MITIGATING URBAN HEAT: HOW NON-STATIONARY MODELS CAN INFORM THE LENGTH SCALES OF INTERVENTION EFFECTS

Fall 2024 Independent Study Report

Previous Work

Calhoun's work on modeling Durham's urban heat island effect has been revolutionary. The causal inference nature of the model provides a point estimate of how much temperature will decrease in a certain location in the city when a heat-mitigating intervention is placed in said location. This model introduced the ability to influence urban planning policy given the relatively higher temperature in cities. Because the model can show not only the direct effect of the intervention on the temperature but also the indirect effect manifested in the interference of two points in space, policymakers can see the synergy of multiple interventions and determine the optimal distance between the centers of such interventions, ultimately maximizing the temperature decrease in cities needing more heat alleviation.

Last semester, my work attempted to extend this study by applying the model to Boston, a relatively cooler city with different geographic features, one of which is that it is a coastal city. The presence of the body of water in the field of study affected the structure of the unobserved confounding variable U. In addition, the absence of temperature data in the waters skewed the model performance towards the lands. Unfortunately, due to the traversal method of measuring the temperature of a car, bodies of water are off-limits.

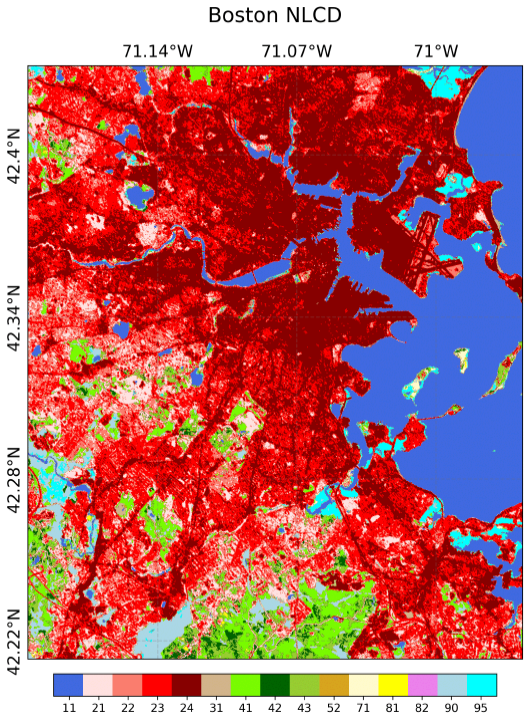


Figure 1. Land cover in Boston

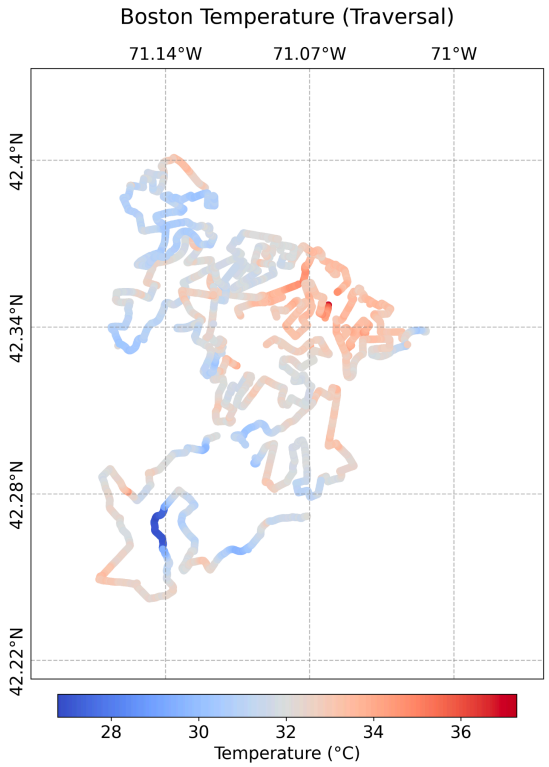


Figure 2.

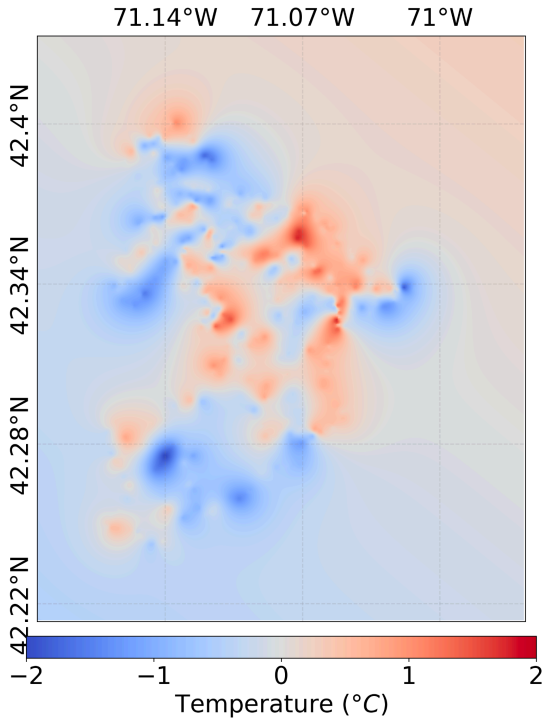
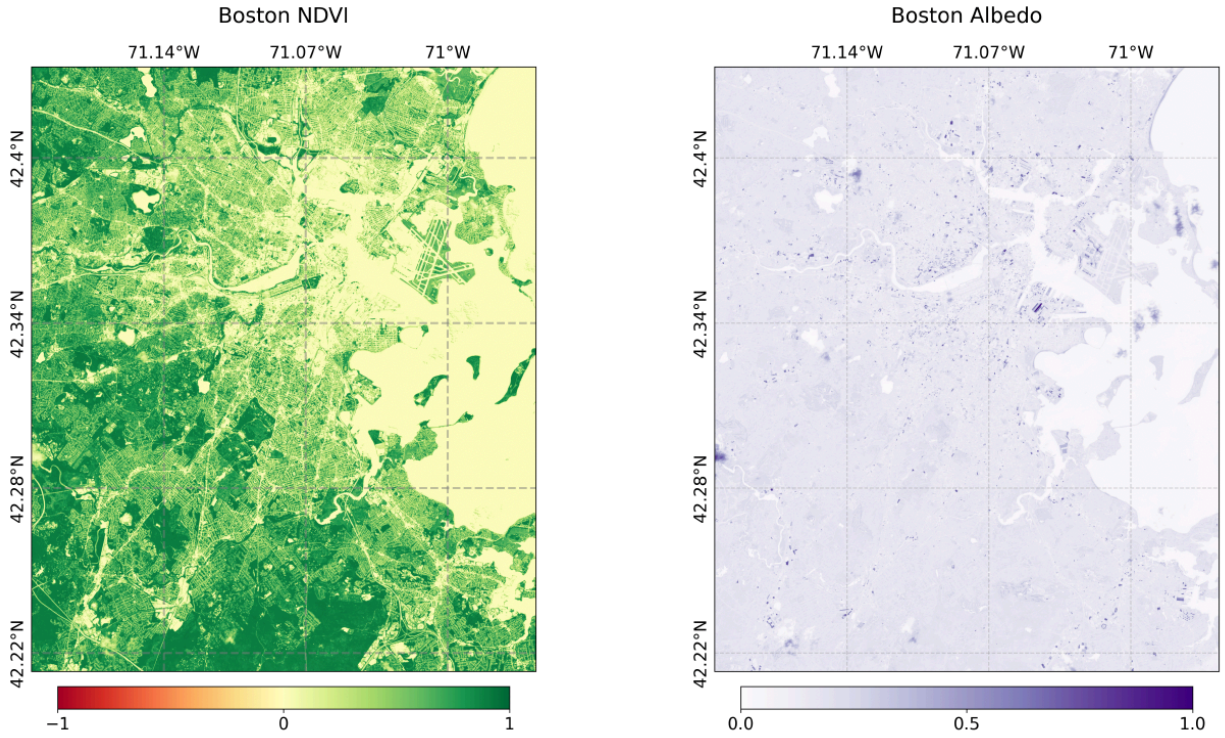


Figure 3. The unobserved confounding U as the model, trained on Boston data, predicted

Based on Figures 2 and 3, the model overestimates the relatively hotter areas, as opposed to the relatively cooler areas located in the southeastern portion of Boston where the model underestimates the air temperature. To reiterate, the unobserved confounding variable is parameterized as a Gaussian process using a stationary kernel, which means that it only has one length scale that defines the decay rate of the interference strength between any given two points in space. [1], [2], [3] This can be deemed a limitation for the bodies of water in the image, and in general, the variation differences of geographic features amongst localized areas. For instance, there is little to no variance in the ocean regarding NDVI, which is not the case on land. Moreover, in terms of albedo, we can only see variance where infrastructures are built, usually in the heart of Boston. These differences are discounted in our current model. Consequently, the model operates on a strong assumption that a tree in the city provides the same cooling effect as another tree in the suburbs to its neighboring points or areas.



Figures 4a, 4b. Visual differences in the distribution of NDVI and albedo in certain parts of Boston

This limitation was mentioned in [1] primarily highlighting the need to relax the constant, isotropic interference assumption. We know that the level of interference depends on space and time due to the changing urban morphology and weather conditions. This limitation also manifested in the difference in the model results between Durham and Boston. As seen in Figure 5, the learned parameters in Boston possess a low magnitude, indicating the interference level is not as high as that in Durham. Does this mean that length scales are not supposed to be fixed when learning the underlying spatial process within cities?

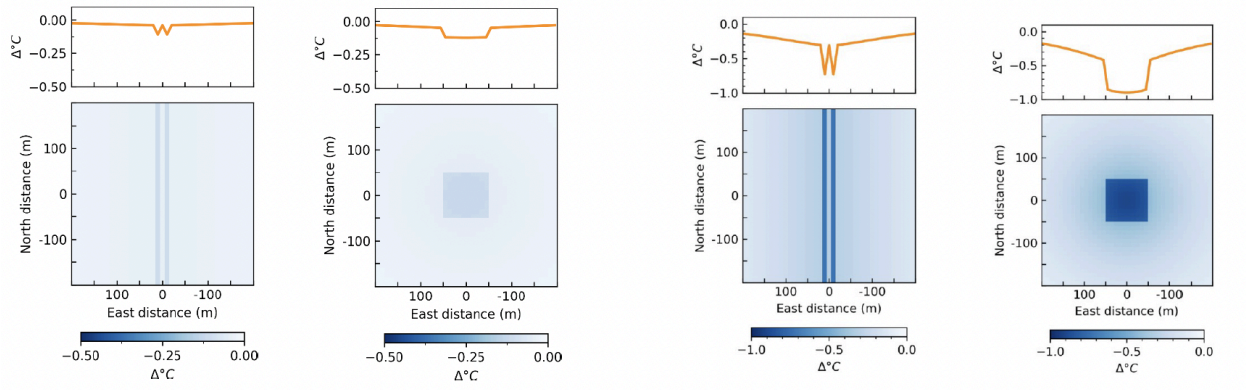


Figure 5. Estimates of intervention effects produced by the spatial causal inference in Durham and Boston

Exploring New Questions

Extending the model to Boston provided us with significant insights regarding the model's limitations. Moving forward, the questions in mind I am trying to explore are:

1. In the context of modeling the urban heat island effect in multiple cities in the US, is there a difference in using stationary and non-stationary kernels within the Gaussian process? If so, is the difference large enough to indicate a preference for non-stationary kernels, given its computational overhead[1]? [2]
2. Assuming the non-stationary nature of the relationship between the model covariates and air temperature, how can we incorporate modified Gaussian processes in our spatial causal inference model? [1]

Within the exploration of the first question, the next logical step for the project was to prove that there exists a varying level of interference across cities with different geographic features. This involves analyzing the kernel behavior of the Gaussian process applied to the cities across the US. Currently, there are over 80 cities and counties that possess readily available air temperature data as part of NOAA's Urban Heat Island Mapping Campaign. [4] To enable this comprehensive analysis, it is imperative to build a pipeline containing scripts that make downloading data and training models computationally efficient.

The next step would be to increase the granularity of the study by partitioning cities into well-defined regions, showing that varying optimal length scales exist in a city. This proves the strong non-stationary nature of the dataset, bringing us to explore other methods to learn this said nature of the dataset like using deep kernels and deep Gaussian processes (GPs). [2]

[1] Gaussian processes have a cubic time complexity, O(n^3) where n is equal to the number of data points.

As such, the primary objectives for this semester were to:

1. collect data from multiple cities;
2. develop a pipeline to fit a stationary Gaussian process to any city;
3. analyze and compare the results among cities; and
4. fit a baseline non-stationary model(s) via a parametric function defined in [2]

Related Literature

We have demonstrated how causal inference could inform urban planning policy by providing point estimates of temperature reduction due to heat-mitigating interventions. This model also highlighted the synergy of multiple interventions and the importance of spatial interference effects. However, extending the work in Boston, MA revealed challenges associated with coastal cities, such as the influence of water bodies and the lack of temperature data over water. These studies underscore the need for non-stationary kernels to address geographic heterogeneity.

Effects of urban morphology on urban heat island intensity

Currently, our model is designed to address such heterogeneity with the land cover covariates X. Within the Gaussian process regression portion of the training, the unobserved confounding variable U is expected to capture meteorological variables including wind direction and speed. Wind circulation in a city heavily influences the intensity of the urban heat island in a city. There is a study revealing that the increase of wind speed by 1 m/s reduces the mean temperature in an urban heat island by 0.14 deg. C. [5] Thus, we know that the mechanics of air circulation in a city influence the temperature in a city.

The movement of air is influenced by the objects in its path. [5] highlights that urban morphology, particularly building density, plays a significant role, as wind speed tends to decrease with an increase in the buildings' coverage ratio. While computational fluid dynamics-based models can simulate the high variability of wind speed within a city, such approaches are counterproductive to our goal of developing a simple and interpretable model.

Still, in our model, the use of a single, city-wide length scale for the exponential kernel in the Gaussian process ignores the smooth facilitation of heat propagation in open areas and emphasizes the less horizontal mixing in areas with high surface roughness. This is precisely why in Boston, the cooling effect in highly vegetated areas is being underestimated, while the cooling effect in highly developed areas is being inflated.

Non-stationary nature of environmental data

This phenomenon is not new to the urban heat island literature and the general climate science literature. This is often referred to as the non-stationary nature of climate data over space, and

there have been studies that employ artificial intelligence technology to address the variability of certain factors across space and time. [3], [6], [7] Environmental data often exhibit significant non-stationary characteristics due to variations in both spatial and temporal dimensions, making stationary models insufficient for capturing their complexities. Real-world phenomena, such as ozone levels, wind speeds, and temperatures, are influenced by diverse local geographic features and temporal conditions that vary across locations. Stationary models, which assume uniform covariance based solely on distance, fail to account for these dynamics.

Non-stationary covariance functions address this limitation by allowing the relationships between data points to vary across space and time. [2] For instance, latent length scales in non-stationary models enable the correlation of data to differ based on the local environment—densely built areas exhibit smoother changes over short distances, while less developed or diverse regions might experience larger-scale variations. [7] This difference manifests in the magnitude of the length scales in our kernel. Lower length scales are usually found in areas with high building density and higher length scales are usually observed in more open and less developed spaces.

Non-stationary kernels

To learn the multiple length scales present in a city, we rely on non-stationary kernels $K_{NS}(x_i, x_j)$ that do not only depend on the distance between two points $\left|x_i - x_j\right|$ but also the respective location in the input space explicitly. This is done by subjecting multiple kernels into convolutions and additions. [2], [7]

$$K_{NS}(x_i, x_j) = \int_{\Re^2} K_{x_i}(u) K_{x_j}(u) du$$

One instance of a non-stationary kernel is choosing specific points u that allow the resulting covariances found in the kernel to be dependent on the length scale. This approach is referred to as a "parametric" non-stationarity. [2] Here, we define the points x_k such that:

$$K_{NS}(x_i, x_j) = g(x_i) g(x_j) k_{stat}(x_i, x_j)$$

, where

$$g(x) = \sum_{k=1}^{N} c_k \beta(x_k, x) \text{ and } \beta(x_k, x) = e^{-\left|x_k - x\right|^2 / w}$$

Because of the parametric nature of the non-stationary kernel above, we can define discrete points in space and learn the length scale w for each point.

Methodology

Collect data from multiple cities

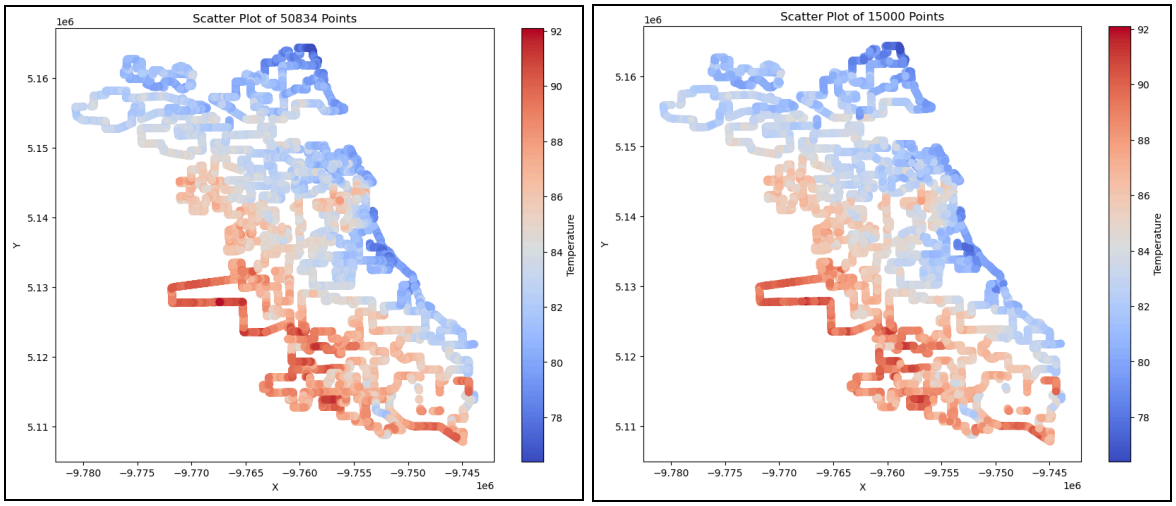
The experiments began with the collection of geospatial temperature data for multiple cities from NOAA's Urban Heat Island Mapping Campaign. The shapefiles contain spatial data points such as latitude, longitude, and temperature observations. These data are processed and converted into a uniform coordinate reference system (EPSG=3857) to ensure compatibility across cities. A pipeline was developed to extract and preprocess these datasets into a format suitable for Gaussian Process modeling. By focusing on 20 cities, the analysis aims to uncover patterns of spatial heterogeneity in urban heat island effects and test the adaptability of stationary and non-stationary models across varied urban morphologies.

Develop a pipeline to fit a stationary Gaussian process to any city

To fit a stationary Gaussian Process to temperature data from any city, a pipeline was developed and implemented in Python using libraries such as GeoPandas, NumPy, and PyTorch. The pipeline involves preprocessing raw geospatial data, standardizing input features, and normalizing temperature values. The stationary Gaussian Process is implemented using GPyTorch, which provides a scalable framework for exact GP inference. This pipeline applies K-fold cross-validation (k=5) to split the data into training and validation sets, ensuring robust evaluation of model performance. The learned stationary GP provides baseline estimates of length scales, which describe how temperature varies spatially in each city. *See Appendices C and D for the code used in the pipeline and a sample log output file from running the script.*

GPyTorch

The stationary Gaussian Process was trained using the GPyTorch library, leveraging its efficient implementation of Gaussian Process models for large-scale datasets. The pipeline utilizes GPyTorch's **`ExactGP`** class with a FixedNoiseGaussianLikelihood to account for observational noise in the temperature data. The noise was set at 0.05. The model was set to have a zero mean since the temperature and coordinates are normalized, whereas the covariance is defined by a Matern kernel, which assumes stationary spatial relationships. The training process optimizes the marginal log-likelihood using Adam, and the results include the learned length scales, which quantify the smoothness of spatial variations. The number of data ranges from around 10k to 50k. Even using a GPU on Google Colab, training a Gaussian process on more than 20k data points lasts more than 10 minutes for only one epoch. This is why only 20k random data points were chosen to be used in training the Gaussian process regression model. Figure 6 shows that choosing only a subset of the available data points only means a lower resolution. This, however, does not impose a significant change in the representation of available air temperature data in the traversal.



Figures 6a and 6b. Scatter plot of available air temperature data in Chicago

Analyze and compare the results among cities.

Once the stationary Gaussian Process models are trained, their results are analyzed and compared across multiple cities. Key metrics, such as average length scales in latitude and longitude directions, are evaluated to understand spatial variations in urban temperature patterns. The analysis also examines the R^2 values obtained from cross-validation to assess the models' goodness-of-fit. By comparing results among cities with varying levels of development, vegetation, and geographic features, insights are gained into the limitations of stationary models and the necessity of incorporating non-stationary kernels for capturing intra-city and inter-city heterogeneity.

To make sense of the variance of length scales among cities, the developed land area percentage was computed from the US Geological Survey (USGS) land cover data. The number of pixels classified as class 21-24, which indicate varying degrees of development, was divided by the total number of pixels found in the city. This is only a rough estimate; there is still a need to define the boundaries of the city and focus on the areas near the traversal data in the future.

Fit a baseline non-stationary model(s)

To address the limitations of stationary models, a baseline non-stationary Gaussian Process model was implemented. The non-stationary kernel incorporates varying length scales across spatial dimensions, enabling the model to capture spatial heterogeneity. Using GPyTorch, the non-stationary model builds on the stationary pipeline by introducing adaptive kernels that allow relationships to vary based on location. For example, cities with diverse urban morphologies benefit from localized length scales that adapt to varying levels of development, vegetation, and other environmental factors. These non-stationary models provide deeper insights into the spatial interference effects of urban heat mitigation strategies, overcoming the oversimplifications inherent in stationary models.

Results and Discussion

Table 1. Varying learned length scales (in m) learned by the Gaussian process and the percentage of developed land across cities in the US

	City	Developed Land (%)	avg_r2	avg_length_scale_lat	avg_length_scale_lon
0	Albuquerque	58.151796	0.692405	133.363246	128.842340
1	Atlanta	89.896299	0.951403	2284.323884	2543.104557
2	Baltimore	59.315466	0.910260	1201.644736	1506.423088
3	Boston	85.167840	0.944027	1883.587032	2546.783981
4	Boulder	56.010666	0.913970	723.821369	783.376625
5	Chicago	75.309272	0.959714	3347.563894	5144.394571
6	Detroit	76.005453	0.973727	1102.439372	697.271994
7	Durham	58.902319	0.968236	851.147800	795.692532
8	Houston	77.403930	0.978040	1756.959841	939.118726
9	Kansas	86.593013	0.809622	66.061205	136.489363
10	Las Vegas	47.930647	0.945271	5940.768807	5461.241298
11	Los Angeles	78.940540	0.937691	3460.353745	3130.311412
12	Miami	90.797372	0.987842	1606.636514	1444.118909
13	Nashville	78.237762	0.942497	2370.178449	2498.672705
14	New Orleans	54.887408	0.933614	2575.452126	2087.512877
15	NYC	83.353475	0.924867	763.051377	1008.741148
16	Oklahoma	55.235286	0.959531	5887.402478	6177.863241
17	San Francisco	57.360130	0.966325	2237.527298	2336.492919
18	Seattle	43.309175	0.948367	4162.298029	5188.619780
19	Washington DC	89.328412	0.938826	1596.749067	1858.900111

Table 1 shows the results for the Gaussian process regression modeled on 20 cities in the US. Here, we can see that cities' percentage of developed area spans from 43% to 90%. The city-wide length scales also range from around 60 meters to 6000 meters. As shown in Figure 7, a scatter plot was generated to see if there is an existing association between the length scale and developed land area percentage.

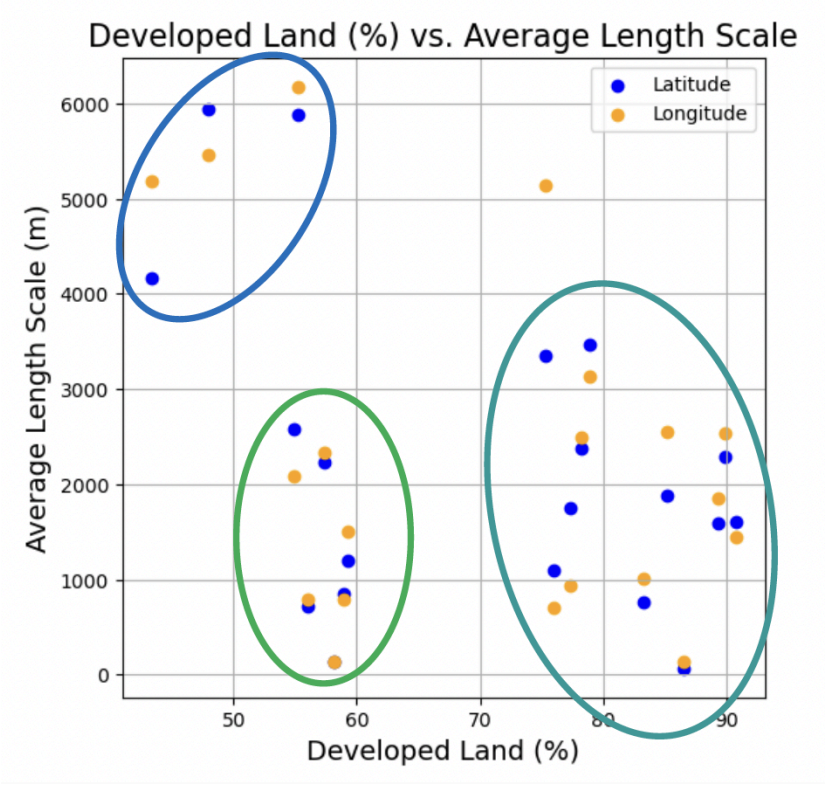


Figure 7. Three types of cities are depicted by the scatter plot of developed land percentage vs. average length scale of the longitude and latitude

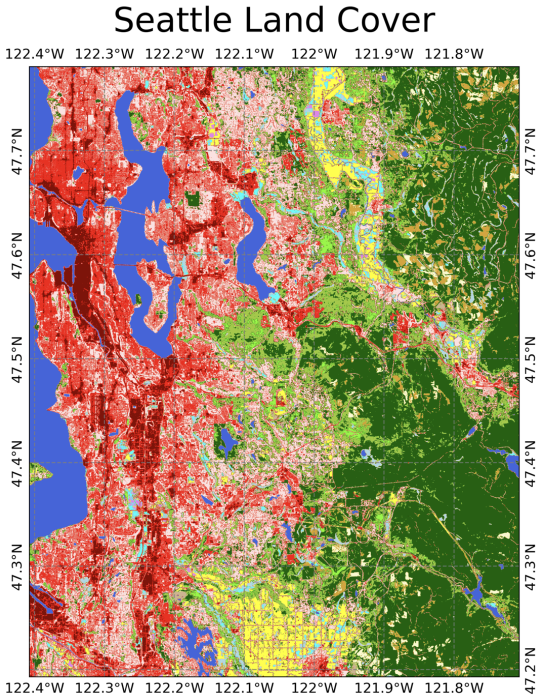


Figure 8. Land cover data of Seattle, a Type 1 city

There is no apparent linear association between the two variables. However, we can see three well-defined clusters in the data that have interesting geographic similarities within the group and differences among the groups.

Type 1 Cities (High Length Scales, Low Developed Land Percentage)

The Type 1 cities agree with physics: there is more horizontal mixing in open areas due to smooth wind circulation. These cities include Las Vegas, Oklahoma, and Seattle. These cities typically have their vegetated areas and developed areas more concentrated.

Type 2 Cities (Low Length Scales, High Developed Land Percentage)

Type 2 cities also support the fact that there is less horizontal mixing in areas with high surface roughness due to lesser wind speeds. These cities typically include the largest metropolitans in the US including Chicago, New York City, Boston, Los Angeles, Washington, etc. Like Type 1 cities, their developed areas are concentrated in a single cluster.

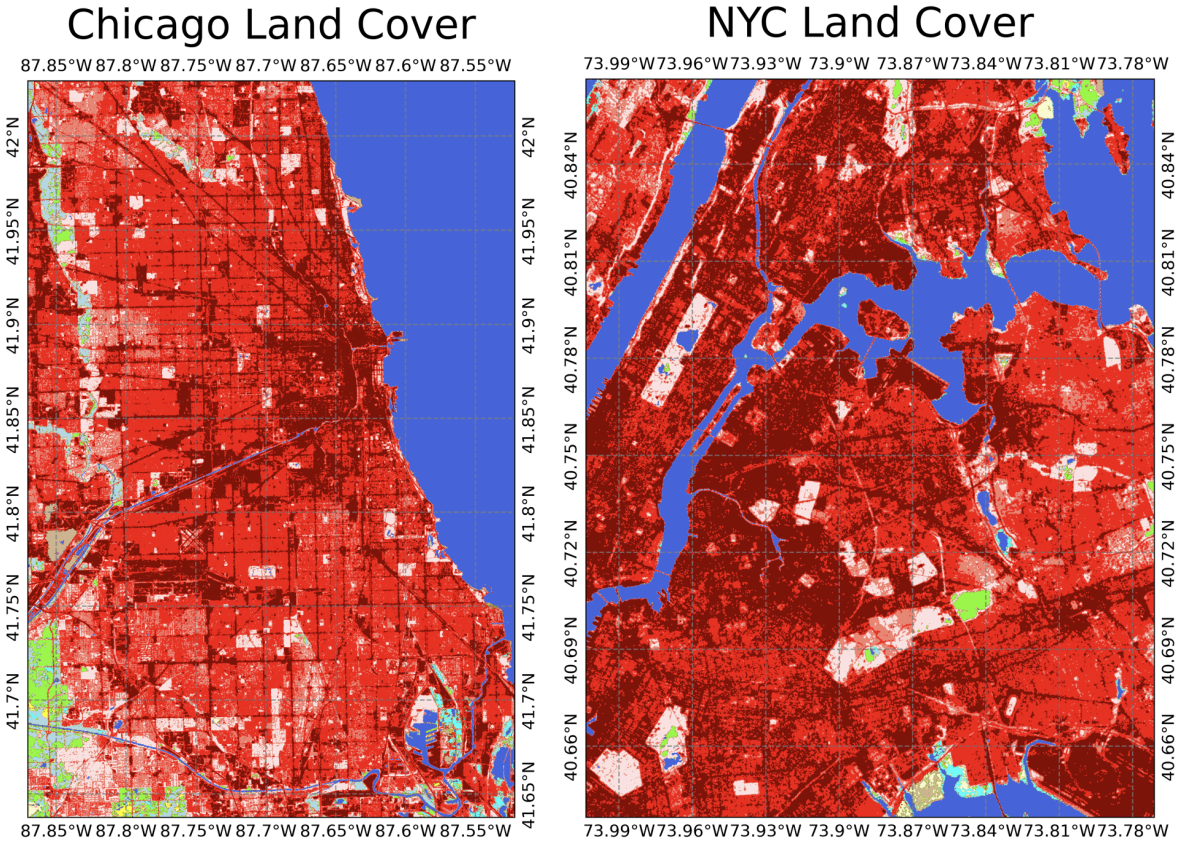


Figure 9: Land cover data of two Type 2 cities

Type 3 Cities (Low Length Scales, Low Developed Land Percentage)

Finally, this cluster of cities does not agree with our hypothesis since we expect higher length scales for areas that are less developed. These cities include Baltimore, Durham, Boulder, Albuquerque, etc.

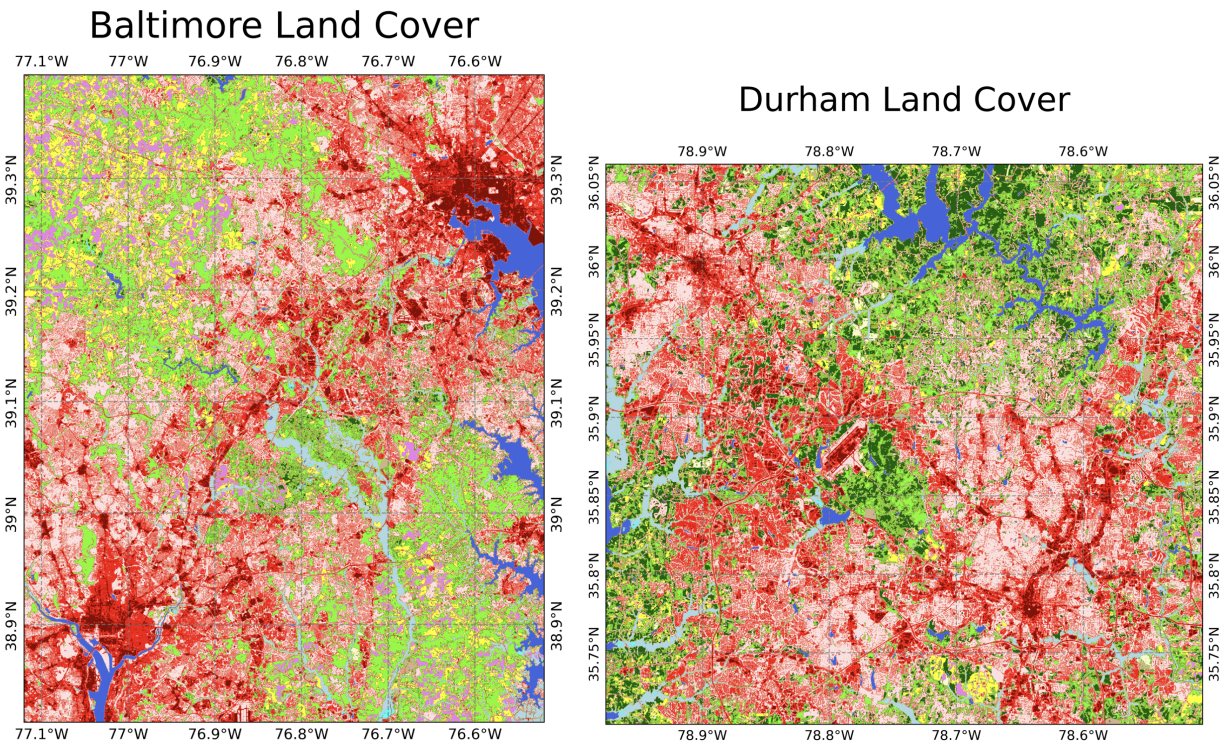


Figure 10: Land cover data of two Type 3 cities

Interestingly, Type 3 cities have their developed and vegetated areas more distributed over space. In Baltimore and Durham, we can see two clusters of developed areas, but in the areas between them, the developed and vegetated areas tend to mix more. It seemed like the distribution of the developed areas in a city also affects the optimal length scale of the Gaussian process model. This can be considered by learning a distribution of length scales over space, which as we discussed above is the non-stationary nature of environmental data. In the context of urban heat islands, the air temperature data has been proven to be non-stationary over space.

Non-stationary results

The baseline non-stationary model mentioned under the section “Non-stationary kernels” was implemented to not only learn the length scales w given the pre-defined points but also to see if the length scales learned agree with the nature of the Type 1 and Type 2 cities. Figure 11 shows the four points defined in all four corners of the Raleigh-Durham area. Table 2 shows the learned length scales which confirm that the points, top left and bottom right, located in the developed areas have virtually low length scales; and the points, top right and bottom left, located in the vegetated areas have higher length scales.

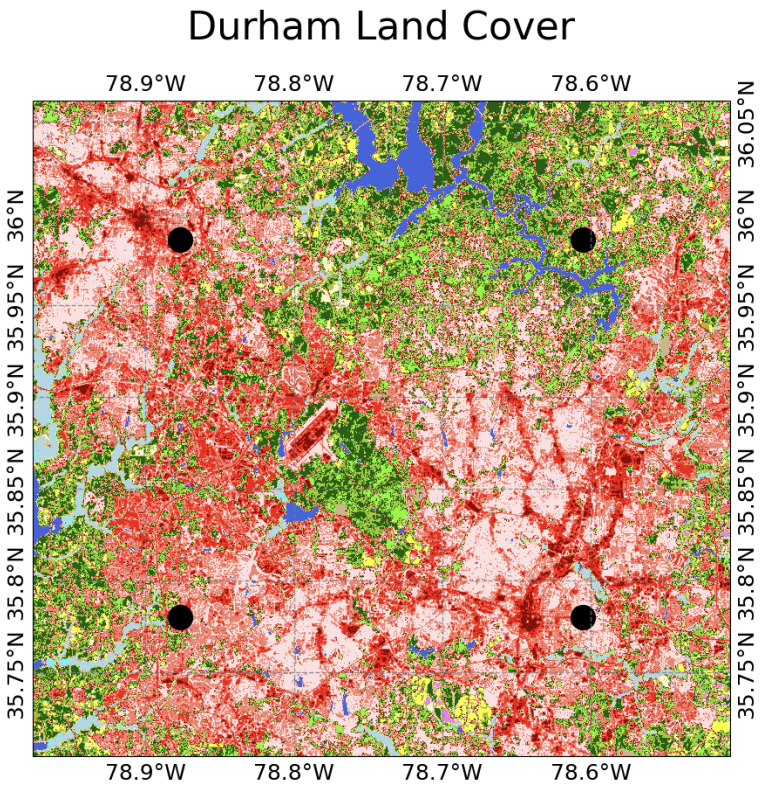


Figure 11: Manually defined points on the Durham traversal data where length scales are going to be learning

Table 2. Localized learned length scales on the coordinates specified below

Coordinates (lat, lon)	**Length Scale (km)**
(35.8 N, -78.9 W)	(20, 19)
(36.0 N, -78.9 W)	(6.4, 5.9)
(36.0 N, -78.6 W)	(21, 20)
(35.8 N, -78.6 W)	(~0, ~0)

The non-stationary Gaussian process model attained the minimum log-likelihood when these length scales were used in the non-stationary kernels. The length scale for each point is a product of the local length scale w and the non-stationary kernel. These length scales underscore the relationship between building density (or surface roughness) and wind speed and circulation.

Conclusion and Future Work

We have shown the necessity of considering different length scales across the city. The use of a single-length scale to model the urban heat island effect in a city proves limiting. Non-stationary modeling is necessary to capture the variance of length scales in a city.

Moreover, we have demonstrated the necessity of accounting for varying length scales when modeling the urban heat island effect within cities. Our results highlight the limitations of using a single, city-wide length scale in stationary Gaussian Process models, as it fails to capture the spatial heterogeneity inherent in urban environments. Different regions within a city, influenced by varying levels of development, vegetation, and geographic features, exhibit distinct spatial patterns that require localized modeling approaches. By incorporating non-stationary Gaussian Process models, we have shown that it is possible to adapt the length scales dynamically, allowing for a more accurate representation of spatial variations in temperature and improving the interpretability of the results.

Explore more non-stationary models

The next step for this project is to explore other non-stationary models like deep kernels and deep GPs. [2] We need to find a method to compare the performance of non-stationary kernels and the stationary Matern kernel used by the Gaussian process within the overarching spatial causal inference model. Once we have a better understanding of the implications of the non-stationary models of the urban heat island effect, we could go back to quantifying the total effect of existing heat-mitigating interventions on the air temperature in a city. By this time, we capture more variance of our point estimate across regions in a city and across cities. This ultimately leads to more informed policies: targeting specific areas for installing parks or applying reflective coating on buildings.

Towards optimal policy evaluation

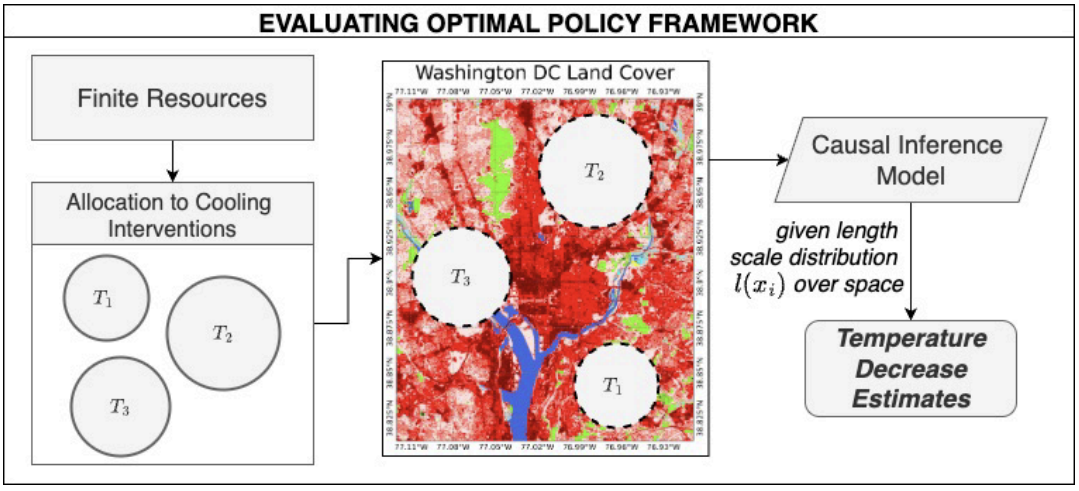


Figure 12: The optimal policy framework housing the spatial causal inference

This framework provides an evidence-based approach to optimizing the placement of heat-mitigating infrastructure, such as green spaces, reflective surfaces, or trees, under finite resource constraints. Using the spatial causal inference model, the framework estimates the temperature reduction across different spatial zones by incorporating the length scale distribution, which accounts for spatial heterogeneity in the cooling effects of interventions. This allows policymakers to prioritize intervention areas where resources will achieve the maximum temperature decrease.

For example, as shown in Washington DC, resources are allocated to intervention zones T_1, T_2, T_3, guided by the causal model's temperature decrease estimates. These treatments are a function of the extent and location to which and where they are applied, respectively. By capturing spatial variations through the learned length scales, the model identifies regions with the greatest potential for cooling impact. This enables policymakers to make targeted and efficient decisions, balancing limited resources while addressing urban heat island effects effectively.

References

[1] Z. D. Calhoun, F. Willard, C. Ge, C. Rodriguez, M. Bergin, and D. Carlson, “Estimating the effects of vegetation and increased albedo on the urban heat island effect with spatial causal inference,” *Sci. Rep.*, vol. 14, no. 1, Art. no. 1, Jan. 2024, doi: 10.1038/s41598-023-50981-w.

[2] M. M. Noack, H. Luo, and M. D. Risser, “A unifying perspective on non-stationary kernels for deeper Gaussian processes,” *APL Mach. Learn.*, vol. 2, no. 1, p. 010902, Feb. 2024, doi: 10.1063/5.0176963.

[3] L. Shand and B. Li, “Modeling Nonstationarity in Space and Time,” *Biometrics*, vol. 73, no. 3, pp. 759–768, Sep. 2017, doi: 10.1111/biom.12656.

[4] “Mapping Campaigns.” Accessed: Dec. 14, 2024. [Online]. Available: https://www.heat.gov/pages/mapping-campaigns

[5] P. Rajagopalan, K. C. Lim, and E. Jamei, “Urban heat island and wind flow characteristics of a tropical city,” *Sol. Energy*, vol. 107, pp. 159–170, Sep. 2014, doi: 10.1016/j.solener.2014.05.042.

[6] G.-A. Fuglstad, D. Simpson, F. Lindgren, and H. Rue, “Does non-stationary spatial data always require non-stationary random fields?,” *Spat. Stat.*, vol. 14, pp. 505–531, Nov. 2015, doi: 10.1016/j.spasta.2015.10.001.

[7] S. Garg, A. Singh, and F. Ramos, “Learning Non-Stationary Space-Time Models for Environmental Monitoring,” *Proc. AAAI Conf. Artif. Intell.*, vol. 26, no. 1, Art. no. 1, 2012, doi: 10.1609/aaai.v26i1.8166.

Appendices

Appendix A. All the code in the experiments described above is found on GitHub https://github.com/edrian-liao/uhi-deep-learning.

Appendix B. The land cover images of all 20 cities are found here: figures.

Appendix C. The pipeline script used to process normalized coordinates and air temperature from the shapefiles and to train an ExactGP on < 20k points, randomly sampled from the traversal data, using a GPU

```
import argparse
import logging
import os
import geopandas as gpd
import pandas as pd
import numpy as np
import torch
import gpytorch
from sklearn.metrics import r2_score
from sklearn.model_selection import KFold

from src.models import ExactGPModel


def main(args):
    # Initialize a logger
    logging.basicConfig(
        filename=os.path.join(args.output_dir, f"logs/{args.city}_output.log"),
        filemode="w",
        level=args.log_level,
        format="%(asctime)s - %(levelname)s - %(message)s",
    )
    logging.info(args)

    # Load and preprocess GeoDataFrame
    shapefile_path = f"{os.path.join(args.data_dir, args.city)}/pm_trav.shp"
    gdf = gpd.read_file(shapefile_path)

    # Convert CRS
    gdf = gdf.to_crs(epsg=3857)
    gdf['x'] = gdf.geometry.x
    gdf['y'] = gdf.geometry.y

    # Identify the temperature column
    temp_column = get_temperature_column(gdf)

    # Prepare data
    X = gdf[['x', 'y']].to_numpy()
    y = gdf[temp_column].to_numpy()
```

```
    logging.info(f"Loaded {len(X)} data points")

    # Sample random points
    if len(X) > args.num_random_points:
        random_indices = np.random.choice(len(X), size=args.num_random_points,
replace=False)
        X = X[random_indices]
        y = y[random_indices]

    # # Standardize the data
    x_shift = X.min(axis=0)
    x_scale = X.max(axis=0) - x_shift
    X = (X - x_shift) / (x_scale + 1e-8)

    y_mean = y.mean()
    y_std = y.std()
    y = (y - y_mean) / y_std

    # Convert to PyTorch tensors
    X_tensor = torch.tensor(X, dtype=torch.float32)
    y_tensor = torch.tensor(y, dtype=torch.float32)

    # Initialize GroupKFold
    kfold = KFold(n_splits=args.k_folds, shuffle=True, random_state=42)
    fold_r2_scores = []
    learned_length_scales = []

    for fold, (train_idx, val_idx) in enumerate(kfold.split(X_tensor)):
        # Split data into training and validation sets
        x_train, x_val = X_tensor[train_idx], X_tensor[val_idx]
        y_train, y_val = y_tensor[train_idx], y_tensor[val_idx]

        # Set up the Gaussian Process model
        fixed_noise = torch.full_like(y_train, args.fixed_noise)
        # likelihood = gpytorch.likelihoods.GaussianLikelihood()
        likelihood =
gpytorch.likelihoods.FixedNoiseGaussianLikelihood(noise=fixed_noise)
        model = ExactGPModel(x_train, y_train, likelihood)

        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model = model.to(device)
```

```
        likelihood = likelihood.to(device)
        x_train, y_train = x_train.to(device), y_train.to(device)
        x_val, y_val = x_val.to(device), y_val.to(device)

        # Train the model
        model.train()
        likelihood.train()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
        mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

        min_loss = float("inf")
        learned_length_scale = 0
        best_model_path = os.path.join(args.output_dir,
f"models/{args.city}/best_model_fold_{fold + 1}.pth")
        for epoch in range(70):
            optimizer.zero_grad()
            output = model(x_train)
            loss = -mll(output, y_train)

            # Save the model if the current loss is the lowest
            if loss.item() < min_loss:
                min_loss = loss.item()
                torch.save(model.state_dict(), best_model_path)
                learned_length_scale =
model.covar_module.base_kernel.lengthscale.item()
                logging.info(f"New best model saved for fold {fold + 1}")

            loss.backward()
            optimizer.step()

            logging.info(f"Fold {fold + 1}, Epoch {epoch + 1}/70 - Loss:
{loss.item():.3f}")

        learned_length_scales.append(learned_length_scale)
        logging.info(f"Finished training for fold {fold + 1} with best loss:
{min_loss:.3f} and learned length scale: {learned_length_scale:.3f}")

        # Load the best model for validation
        model.load_state_dict(torch.load(best_model_path))
        model.eval()
        likelihood.eval()
```

```
        # Evaluate on validation set
        with torch.no_grad():
            val_output = model(x_val)
            y_val_pred = val_output.mean.cpu().numpy()
            y_val_true = y_val.cpu().numpy()

            # Calculate R^2 metric
            r2 = r2_score(y_val_true, y_val_pred)
            fold_r2_scores.append(r2)

        logging.info(f"Validation R^2 for fold {fold + 1}: {r2:.3f}")

    # Compute average \( R^2 \) across all folds
    avg_r2 = np.mean(fold_r2_scores)
    avg_length_scale = np.mean(learned_length_scales)
    avg_length_scale_scaled = avg_length_scale * x_scale
    logging.info(f"Average validation R^2 across all folds: {avg_r2:.3f}")
    logging.info(f"Average learned length scale across latitude:
{avg_length_scale_scaled[0]:.3f}")
    logging.info(f"Average learned length scale across longitude:
{avg_length_scale_scaled[1]:.3f}")

    # Update the CSV file
    results_csv = os.path.join(args.output_dir, "results.csv")
    results = {
        "city": args.city,
        "avg_r2": avg_r2,
        "avg_length_scale_lat": avg_length_scale_scaled[0],
        "avg_length_scale_lon": avg_length_scale_scaled[1],
    }

    # Check if the file exists
    if os.path.exists(results_csv):
        results_df = pd.read_csv(results_csv)
        if args.city in results_df["city"].values:
            results_df.loc[results_df["city"] == args.city, ["avg_r2",
"avg_length_scale_lat", "avg_length_scale_lon"]] = [
                avg_r2,
                avg_length_scale_scaled[0],
                avg_length_scale_scaled[1],
            ]
        else:
```

```
            results_df = pd.concat([results_df, pd.DataFrame([results])],
ignore_index=True)
    else:
        results_df = pd.DataFrame([results])

    # Save the updated DataFrame
    results_df.to_csv(results_csv, index=False)

    return


def get_temperature_column(gdf):
    # List of potential column names for temperature
    possible_columns = ["temp", "temp_f", "t_f", "T", "T_F"]  # Update this list as
needed

    # Check if any of these columns exist in the GeoDataFrame
    for col in possible_columns:
        if col in gdf.columns:
            return col

    # Raise an error if none are found
    raise ValueError("No temperature column found in the GeoDataFrame.")


if __name__ == "__main__":
    # Create the parser
    parser = argparse.ArgumentParser(description="Train an Exact Gaussian process on a
city-level dataset")
    parser.add_argument(
        "data_dir",
        type=str,
        default="data/shapefiles",
        help="The directory where the data is stored"
    )
    parser.add_argument(
        "city",
        type=str,
        help="The city where data is being collected from"
    )
    parser.add_argument(
        "fixed_noise",
```

```
        type=float,
        help="The fixed assumption of noise from the observations during training"
    )
    parser.add_argument(
        "--num_random_points",
        type=int,
        default=20000,
        help="The number of random points to sample from the dataset"
    )
    parser.add_argument(
        "--k_folds",
        type=int,
        default=5,
        help="The number of folds to use for cross-validation"
    )
    parser.add_argument("--output_dir", type=str, help="Path to the output directory")
    parser.add_argument("--log_level", type=str, default="INFO", help="Logging level")

    args = parser.parse_args()
    main(args)
```

Appendix D. The output log file from running the pipeline script on all 20 cities: logs

Appendix E. The code used to define the non-stationary Gaussian process model via "parametrized signal"

```
class NonStationaryKernel(Kernel):
    """
    A non-stationary kernel that applies a parametric signal variance
    on top of a base stationary kernel, with multiple RBF widths.
    """
    def __init__(self, base_kernel=None, num_rbf_centers=10, input_dim=2):
        super(NonStationaryKernel, self).__init__()
        self.base_kernel = base_kernel

        # Define fixed RBF centers
        self.rbf_centers = torch.tensor(
            [[0.2, 0.2], [0.2, 0.8], [0.8, 0.8], [0.8, 0.2]],
            requires_grad=False,  # Make these non-trainable
        )



        # Define coefficients (c_k in g(x))
        self.register_parameter(
            name="coefficients",
            parameter=torch.nn.Parameter(torch.randn(num_rbf_centers))
        )
        # Initialize RBF widths constrained to (0, ∞)
        self.register_parameter(
            name="raw_rbf_widths",
            parameter=torch.nn.Parameter(torch.ones(num_rbf_centers))
        )

        # Register transforms for constraints
        self.register_constraint("raw_rbf_widths", gpytorch.constraints.Positive())

@property
    def rbf_widths(self):
        # Apply softplus transformation to ensure widths are positive
        return torch.nn.functional.softplus(self.raw_rbf_widths)

    def forward(self, x1, x2, **params):
        # Compute g(x) for x1 and x2
        g_x1 = self._compute_signal_variance(x1)
        g_x2 = self._compute_signal_variance(x2)
```

```
        # Compute the base kernel k_stat(x1, x2)
        base_k = self.base_kernel(x1, x2, **params)

        # Return g(x1) * g(x2) * k_stat(x1, x2)
        return g_x1.unsqueeze(-1) * g_x2.unsqueeze(-2) * base_k

    def _compute_signal_variance(self, x):
        """
        Compute g(x) = sum(c_k * RBF(x_k, x)) with per-center RBF widths.
        """
        # Pairwise distances between x and RBF centers
        distances = torch.cdist(x, self.rbf_centers) ** 2

        # Compute RBF values with per-center widths
        rbf_values = torch.exp(-distances / self.rbf_widths.unsqueeze(0))

        # g(x) = sum(c_k * RBF(x_k, x))
        g_x = torch.matmul(rbf_values, self.coefficients)
        return g_x
__
class NonStationaryGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(NonStationaryGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = NonStationaryKernel(
            base_kernel=MaternKernel(), num_rbf_centers=4, input_dim=train_x.shape[-1]
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```