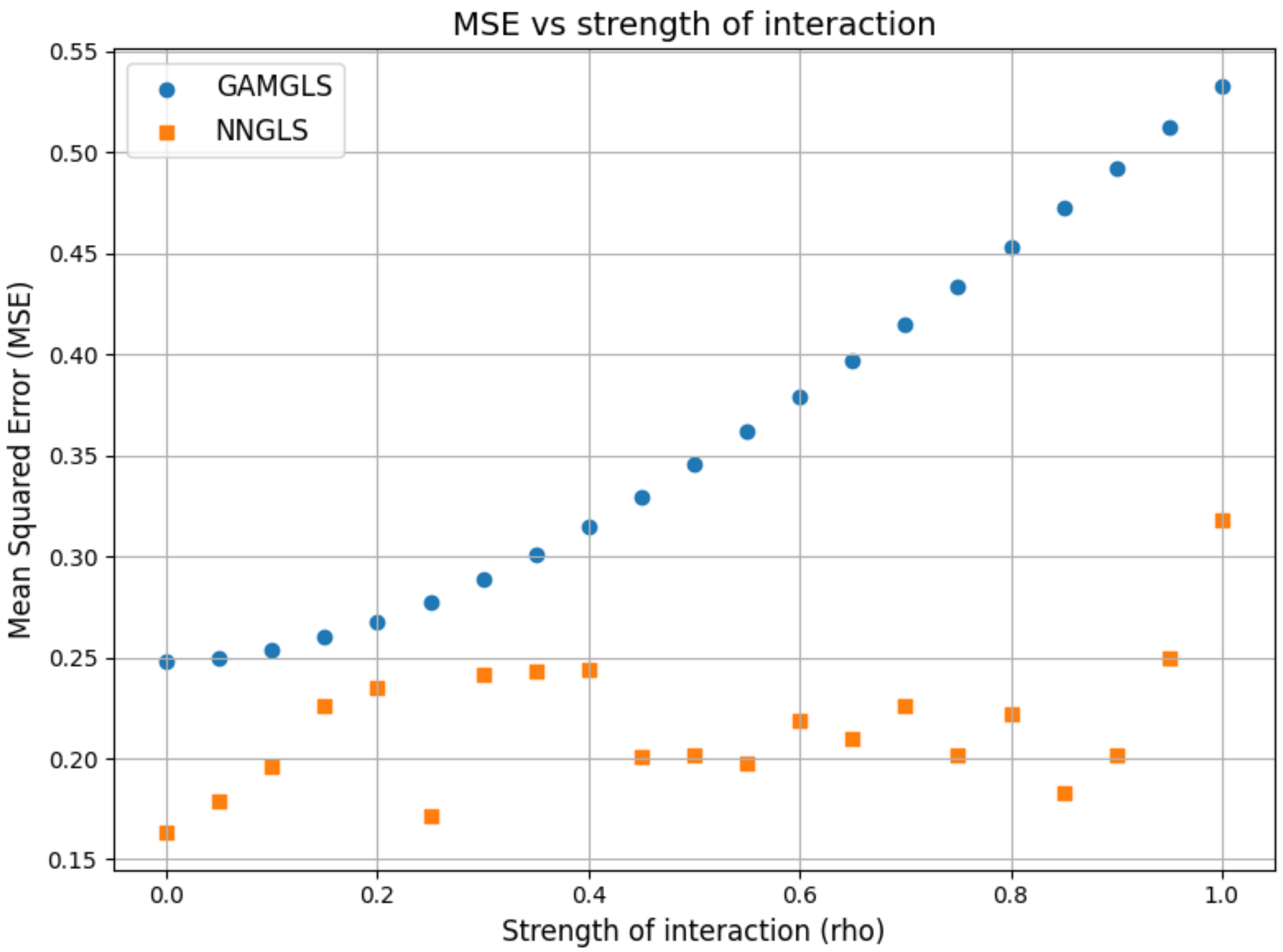
GAM assumes additive effects from $X_1, \cdots, X_p$.

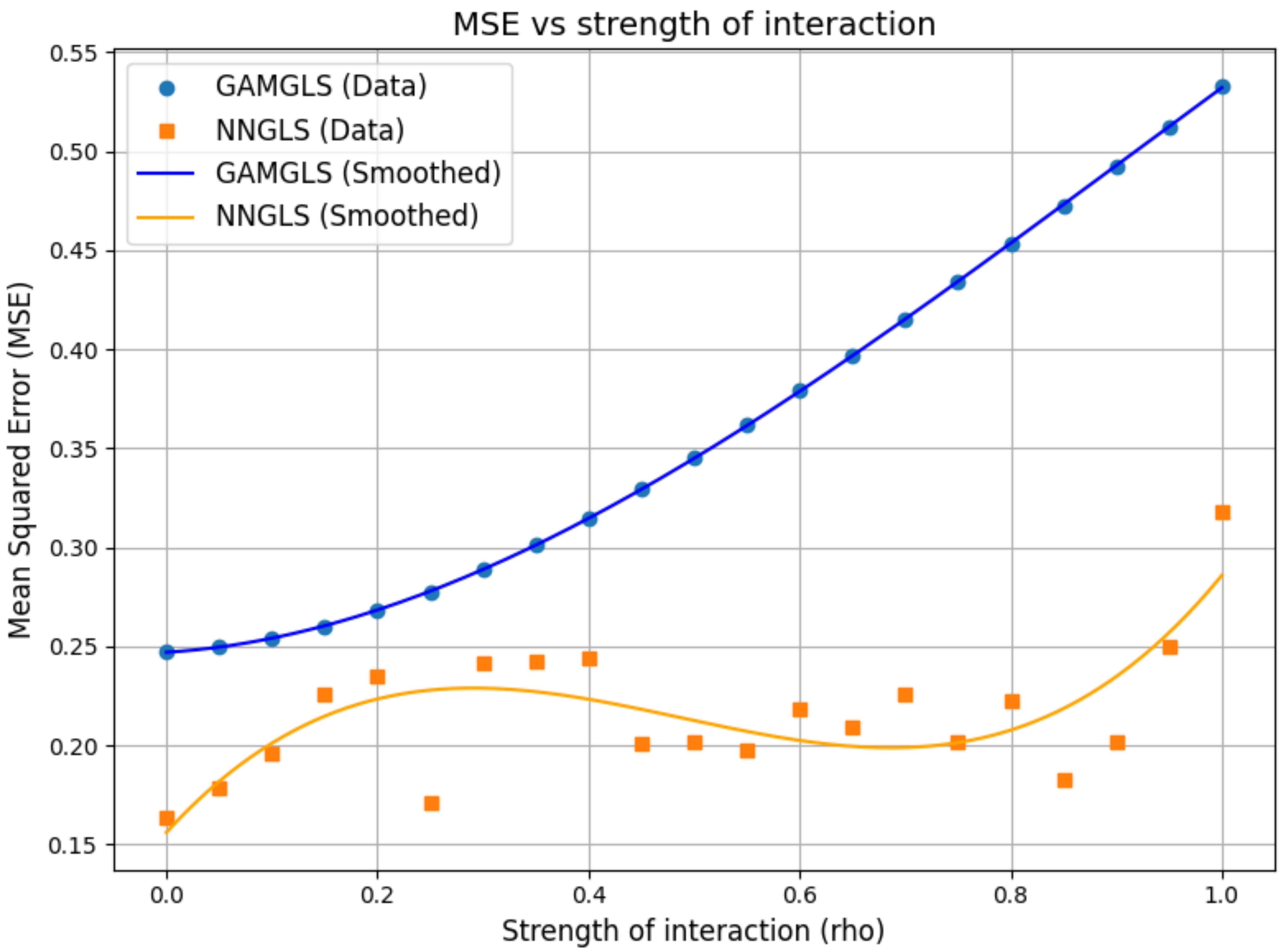
NN (NN-GLS) should outperform GAM (GLS version of GAM) by considering interaction terms.

GAM and the interaction term

Interaction

$$m(X) = \rho \boxed{\frac{10\sin(\pi X_1 X_2)}{3}} + (1-\rho)\frac{20(X_3-0.5)^2 + 10X_4 + 5X_5)}{3}$$





Outline

1. Basic functions

2. **Simulation examples**

    A. General Architecture design

    B. NNGLS handles complex interaction

    C. **NNGLS vs added-spatial-features approaches**

3. Real data example

Added-spatial-features approaches:

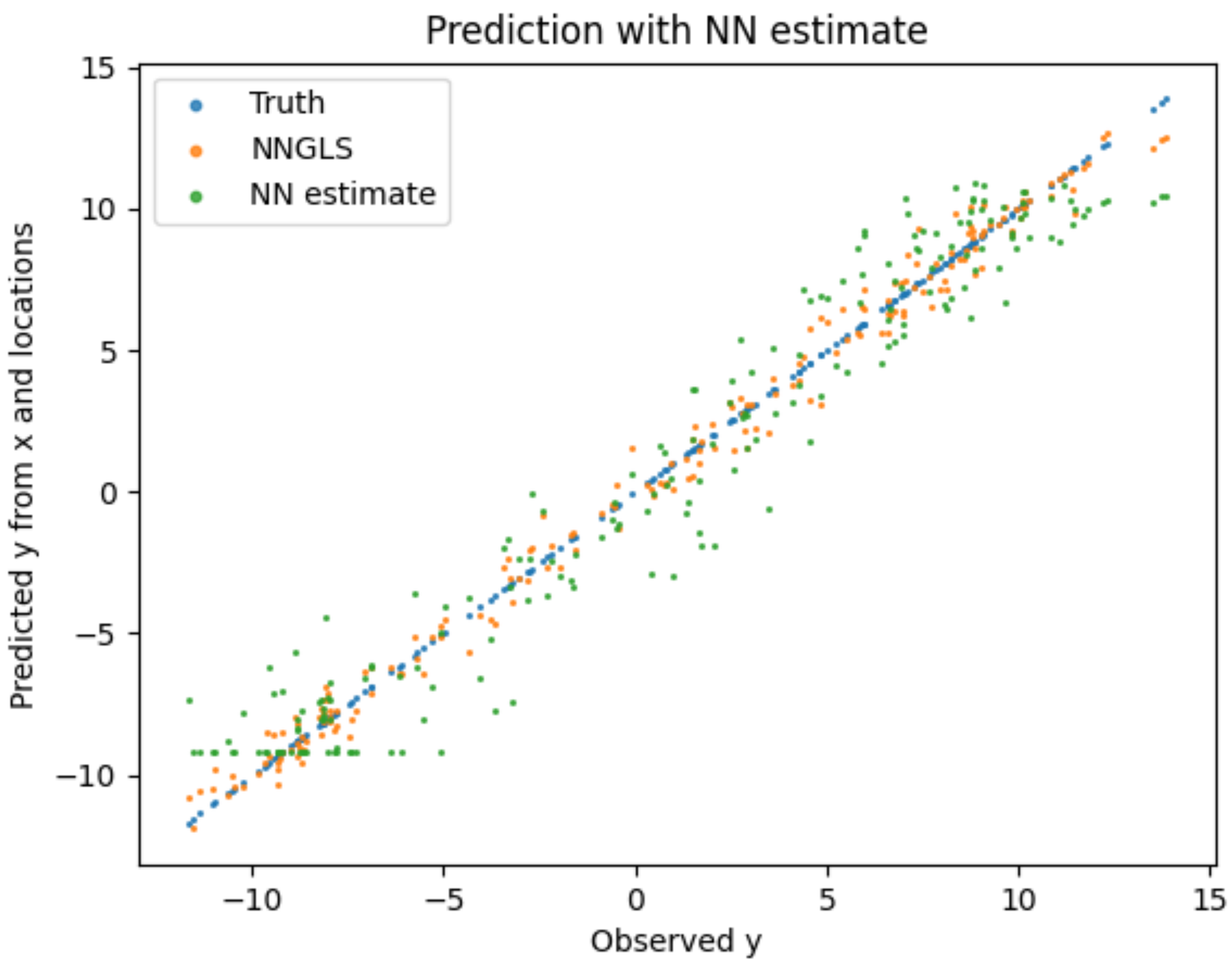
Model the spatial response $Y(X(s))$ as a fixed function of (X, s)

$$Y_i(s) = m\big(X_i(s)\big) + w(s) + \epsilon(s) = g(X_i, b(s)) + \tilde{\epsilon}(s)$$

Where $m(s)$ can be location, distance, or splines purely from s;

- $g(\cdot\,,\cdot)$ is used to predict at new location, but not able to separate fixed effect and spatial effect.
- Chen et.al. (2024) shows spline expansion is asymptotically equivalent to kriging (DeepKriging).

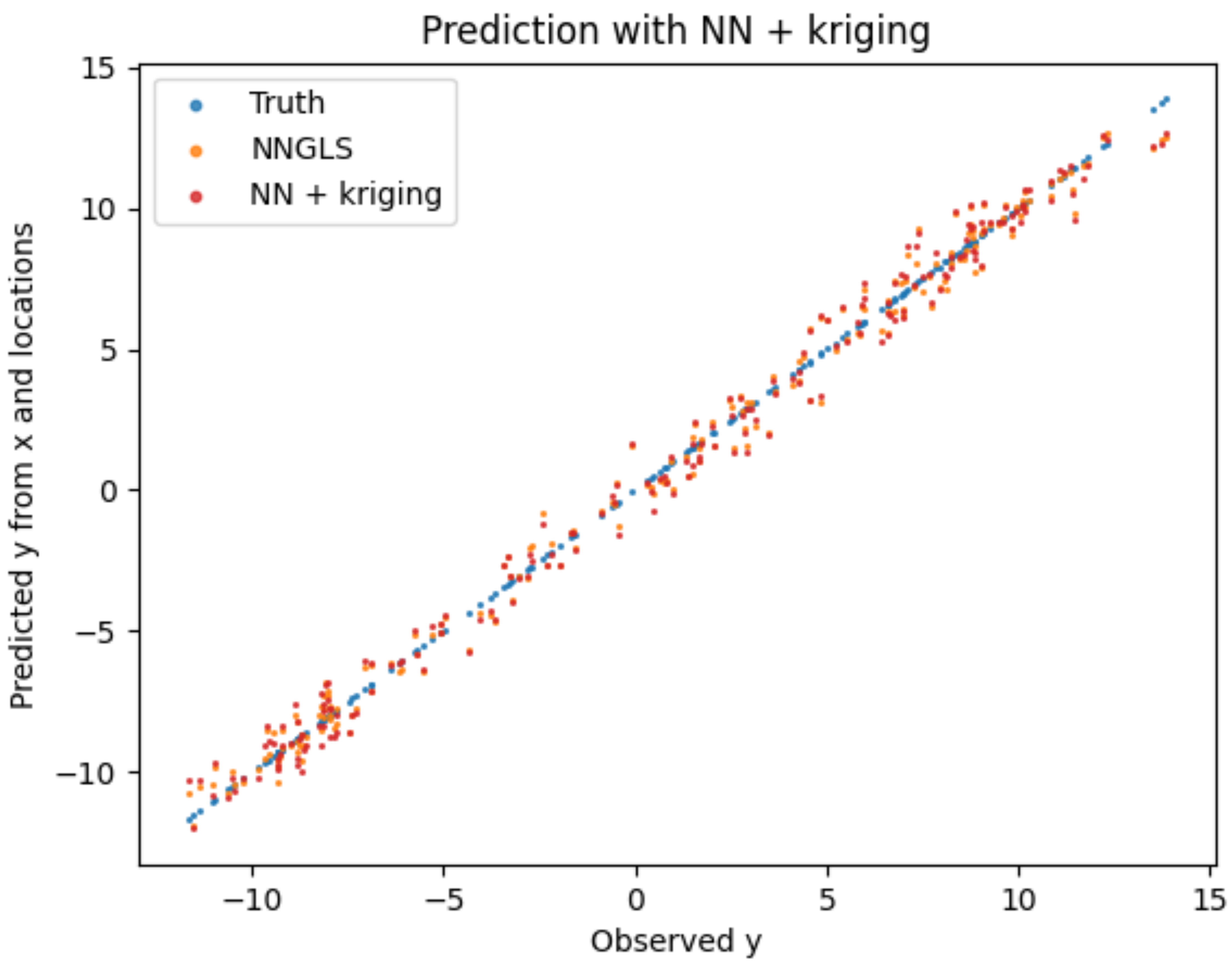
Versus added-spatial-features approaches:



Prediction vs Truth

```
RMSE nn-estimate:  2.98
RMSE nngls:  0.45
RMSE nn+kriging: 0.52
RMSE nn-add-coordinates: 2.84
RMSE nn-Deepkrig: 1.59
```

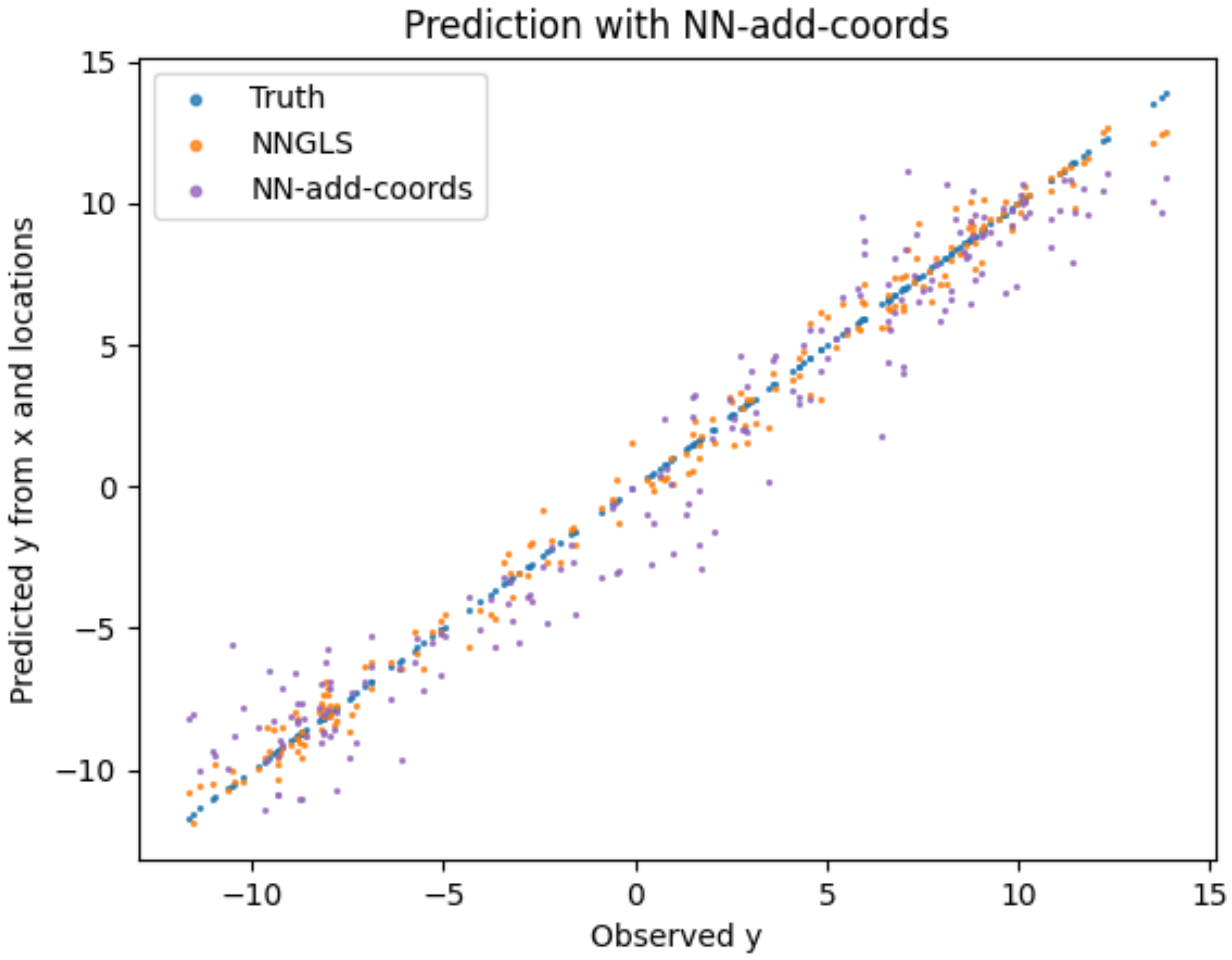
Versus added-spatial-features approaches:



Prediction vs Truth

```
RMSE nn-estimate:  2.98
RMSE nngls:  0.45
RMSE nn+kriging: 0.52
RMSE nn-add-coordinates: 2.84
RMSE nn-Deepkrig: 1.59
```

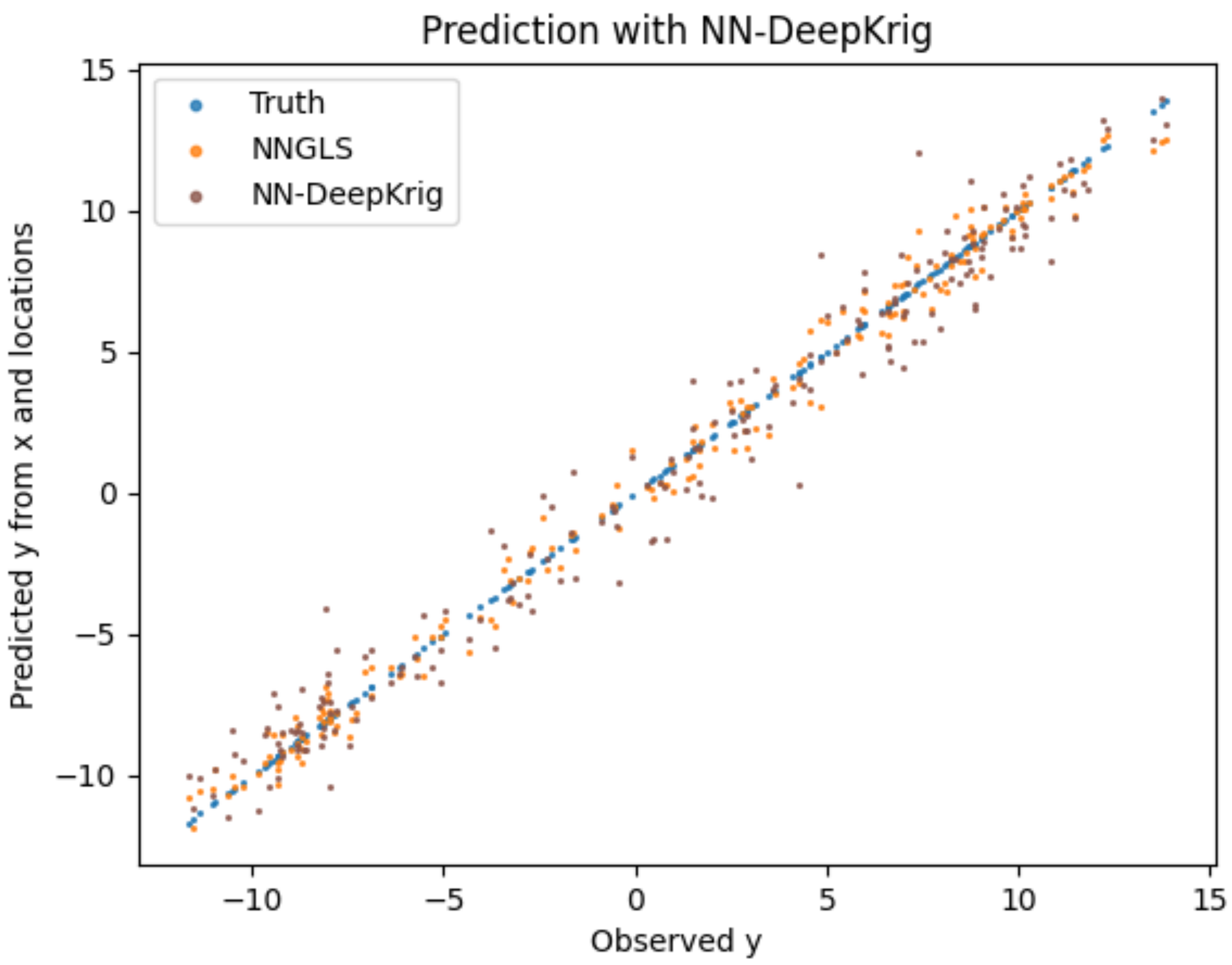
Versus added-spatial-features approaches:



Prediction vs Truth

```
RMSE nn-estimate:  2.98
RMSE nngls:  0.45
RMSE nn+kriging: 0.52
RMSE nn-add-coordinates: 2.84
RMSE nn-Deepkrig: 1.59
```

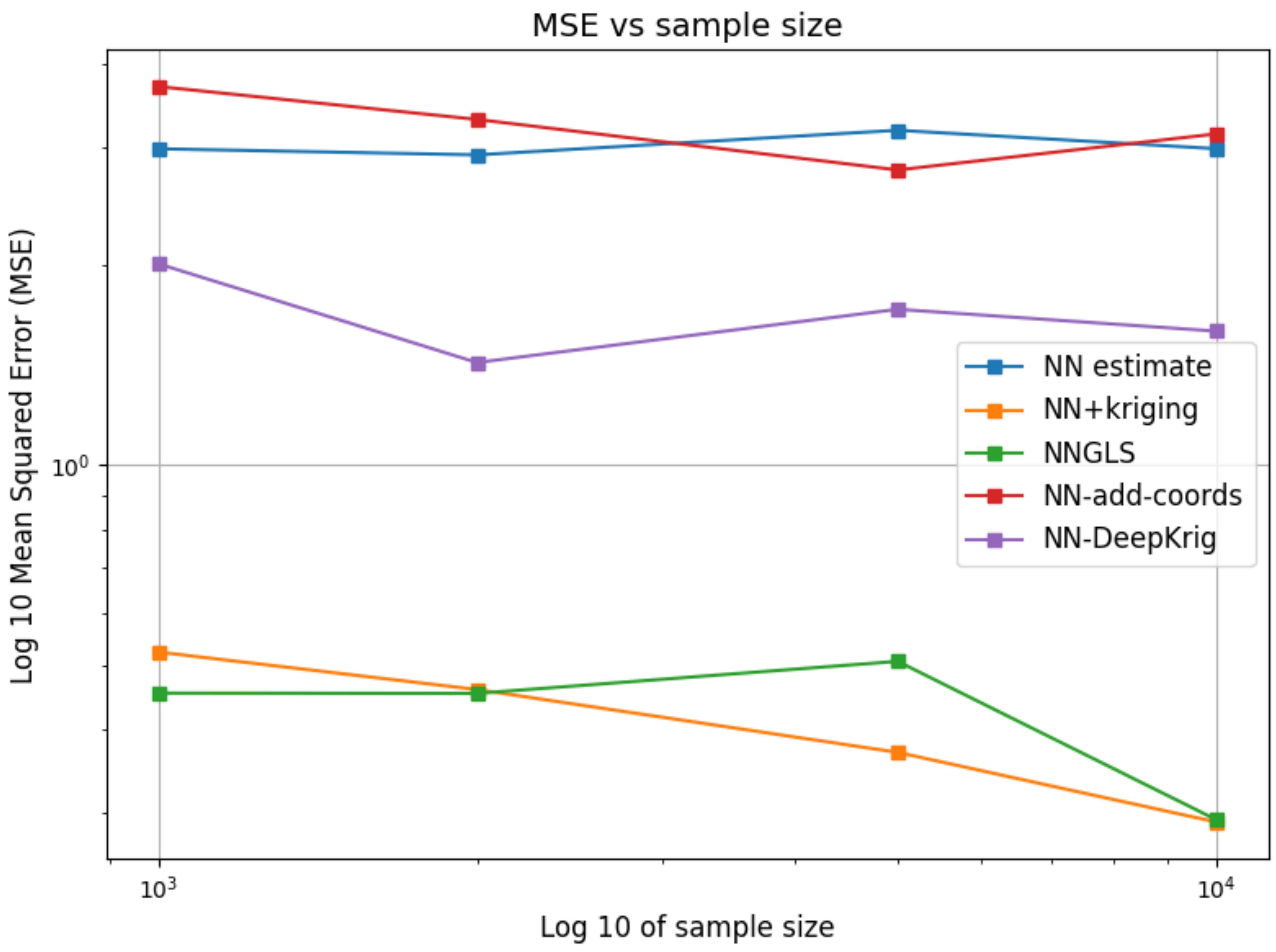
Versus added-spatial-features approaches:



Prediction vs Truth

```
RMSE nn-estimate:  2.98
RMSE nngls:  0.45
RMSE nn+kriging: 0.52
RMSE nn-add-coordinates: 2.84
RMSE nn-Deepkrig: 1.59
```

Versus added-spatial-features approaches:



Prediction performance agains sample size

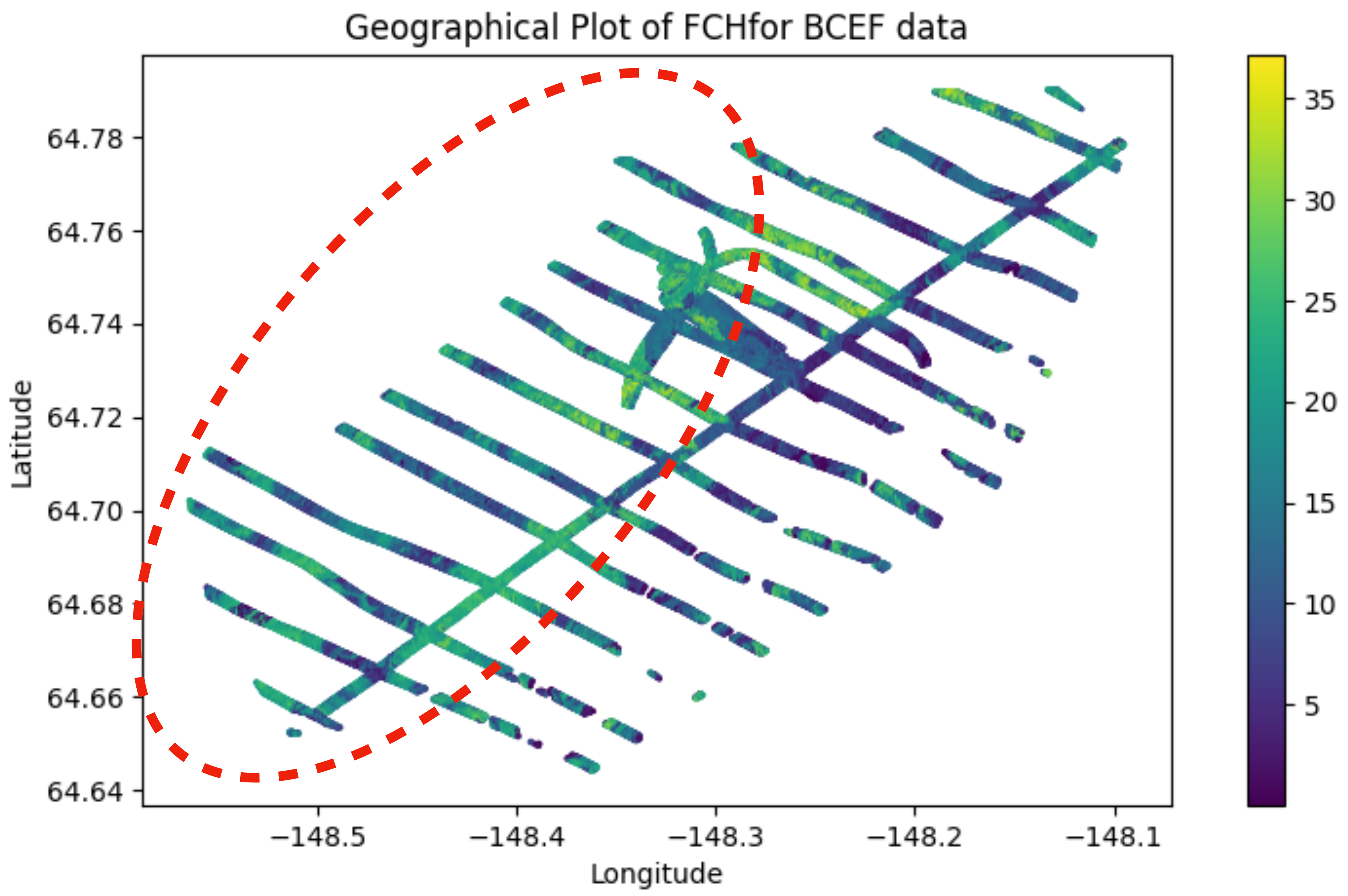
Outline

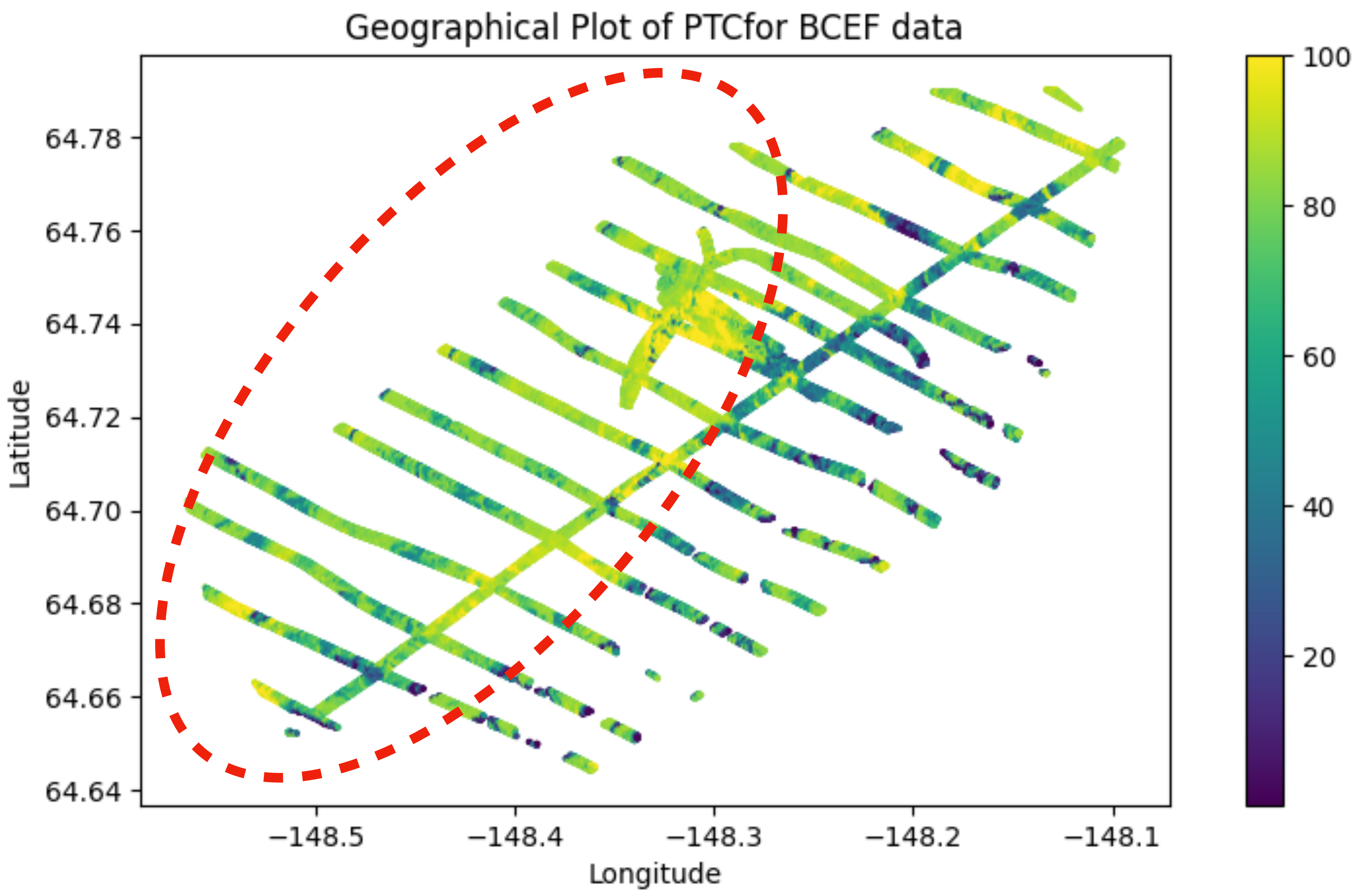
1. Basic functions
2. Simulation
3. **Real data example**

BCEF data: choice of "testing area"

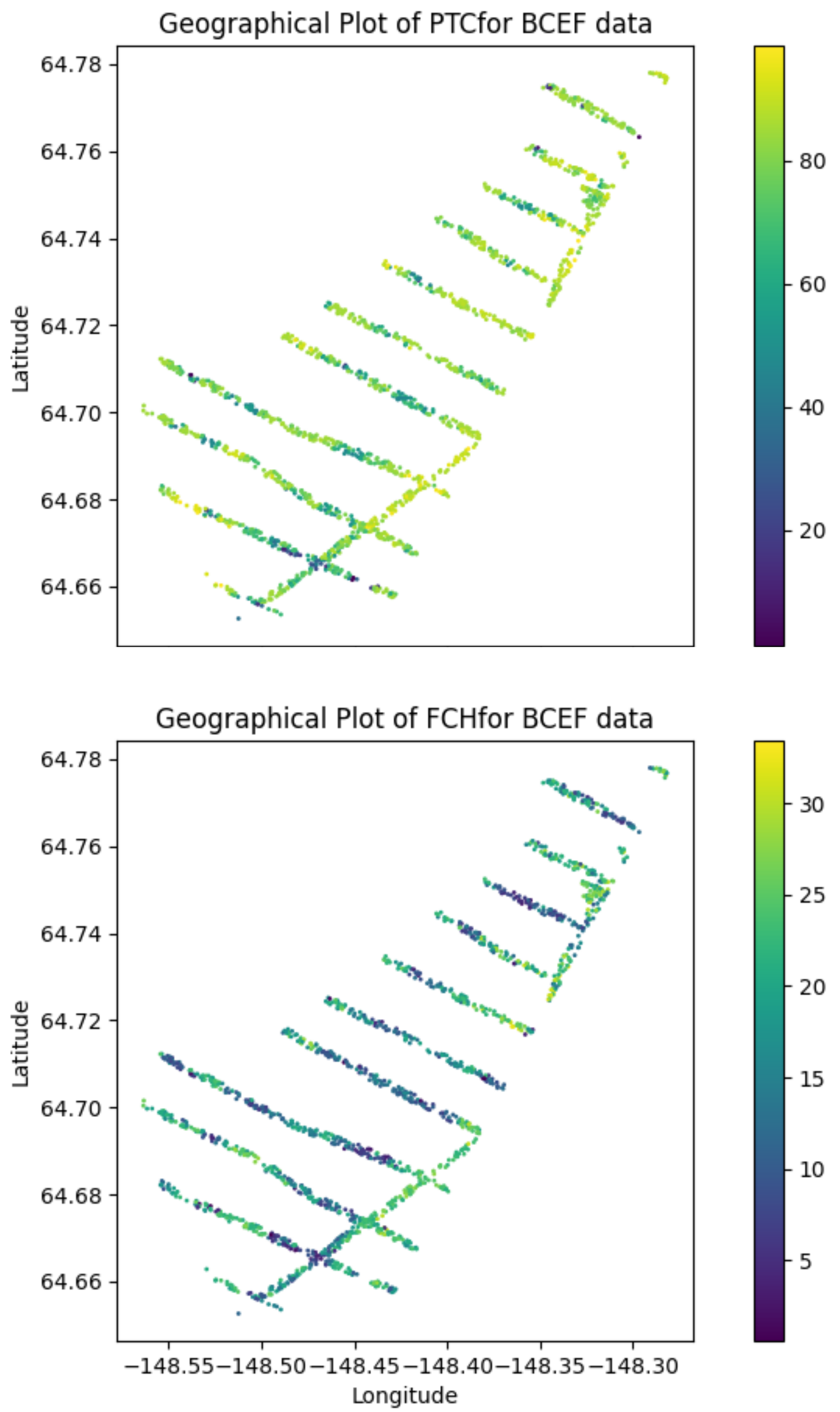
Sample size 30k (a subsample of the whole BCEF data).

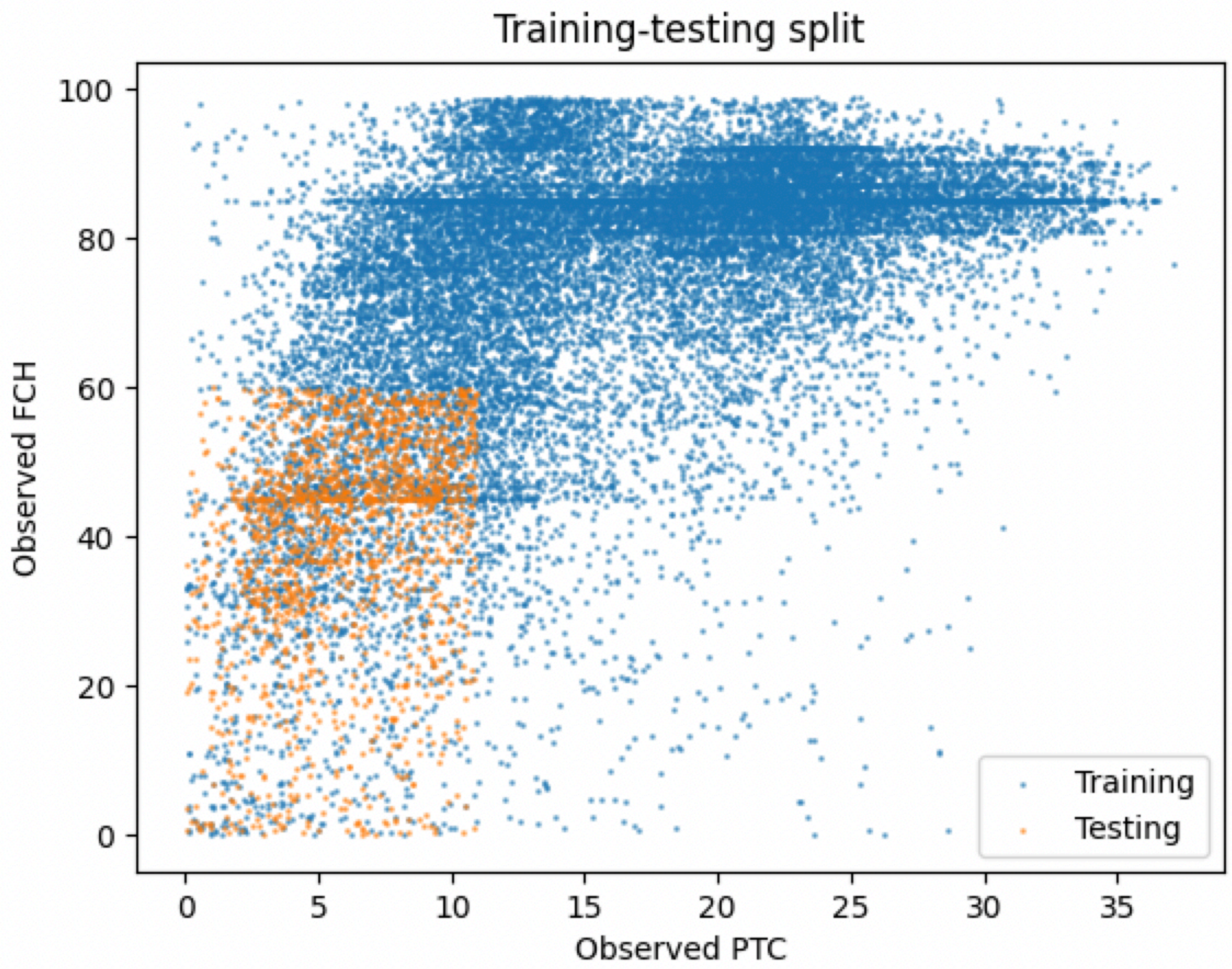
Running time: 3 minutes.





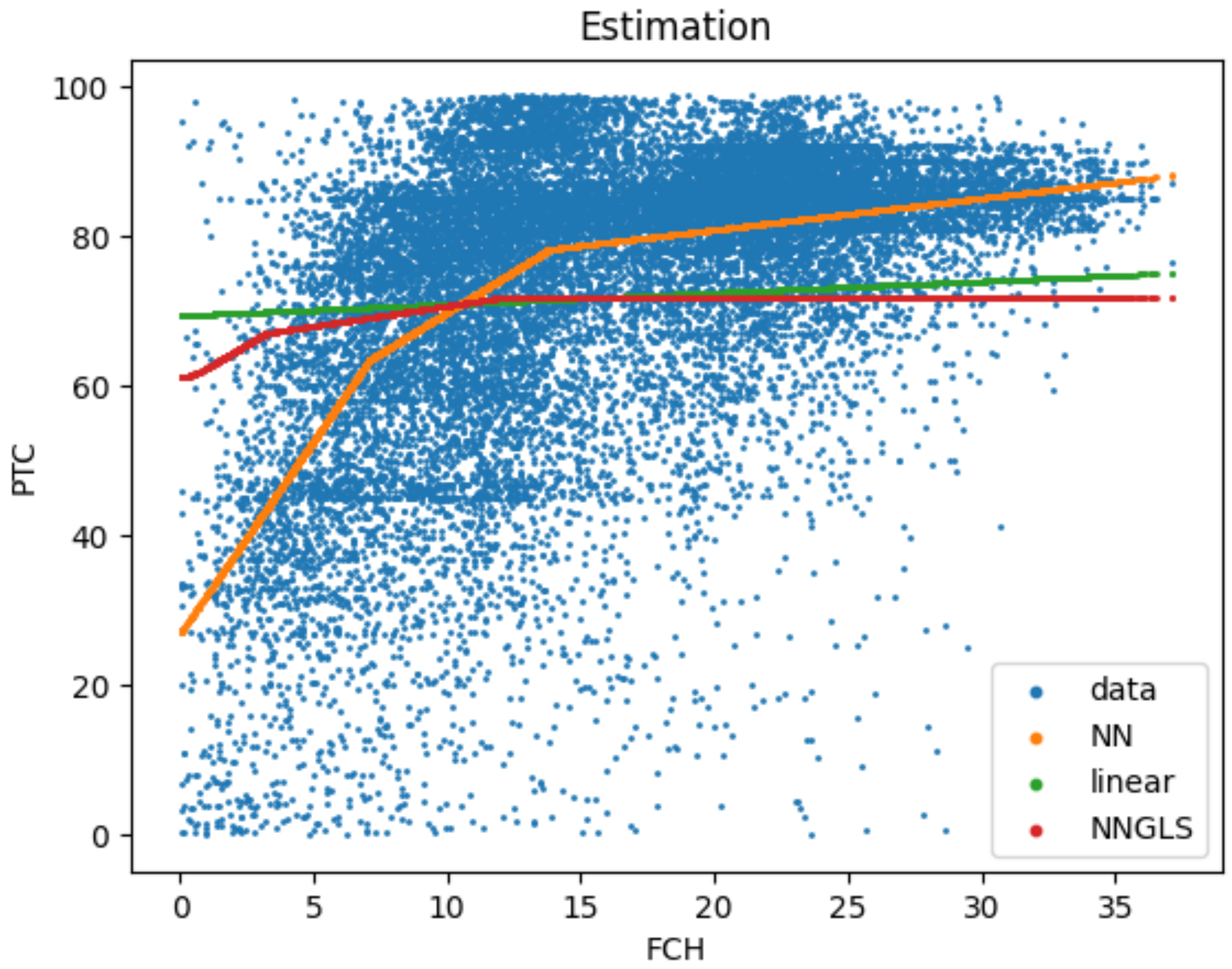
Training testing splitting:

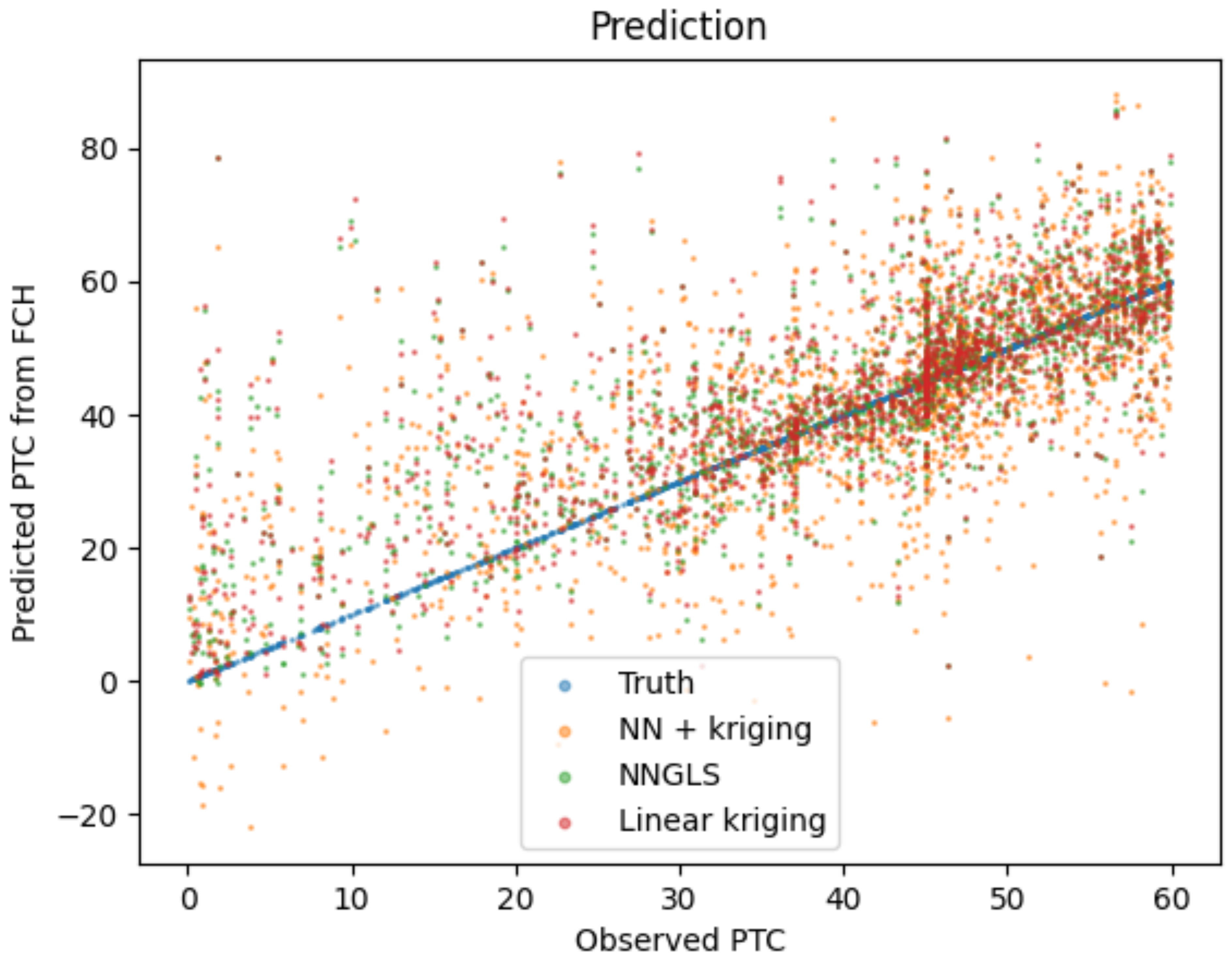




BCEF: special random split

PTC < 60%, FCH < quantile(FCH, 0.3), restricted in the testing area





BRISC (σ^2, ϕ, τ) = (461, 66.2, 0.001)

NN-GLS (σ^2, ϕ, τ) = (431, 70.3, 0.001)

NN-kriging (σ^2, ϕ, τ) = (265, 141.7, 0.037)

MSE:
BRISC 131.13
NNGLS 126.64
NN+kriging 177.97

References

Banerjee, S., Carlin, B. P., & Gelfand, A. E. (2003). *Hierarchical modeling and analysis for spatial data*. Chapman and Hall/CRC.

Chen, W., Li, Y., Reich, B. J. and Sun, Y. (2024), *Deepkriging: Spatially dependent deep neural networks for spatial prediction,* Statistica Sinica 34, 291–311.

Datta, A., Banerjee, S., Finley, A. O., & Gelfand, A. E. (2016). *Hierarchical nearest-neighbor Gaussian process models for large geostatistical datasets*. Journal of the American Statistical Association, 111(514), 800-812.

Zhan, W., & Datta, A. (2024). Neural networks for geospatial data. Journal of the American Statistical Association, (In press), 1-21.

Thanks