| Problem Chosen | 2024 ShuWei Cup Summary Sheet | Team Control Number |
|---|---|---|
| B | | 2024093029322 |

# Spatial Interpolation and Collaborative Variable Analysis

Abstract

This study aims to model and predict spatial variables using dynamic graphs and **hybrid Graph Neural Networks (GNN)**, addressing the computational complexity issues of traditional spatial statistical methods in high-dimensional and large-scale data processing. The research employs a combined model of **Graph Convolutional Networks (GCN)** and **Graph Attention Networks (GAT)**, integrating dynamic **K-nearest neighbor graphs** (with K values of 2, 4, 6, 8, and 10) to analyze the impact of different K values on model performance. Data preprocessing for the target variable (F1_target) and its four collaborative variables includes data loading, format checking, anomaly handling, and missing value checks to ensure data quality. For problems one to four, analyses were conducted on the variation patterns of the target variable, the correlation of collaborative variables, the resampling and interpolation estimation of spatial variables, and dynamic KNN interpolation under conditions of insufficient samples. Experimental results indicate that when K=10, the model efficiently integrates information from the 10 nearest points, significantly improving the prediction accuracy of the target variable; the dynamic KNN interpolation method outperforms the **Inverse Distance Weighting (IDW)** method at various sampling ratios, particularly excelling at low sampling ratios. The Pearson correlation coefficient between collaborative variable 1 (F1_collaborative_variable1) and the target variable is 0.76, while for collaborative variable 4 (F1_collaborative_variable4) it is 0.32, demonstrating their critical roles in prediction. Compared to traditional Kriging methods, GNN shows superior computational efficiency and predictive performance. Ultimately, the dynamic KNN interpolation method further optimizes the model's spatial estimation capabilities, showcasing the immense potential of GNN in handling complex spatial data and achieving efficient predictions.

**Keywords:** Spatial Statistics, Graph Neural Networks, Dynamic Graphs, Graph Convolutional Networks, Graph Attention Networks, Dynamic K-Nearest Neighbors, Spatial Interpolation, Collaborative Estimation, Predictive Model

# Content

# 1. Introduction

## 1.1 Background

Traditional statistical methods typically assume that sample points are independent, while spatial statistics recognize that there are dependencies and spatial trends among sample points of spatial variables. This allows known data to be used to infer values at unsampled locations. In practical applications, the same spatial variable can be measured using different methods, which may result in numerical differences due to varying measurement principles, yet still maintain significant spatial correlation. Additionally, there may also be correlations between spatial variables with different physical meanings, providing opportunities for joint estimation.

In many engineering studies, certain spatial variables are expensive to measure and difficult to obtain, resulting in limited samples; while other variables are low-cost and easy to sample on a large scale. Researchers often rely on other spatial variables that are highly correlated with the target variable to fill in the gaps in the sample through co-estimation. Although the Co-Kriging method is theoretically well-recognized, practical applications face challenges due to the complexity of calculating cross-covariance or variogram functions.

With the development of artificial intelligence and machine learning, these technologies have achieved success in multiple industries. This competition provides two datasets containing measurements of four spatial attributes from the same region, with the target variable being costly to measure. Participants are required to select appropriate co-estimation methods to process the given spatial attribute data, explore the variation patterns of the target variable, analyze the correlations among the covariates, and effectively perform spatial estimation in cases of insufficient samples.

## 1.2 Restatement of the Problem

Spatial statistics posits that there are dependencies between sampling points of spatial variables, and different measurement methods may lead to variations in the measured values of the same physical quantity, yet they still exhibit significant spatial correlation. Based on this context, this study aims to address the following four questions:

Question 1: Using the data in Attachment 1, study the variation pattern of the target variable (F1_target variable). Participants are required to randomly and uniformly resample the target variable and estimate the spatial variable values at unsampled locations, with the results presented in the form of a contour map. Additionally, the relationship between sample size and estimation error should be explored.

Question 2: Analyze the correlation between the target variable and the covariates, and select two covariates that are most correlated with the target variable for subsequent estimation.

Question 3: Using the data in Attachment 1 and the results from Question 2, select one or two covariates to study the variation pattern of the target variable (F1_target variable). Participants are required to randomly and uniformly resample the target and covariates, estimate the spatial variable values at unsampled locations, and present the results using contour maps. Additionally, the relationship between sample size and estimation error should be explored, and the effectiveness of at least two estimation methods should be compared.

Question 4: Due to insufficient sample data for the target variable (F2_target variable) in Appendix 2, it is necessary to select the best estimation method from Question 3 to infer the trend of the target variable and present the results in the form of a contour map.

# 2. Problem analysis

## 2.1 Analysis of question one

For problem one, to analyze the variation patterns of the F1_target variable and estimate values at unsampled locations using random uniform resampling, we will employ Graph Neural Networks (GNN) comprising Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT). GCN captures spatial dependencies and correlations between adjacent data points, extracting local features by aggregating neighbor information, thereby addressing traditional method limitations. The GAT module incorporates an attention mechanism that dynamically adjusts weights based on the relationships among data points, effectively handling spatial heterogeneity and enhancing estimation accuracy by identifying significant covariates. By training the GNN model on resampled data, we can accurately estimate F1_target values at unsampled locations and analyze the relationship between sample size and estimation error, ultimately presenting results in a contour map.

## 2.2 Analysis of question two

For problem two, we first analyze the relationship between the target variable (F1_target variable) and all the covariates using a correlation coefficient matrix to identify the two covariates most related to the target variable. After selecting these two covariates, we will construct a model that combines Dynamic Graph Neural Networks (DGN), Graph Convolutional Networks (GCN), and Graph Attention Networks (GAT) to utilize these two covariates for estimating the target variable. Through this approach, we can effectively capture the structural characteristics of spatial data and dynamically adjust the influence of different covariates on the estimation of the target variable, thereby improving the accuracy of the estimates. Finally, we will evaluate the model's performance to validate the effectiveness of the chosen method.

## 2.3 Analysis of question three

For question three, we will select two covariates that are most strongly correlated with the target variable based on the analysis results from question two, to study the variation patterns of the target variable (F1_target variable). Specifically, we will randomly and uniformly resample both the target variable and the covariates, and use dynamic KNN and IDM interpolation methods to estimate the spatial variable values at unsampled locations. By comparing the relevant error metrics of the two interpolation methods, we will be able to evaluate the performance of different methods in spatial estimation, further analyze the impact of sample size on estimation error, and ultimately present the estimation results in the form of contour maps. This process will help deepen our understanding of the influence of covariates on the estimation of the target variable and provide a basis for subsequent spatial variable analysis.

## 2.4 Analysis of question four

For question four, due to the insufficient sample data for the target variable (F2_target variable) in Appendix II, we will employ the dynamic KNN (dynamic k-nearest neighbors) method to address the issue of missing values and generate trend maps. This method effectively utilizes known spatial variable data by considering the spatial correlation of neighboring samples to infer the target variable values at unsampled locations. We will randomly select a certain number of sample points and dynamically adjust the number of neighbors in the KNN algorithm based on the spatial distribution and interrelationships of these points to enhance the accuracy of the estimates. Ultimately, we will present the estimated results in the form of contour maps to visually illustrate the spatial variation trends of the target variable. This process will help obtain reasonable estimates of the target variable even in data-scarce situations and provide a reference for subsequent research.

# 3. Symbol and Assumptions

**1. Spatial Stationarity:** The target variable within the study area exhibits consistency in mean and variance, with measurements taken at different locations fluctuating around a stable mean, and its distribution characteristics remaining unchanged during the spatial interpolation process.

**2. Representative sampling:** The selected sampling points ensure sufficient spatial coverage, and the sampling method is unbiased, ensuring that the collected data accurately reflects the characteristics of the study area.

**3. Spatial correlation:** In spatial data, measurements at adjacent locations exhibit a high degree of similarity.

**4. Accuracy of Measurement:** The data for the target variable and its covariates possess high accuracy, and there is no significant noise in the data. The quality of the data used will not negatively impact the final results.

## 4. Explanation of Symbols

| Symbols | Explanation | Unit |
|---|---|---|
| $\omega_i$ | Point weight | Scalar |
| $d_i$ | Distance to the target point | Scalar |
| $Z(x)$ | Estimated value of the target point | Scalar |
| $\|\|W\|\|_1$ | Regularization term | Scalar |
| $S_{ij}$ | Weight matrix | Scalar |
| $W^*$ | Optimal Correlation Matrix | Scalar |
| $D_{ii}$ | Importance of Sample Measurement | Scalar |

## 5. Establishment and Solution of the Model

### 5.1 Data Preprocessing

Data preprocessing aims to ensure that the data is correctly loaded, processed, and validated before analysis. The specific steps are as follows:

**1. Data loading:** Read the target variable from the file F1_target_variable.txt and the collaborative variables from F1_collaborative_variable1-4.txt in the Attachment 1 folder. Each file contains 266x266 grid data. Use Python to convert the text data into numpy arrays.

**2. Format Check:** Skip the first 6 lines of metadata in each file and parse the subsequent data as numerical values. Verify that each file contains 70,756 data points (266x266).

**3. Missing Value Check:** Use numpy.isnan() to check for missing values in the matrix and output a warning if any are found.

**4. Data Transformation:** Valid data is reconstructed into a 266x266 numpy matrix, while data that does not meet the criteria is treated as an empty array.

After the above steps, all variable data has been successfully converted into a 266x266 matrix, with both the data format and content meeting expectations, and there are no missing values.

### 5.2 Establishment and Solution of Model for Problem One

**Load data and construct the spatial coordinate system:**

Data loading: Read the preprocessed F1_target_variable data file and load it as a matrix of size 266x266, accurately representing the actual distribution of spatial variables.

Generate grid coordinates: The row distance and column distance generate corresponding X and Y coordinates for each data point, constructing data for GNN model learning.

**Exploratory Data Analysis:**

Descriptive statistical analysis: Examine the basic characteristics of the data and confirm the presence of any outliers.

Spatial Visualization: By creating heatmaps, you can quickly view the spatial distribution of data, identify areas of variable concentration, trends, and potential outliers. When drawing heatmaps, set appropriate themes, titles, and color bar labels to facilitate understanding of the spatial trends in the data.

**Observation results:**

The spatial distribution of the target variable F1_target_variable shows significant variation within the study area. In the heatmap, black areas represent high-value regions, while yellow areas indicate low-value regions. The mean and standard deviation of the data suggest that values are primarily concentrated around the mean, with very high or very low values being rare, which is also reflected in the spatial distribution map.

**Set the sampling ratio and perform random sampling:**

Define sampling ratio: Set different sampling quantities (such as 100, 200, 1600, 3600, 6400) to control the number of randomly selected known data points, in order to evaluate the performance of the GNN model under different data densities.

Randomly select sampling points: For each sampling ratio, randomly select a specified number of sampling points from the target variable dataset, and record the coordinates of these sampling points along with their corresponding data values.

**Prepare model input:**

Organize the X coordinates, Y coordinates, and corresponding data values of the sampled points into the input format for the model algorithm. Using the aforementioned model input, apply appropriate spatial modeling methods to generate spatial predictions for the entire area, enabling data estimation for unsampled regions.

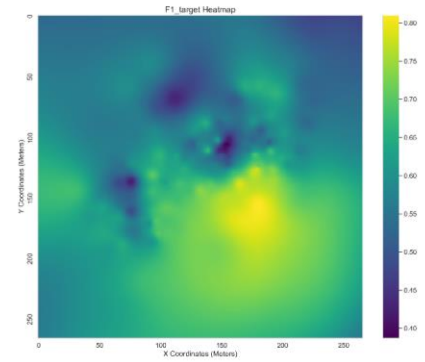

Fig 1 Heatmap of Target Variables

Data points of the target variable were randomly selected based on different sampling ratios. By conducting random sampling at various sampling ratios, we can provide known points of different densities for subsequent model analysis, in order to explore the impact of sampling density on the accuracy of spatial models.

The GNN model established in this task aims to analyze, model, and predict spatial variable data. The main objective is to read the given spatial data file, perform preprocessing and visualization, and then use a hybrid Graph Neural Network (GNN) model to train and predict the data, as well as evaluate the model's performance.

The GNN model established in this task consists of three layers of GCN, each followed by batch normalization, LeakyReLU activation function, and dropout; two layers of GAT, each followed by batch normalization, an activation function, and dropout; and employs residual connections to enhance the model's feature representation capability. It uses a GATConv layer with an output dimension of 1 to predict the target variable.

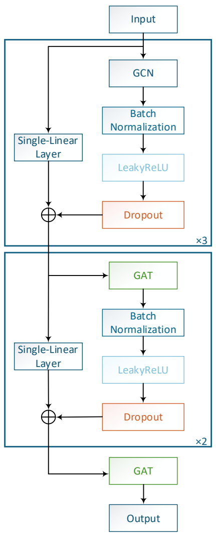

Fig 2 Flowchart of the Mixed GNN

GCN(Graph Convolutional Network):

Based on the convolution operation extended to graph structures, the representation of each node is updated by aggregating information from neighboring nodes. The formula is expressed as:

$$H^{(l+1)} = \sigma(\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2} H^{(l)} W^{(l)}) \tag{1}$$

Among them, $\hat{A} = A + I$ is the adjacency matrix with self-loops, $\hat{D}$ is the degree matrix, $W^{(l)}$ is the learnable weight matrix, and σ is the activation function.

GAT(Graph Attention Network):

Introducing the attention mechanism assigns different weights to neighboring nodes, automatically learning the importance of the relationships between nodes by calculating attention coefficients. The formula is expressed as:

$$\alpha_{ij} = \frac{exp\left(LeakyReLU\left(a^T\left[Wh_i // Wh_j\right]\right)\right)}{\sum_{k\in\mathcal{N}(i)} exp\left(LeakyReLU\left(a^T\left[Wh_i // Wh_k\right]\right)\right)} \tag{2}$$

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij} Wh_j\right)$$

In this context, a is the learnable attention weight, W is the linear transformation matrix, and N(i) is the set of neighbors of node i.

A graph is an important data structure in mathematics and computer science, used to represent objects and the relationships between them. Graph construction and k-hop are two core concepts in graph theory and its applications. A graph consists of a set of vertices (nodes) and edges (links) that connect the vertices. Formally, a graph G can be represented as G=(V,E), where V is the set of vertices and E is the set of edges, with each edge connecting two vertices.

In the diagram, k-step refers to all the vertices that can be reached from a starting vertex by traversing exactly k edges. In other words, the vertices at k steps are those that are k distance away from the starting vertex. In GNNs, a node's features are obtained by aggregating information from its multi-hop neighbors, and the k-step determines the range of information propagation; by using the neighbor information from k steps, we learn the low-dimensional representation of the nodes. The diagram below illustrates the graph structures constructed with 4-step, 6-step, and 8-step connections:

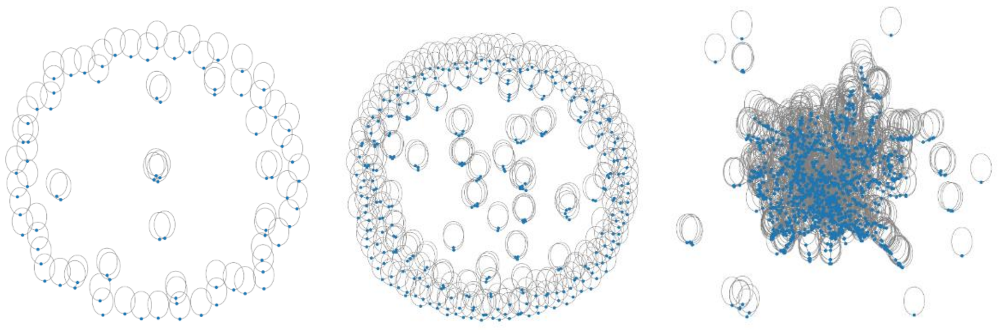
Fig 3: Structure of 4-step, 6-step, and 8-step jumps

The jump step diagram shows that as the number of jump steps increases, the complexity of the relationships between points also increases. This allows the model to learn different features, which is beneficial for understanding the characteristics of sampled data and lays the foundation for inferring un-sampled data.

By reading and processing spatial data, a hybrid graph neural network model was used for modeling and prediction. The experimental results indicate that:

**The impact of data volume on model performance:** As the amount of training data increases, the model's prediction error significantly decreases, indicating that sufficient data is crucial for model training.

**The effectiveness of the model:** A model that combines GCN and GAT can effectively capture the structure and features of spatial data, achieving accurate predictions of spatial variables.

**The importance of graph structure:** An appropriate neighbor distance (k value) is crucial for constructing a reasonable graph structure, which affects the learning effectiveness of the model.

Save the model results as a 266x266 matrix, representing the estimated values across the entire spatial range for subsequent error analysis and visualization.

In spatial model analysis, error assessment is a key step in determining the accuracy and reliability of model results. Through error assessment, the deviation between model results and true values can be analyzed, aiding in the selection of the optimal sampling ratio and modeling method. This step's error assessment includes three main error metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE).

**Mean Squared Error(MSE):**

MSE is an indicator for assessing the overall deviation of model results, reflecting the average of the squared differences between predicted values and actual values. The smaller the MSE, the closer the predicted values are to the actual values, indicating higher model accuracy. The formula is expressed as:

$$\mathrm{MSE}=\frac{1}{N}\sum_{i=1}^{N}(z_i-\hat{z}_i)^2 \tag{3}$$

In this, $z_i$ is the true value at the i position, $\hat{z}_i$ is the model's predicted value at the i position, and N is the number of unsampled points. MSE is the square of the deviation, making it more sensitive to points with larger deviations.

**Mean Absolute Error( MAE):**

MAE is the average of the absolute differences between the predicted values of the model and the actual values, reflecting the average deviation of the model. The formula is expressed as:

$$\mathrm{MAE}=\frac{1}{N}\sum_{i=1}^{N}|z_i-\hat{z}_i| \tag{4}$$

In this, $|z_i-\hat{z}_i|$ represents the absolute error between the i-th predicted value and the true value. Unlike MSE, MAE does not amplify the impact of larger deviations, making it suitable for analyzing average deviation situations.

**Root Mean Squared Error (RMSE):**

Root mean square error is a commonly used metric for measuring the difference between model predictions or estimates and observed values. The formula is expressed as:

$$RMSE=\sqrt{\frac{\sum_{i=1}^{N}(z_i-\hat{z}_i)^2}{N}} \tag{5}$$

We resample the data using various sampling sizes. For each sampling ratio, we calculate MSE, MAE, and RMSE to analyze the impact of the sampling ratio on model accuracy.

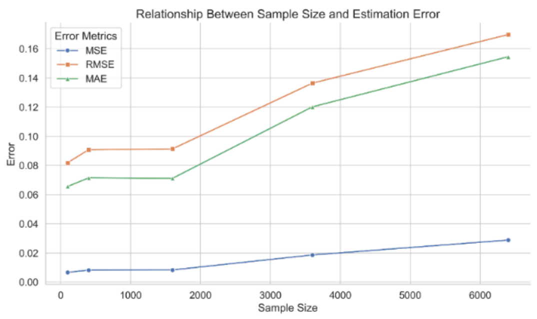

Fig 4 Relationship between Sample Size and Evaluation Error

The analysis results and charts indicate that, at lower sampling quantities, the exponential model generally demonstrates better performance, with relatively low MSE, MAE, and RMSE, showing strong stability. However, as the sampling quantity increases, there are larger errors, particularly with RMSE reaching higher levels, indicating that errors rise with the increase in sampling quantity.

Compared to traditional Kriging methods, GNNs have significant advantages. Kriging relies on strict assumptions of spatial autocorrelation and prior models, which impose high demands on the spatial distribution of data. Additionally, it consumes considerable computational resources and time, especially when dealing with high-dimensional and large-scale data, leading to a sharp increase in computational complexity. In contrast, GNNs can automatically capture nonlinear relationships and complex spatial patterns through an end-to-end learning process, reducing dependence on prior knowledge. Furthermore, GNNs offer greater flexibility, adapting to different types of data and tasks, whether in regression or classification problems, providing robust modeling capabilities.

Through this analysis, we successfully achieved the evaluation of the model and sampling error for the spatial variable F1_target_variable. First, we loaded the data and conducted an outlier check, providing a clear data foundation for the subsequent model. Then, by defining different sampling sizes and applying a mixed GNN model, we performed spatial learning for unsampled locations. Error assessments were conducted for each sampling size, and contour maps were created to display the model results. Finally, we calculated and compared the Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE), generating a graph showing the relationship between error and sampling size.

## 5.3 Establishment and Solution of Model for Problem Two

The target variable (F1_target_variable) and four collaborative variables (F1_collaborative_variable1 to F1_collaborative_variable4) were read from Attachment 1. To facilitate subsequent correlation analysis, we flattened the two-dimensional matrix data of each variable into one-dimensional arrays. By using the .flatten() method, all data points were laid out in the same dimension, ensuring that operations and comparisons could be made on a uniform basis. The flattened data contains measurement values from all locations, simplifying the subsequent correlation calculation process.

During the data processing, we conducted a missing value check for each variable. The results indicated that there are no missing values for the target variable and the four collaborative variables (the number of missing values is 0), which demonstrates that the dataset has good integrity.

The goal of this question is to calculate the two-dimensional correlation coefficient between each collaborative variable and the target variable, using the Pearson correlation coefficient to select the two collaborative variables that are most closely related to the target variable from four collaborative variables.

**Pearson Correlation Coefficient:**

The Pearson correlation coefficient, also known as the Pearson product-moment correlation coefficient, is a statistic used to measure the strength and direction of the linear relationship between two continuous variables. Its value ranges from -1 to +1:

+1: Indicates a perfect positive linear correlation, meaning that when one variable increases, the other variable also increases in a perfectly consistent proportion.

-1: Indicates a perfect negative linear correlation, meaning that when one variable increases, the other variable decreases in a perfectly consistent proportion.

0: Indicates no linear correlation, meaning there is no linear relationship between the two variables.

The formula for calculating the Pearson correlation coefficient r is as follows:

$$r=\frac{\sum_{i=1}^{n}(X_i-\bar{X})(Y_i-\bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i-\bar{X})^2}\cdot\sqrt{\sum_{i=1}^{n}(Y_i-\bar{Y})^2}} \tag{6}$$

Among them: $X_i$ and $Y_i$ represent the i-th data point of X and Y; $\bar{X}$ and $\bar{Y}$ are the means of X and Y, respectively.

By calculating the Pearson correlation coefficient between each collaborative variable and the target variable, we can generate a correlation matrix to more intuitively observe the strength of the relationships between the variables. The heatmap of the correlation matrix is as follows:

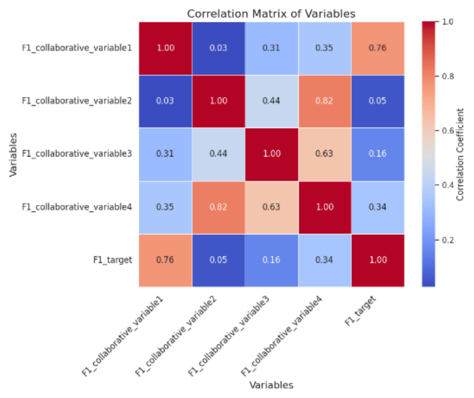

Fig 5 Heatmap of the correlation between collaborative variables and target variables

The heatmap further illustrates the correlation between the target variable and four covariates. The results show that covariate 1 (located in the upper right corner) has the highest correlation with the target variable, while covariate 2 exhibits a weaker correlation. The heatmap also reveals the interrelationships among the covariates, with a correlation coefficient of up to 0.82 between covariate 2 and covariate 4, indicating a significant correlation between the two.

Specifically, the Pearson correlation coefficient between Collaborative Variable 1 (F1_Collaborative_variable1) and the target variable is 0.76, indicating a strong positive correlation between Collaborative Variable 1 and the target variable, which is significant in both linear and monotonic relationships. Collaborative Variable 4 (F1_Collaborative_variable14) shows a moderate degree of correlation with the target variable, suggesting it has some impact on the target variable. In contrast, Collaborative Variable 2 and Collaborative Variable 3 exhibit almost no correlation with the target variable, indicating their minimal influence; the correlation coefficients for Collaborative Variable 2 and Collaborative Variable 3 are 0.02 and 0.13, respectively, demonstrating their limited effect on the target variable.

Through the visual analysis of the heatmap, not only has the strength of the relationship between each collaborative variable and the target variable been clarified, but the interconnections among the collaborative variables have also been revealed, providing important evidence for subsequent model construction and variable selection.

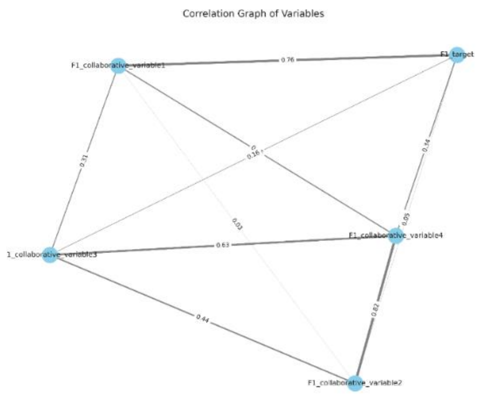

Fig 6 Correlation diagram of collaborative variables and target variables.

The results of the correlation graph and the heatmap overlap significantly, indicating that the target variable has a strong relationship with covariate 1 and covariate 4.

The output results from both the heatmap and the correlation graph show that covariate 1 and covariate 4 are the two covariates with the highest correlation to the target variable.

Using covariate 1 and covariate 4 as features, and the target variable as the learning objective in a hybrid Graph Neural Network (GNN), combined with a Dynamic K-Nearest Neighbors Graph ((D_KNN) Graph), we study the relationship between covariate 1, covariate 4, and the target variable.

The K-Nearest Neighbors Graph (KNN Graph) is a graph structure constructed based on the distance relationships between data points. In a KNN Graph, each node represents a data point, and edges connect each node to its K nearest neighboring data points. The specific steps are as follows:

1.Calculate the distance matrix: For each data point in the dataset, calculate its distance to all other data points (usually using Euclidean distance).

2.Determine the nearest neighbors: For each data point, select the K data points with the closest distances.

3.Construct edges: Add edges to the graph, connecting each node to its K nearest neighboring nodes.

A Dynamic K-Nearest Neighbors Graph ((D_KNN) Graph) refers to a KNN Graph whose structure can change over time or under certain conditions. At different time points or under different conditions, the K value (i.e., the number of nearest neighbors each node is connected ) can vary, resulting in changes in the topology of the graph.

In this problem, the Dynamic K-Nearest Neighbors Graph ((D_KNN) Graph) is constructed using different K values (2, 4, 6, 8, 10) to explore the impact of these K values on the graph structure and subsequent model performance.

Due to the complexity of the complete structure graph, which contains 266×266 nodes, we randomly sample 10,000 nodes for presentation. The following figures show the structure graphs of the (D_KNN) Graph with different K values based on this random sampling of 10,000 nodes:

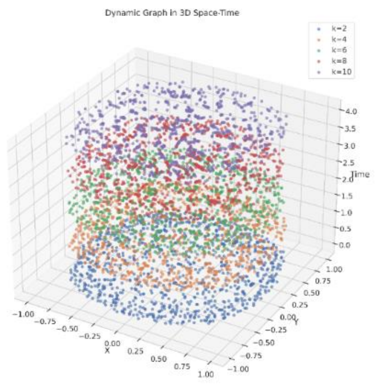

Fig 7 Dynamic Neighbor Graphs with Different K Values

Advantages of Dynamic K-Nearest Neighbors Graph:

1.Flexibility: The K value can be adjusted according to different task requirements or data characteristics.

2.Performance optimization: By adjusting the K value, one can balance the density of the graph and the computational complexity of the model.

3.Exploratory analysis: By observing the changes in the graph structure under different K values, one can gain insights into the intrinsic relationships within the data.

The hybrid GNN neural network constructed in this problem is the same as that in Question 1, which is a combination of GAT and GCN.

The model's prediction results are shown in the following figure:

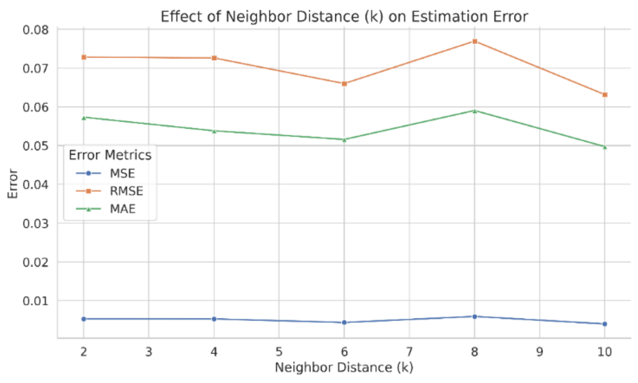

Fig 8 Different K Values and Error

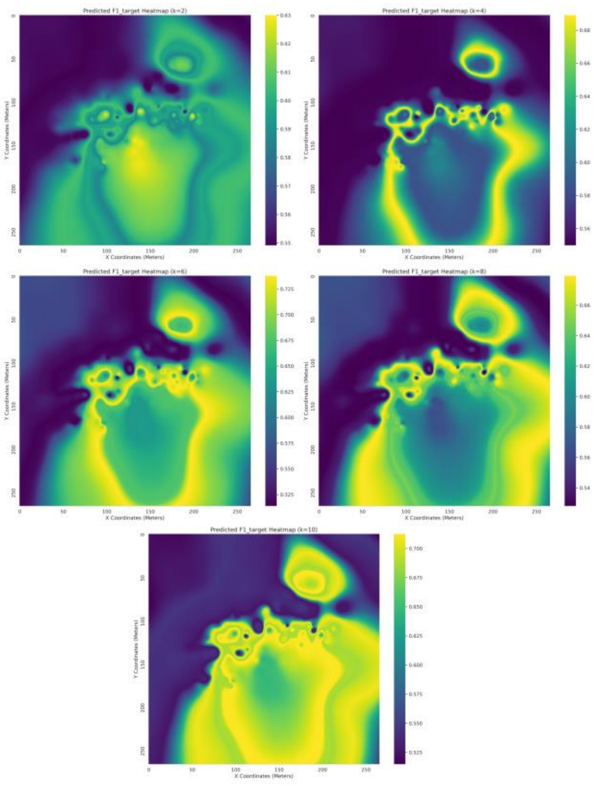

Fig 9 Prediction Result Graph

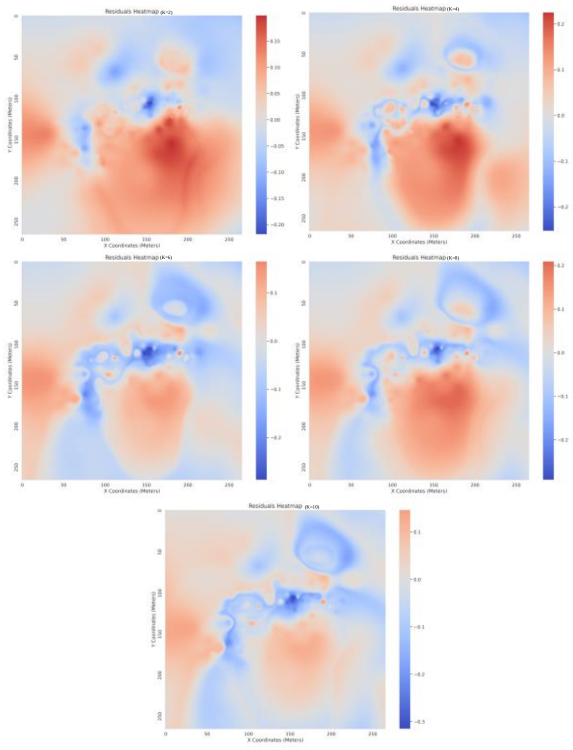

Fig 10 Residual Heatmap

In this problem, we combined dynamic graphs with different K values and hybrid Graph Neural Networks (GNNs) for modeling and analysis. Specifically, when K=10, the model predicts the target variable by integrating information from the 10 neighboring points around each node. The experimental results indicate that utilizing feature information from the surrounding 10 points is sufficient to accurately predict the target variable, demonstrating that selecting K=10 is a reasonable and effective decision in the current dataset and task.

GNNs, through graph convolution and attention mechanisms, efficiently integrate and propagate information from neighboring nodes, capturing complex relationships and dependencies between nodes, especially excelling in handling spatial correlations and feature interactions. This not only improves prediction accuracy but also significantly enhances computational efficiency, particularly suitable for large-scale graph data. Additionally, GNNs perform exceptionally well when dealing with dynamically changing data. As the topology of the dynamic graph adjusts with changes in the K value, GNNs can promptly adapt to these changes, maintaining model stability and prediction performance.

The Residuals Heatmap displays the spatial distribution of the model's prediction errors across different K values. As the K value increases from 2 to 10, the heatmap colors gradually shift from red (indicating high error) to blue (indicating low error), suggesting a significant reduction in prediction error. This demonstrates a close relationship between the collaborative variables 1 (F1_collaborative_variable1) and 4 (F1_collaborative_variable4) and the target variable (F1_target), and incorporating information from more neighboring points enhances the model's predictive capability.

The Pearson and Spearman correlation coefficients between collaborative variable 1 and the target variable are both 0.76, indicating a strong linear and monotonic relationship. Collaborative variable 1 significantly enhances prediction accuracy, especially as the K value increases by integrating more neighboring information. In contrast, collaborative variable 4 has a lower correlation coefficient of 0.32, suggesting moderate predictive ability, but it enriches the feature space and promotes diversity in representation. Combining both variables allows the model to learn diversified features, enhancing generalization capability. Collaborative variable 4 also synergizes with other variables to optimize predictions, reducing residuals and errors.

In summary, collaborative variable 1 provides strong informational support due to its high correlation, while collaborative variable 4 adds valuable supplementary information despite its lower correlation. The integration of dynamic graphs and hybrid GNNs shows promise in computational efficiency and adaptability, effectively handling complex spatial data for improved predictions.

## 5.4 Establishment and Solution of Model for Problem Three

Based on the results of Question 2, it is evident that the collaborative variables collaborative_1 and collaborative_4 have significant correlations with the target variable F1_target_variable. Therefore, we utilize the data of these collaborative variables to study the variation patterns of the spatial variable (F1 target variable).

To facilitate random sampling, we flatten the two-dimensional matrix of each variable into a one-dimensional array. This flattened data structure simplifies the selection of sampling points at a specific proportion and also facilitates subsequent interpolation calculations. By extracting the row and column indices of the matrix, we can generate spatial coordinates for each data point, which will be used in the interpolation algorithm to determine the position of each point in space.

During random sampling, we will sample the target variable and selected collaborative variables according to different sampling ratios. Specifically, we will set multiple sampling ratios such as 1%, 5%, and 10%, and then randomly select the corresponding number of sample points from the flattened arrays.

After sampling is completed, we will use the generated spatial coordinates and sampled values to perform interpolation calculations. The interpolation algorithm will estimate the variable values at unsampled locations based on the values and spatial positions of the known sample points. In this question, we will compare two interpolation estimation methods, namely Inverse Distance Weighting (IDW) and Dynamic K-Nearest Neighbors (Dynamic KNN), under different sampling ratios.

**IDW Interpolation:**

IDW (Inverse Distance Weighting) is a commonly used spatial interpolation method widely applied in Geographic Information Systems (GIS) and spatial data analysis. Its basic principle is based on the distance relationship between spatial data points, assuming that data points closer to the target point have a greater influence on it, while those farther away have a lesser influence. The basic principles are as follows:

Weight Calculation: In the IDW method, the influence of each known data point on the target point is weighted according to its distance from the target point. Specifically, the closer the distance, the greater the weight; the farther the distance, the smaller the weight. The formula for calculating the weight is:

$$\omega_i = \frac{1}{d_i^p} \tag{7}$$

Where $\omega_i$ is the weight of the ith known point, $d_i$ is the distance from this point to the target point, and $p$ is a non-negative power exponent, typically set to 2.

Interpolation Calculation: The estimated value of the target point is the weighted average of the values of all known points, and its calculation formula is:

$$Z(x) = \frac{\sum_{i=1}^{n} \omega_i \cdot Z_i}{\sum_{i=1}^{n} \omega_i} \tag{8}$$

Where $Z(x)$ is the estimated value of the target point, $Z_i$ is the value of the ith known point, and $n$ is the number of known points.

**Dynamic K-Nearest Neighbors Interpolation:**

Dynamic K-Nearest Neighbors (Dynamic KNN) is an extension based on the K-Nearest Neighbors (KNN) algorithm, designed to enhance the ability to process data in dynamic environments. Traditional KNN algorithms rely on the nearest neighbor samples in a static dataset for classification or regression. However, in many practical applications, data is dynamically changing, with samples potentially being continuously added, deleted, or updated, necessitating an algorithm that can adapt to these changes. Dynamic KNN introduces a data update mechanism, enabling the algorithm to quickly adjust its neighbor selection when changes occur in the dataset.

To compare the effectiveness of these two interpolation methods under different sampling ratios, we use mean squared error (MSE), mean absolute error (MAE), and maximum error as error indicators, and visualize the results.

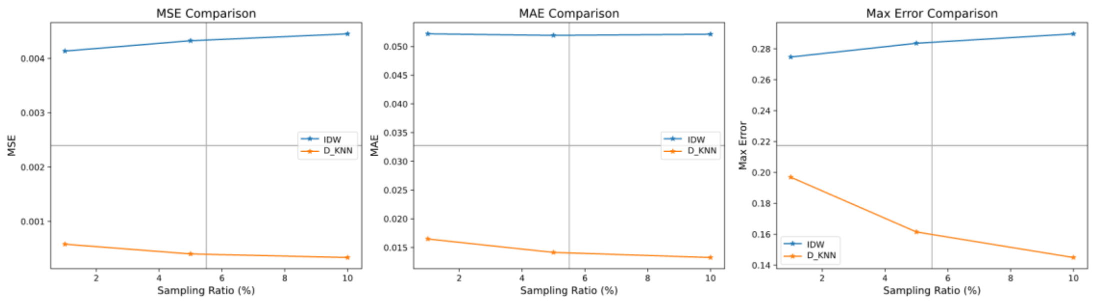

Fig 11 Comparison Chart of MSE, MAE, and MAX Error

As can be seen from the comparison chart, the MSE, MAE, and MAX Error of the D_KNN interpolation method decrease as the sampling ratio increases, and all the error indicators are smaller than those of the IDW interpolation method, demonstrating a good interpolation effect.

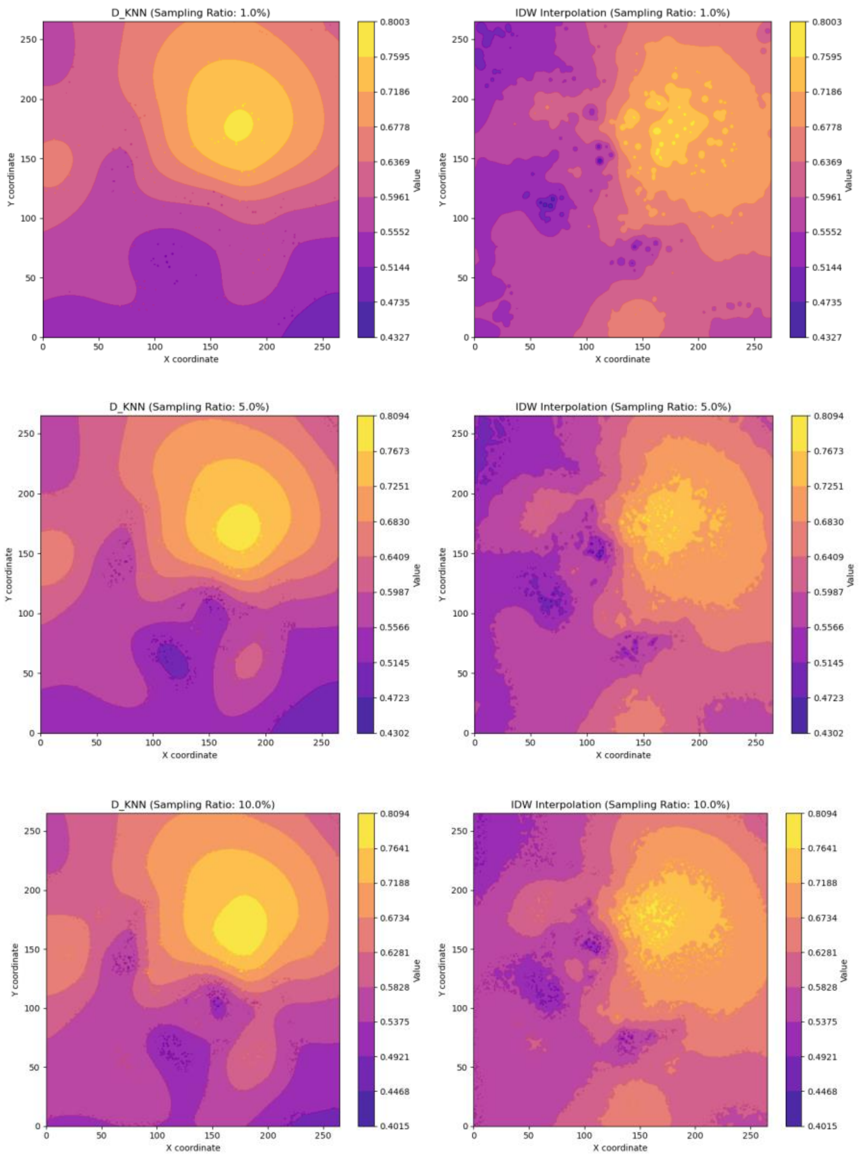

Fig 12 Heatmap of D_KNN Interpolation Method and IDW Interpolation Method

Analyzing the contour maps reveals a significant trend: the D_KNN interpolation method performs better in capturing spatial variations, producing contour maps with smoother contours. As the sampling ratio increases, the spatial distribution accuracy of the D_KNN interpolation method improves.

In contrast, the contour maps of IDW interpolation show more scattered spatial variations, especially when the sampling ratio is low, where the clarity of the spatial structure distribution is significantly lower than that of Kriging interpolation.

In summary, the performance of the D_KNN interpolation method is generally superior to that of the IDW interpolation method at different sampling ratios, especially at lower sampling ratios where the D_KNN method maintains higher accuracy. Furthermore, as the sampling ratio increases, the error of the D_KNN interpolation method decreases further, indicating that it is more suitable for estimating variables with spatial correlation.

## 5.5 Establishment and Solution of Model for Problem Four

Reading the Target Variable from Appendix 2,we successfully loaded the data for the target variable and four collaborative variables, and generated contour maps for each collaborative variable. Below is an analysis of the loaded data and images:

An Excel file containing the target variable was loaded, and its first few rows were displayed. The data included column sequence, row sequence, X coordinates, Y coordinates, and the target property (Target Property). The data files for each collaborative variable were successfully read and converted into a 266x266 matrix format, facilitating subsequent analysis and interpolation operations.

In the generated 2x2 contour map layout, each subplot displays the spatial distribution characteristics of one collaborative variable. The images utilized the "viridis" color scheme, enhancing the visual effect of gradients and making high-value areas more prominent.

Below is an analysis of each collaborative variable:

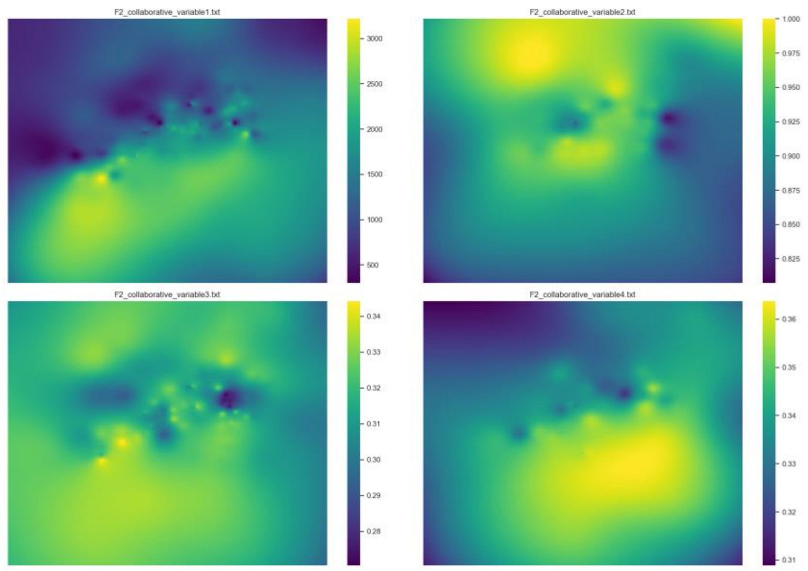

Fig 13 Spatial Distribution Maps of Each Collaborative Variable

1.F2_collaborative_variable1.txt: This variable exhibits significant spatial gradient changes, with particularly prominent high-value areas in the central region and the upper right. Some relatively obvious fluctuations in the spatial distribution may be correlated with the target variable.

2.F2_collaborative_variable2.txt: The high-value areas of this collaborative variable are relatively concentrated, mainly located on the right and top sides. This distribution characteristic indicates that the variable fluctuates greatly in different regions, potentially suggesting unique spatial trends.

3.F2_collaborative_variable3.txt: The variable shows higher values in the central region, gradually decreasing towards the periphery, with an overall relatively flat distribution. The fluctuations in the central region are more pronounced, showing some differences compared to other collaborative variables.

4.F2_collaborative_variable4.txt: The high-value areas of this variable are relatively upper, with an overall uniform distribution. There are fewer fluctuations in the distribution, and the changes are smooth, potentially suitable for use as an auxiliary estimator for the target variable.

The spatial distribution characteristics of each collaborative variable differ significantly, indicating that they may represent different environmental features in terms of physical properties.

The distribution patterns of F2_collaborative_variable1 and F2_collaborative_variable3 exhibit strong spatial gradients and potential high correlation with the target variable, making them suitable as collaborative estimation variables. Subsequent steps will explore the estimation effect of these variables on the target variable through spatial interpolation methods, verifying their consistency and physical significance in spatial distribution.

In this analysis, we extracted the spatial coordinates of the data to accurately reflect each sampling point's position during interpolation. Using Adaptive K-Nearest Neighbor (AKNN) interpolation and a variogram model, we modeled the spatial correlation of the sampling points, fitting parameters like range, sill, and nugget effect. AKNN interpolation was then applied to the target grid to generate estimated values for each location, and contour maps were plotted with appropriate color mappings and resolutions to effectively display spatial trends. By examining these maps, we identified high- and low-value areas and analyzed spatial patterns, enhancing our understanding of the target variable's distribution characteristics and providing a foundation for further research and applications.

In dynamic K-Nearest Neighbor (D_KNN) interpolation, we estimate the target variable at unsampled locations using the already sampled values of the target variable and additional collaborative variables that have spatial correlation with the target variable. Below is a step-by-step explanation of the formula and principles of D_KNN:

Define a set containing n training samples $X = [x_1, x_2, \cdots, x_n] \in R^{d\times n}$, where the ith training sample is defined as $x_i = [x_{i1}, x_{i2}, \cdots, x_{in}]^T \in R^d$, (i=1,2,…,n). The objective function corresponding to dynamic K-Nearest Neighbor (D_KNN) is::

$$\min \| XW - X \|_F^2 + \rho_1 \| W \|_1 + \rho_2 Tr(W^T X^T LXW) \tag{9}$$

In the equation, $W = [w_1, w_2, \cdots, w_n] \in R^{n \times n}$ denotes the correlation matrix among training samples, $L$ is the Laplacian matrix, and $\| W \|_1$ is the regularization term of the objective function. In the above formula, regarding the relationship between two sample points, if two sample points $x_i$ and $x_j$ are adjacent, then their corresponding new sample points $x_i'$ and $x_j'$ should also have the same relationship. Locality Preserving Projections (LPP) can maintain the internal local structure of data while reducing the dimensionality of the space. The relationship maintained between sample points $x_i$ and $x_j$ by LPP can be expressed as:

$$\sum_{i,j} (x_i' - x_j')^2 S_{ij} = \sum_{i,j} (W^T x_i^T - W^T x_j^T)^2 S_{ij} \tag{10}$$

Where, $S_{ij}$ represents the weight matrix, which indicates the degree of correlation between sample points $x_i$ and $x_j$. The specific calculation of the weight matrix $S_{ij}$ is as follows::

$$S_{ij} = \begin{cases} \exp(-\frac{\| x_i - x_j \|_2^2}{2\sigma^2}), x_j \text{ belongs to the } k \text{ nearest neighbors of } x_i \\ 0, \qquad \text{others} \end{cases} \tag{11}$$

The above equation only considers the distances between samples and their nearest neighbors. However, through supervised methods, both the distances to the nearest neighbor samples and the label information can be taken into account simultaneously. That is to say, when a neighborhood has the same label as sample $x_i$, it should be assigned a larger weight. Otherwise, a smaller weight will be assigned. Thus, the weight matrix $S_{ij}$ is obtained in the following form:

$$S_{ij} = \begin{cases} \exp(-\frac{\| x_i - x_j \|_2^2}{2\sigma^2})(1 + \exp(-\frac{\| x_i - x_j \|_2^2}{2\sigma^2})), x_j \text{ belongs to the } k \text{ nearest neighbors of } x_i \text{ and } y_i = y_j \\ \exp(-\frac{\| x_i - x_j \|_2^2}{2\sigma^2})(1 - \exp(-\frac{\| x_i - x_j \|_2^2}{2\sigma^2})),_j \text{ belongs to the } k \text{ nearest neighbors of } x_i \text{ and } y_i \neq y_j \\ 0, \qquad \text{others} \end{cases} \tag{12}$$

The above equation introduces intra-class weights and inter-class weights. Simply put, when $x_i$ and $x_j$ belong to the same class, they will be assigned a larger weight. Otherwise, when $x_i$ and $x_j$ do not belong to the same class, they will be assigned a smaller weight. Therefore, the following formula can be obtained::

$$\frac{1}{2} \sum_{i,j} (x_i' - x_j')^2 S_{ij} = \sum_{i,j} (W^T x_i^T - W^T x_j^T)^2 S_{ij} = \sum_{i,j} W^T x_i^T S_{ij} x_{ij} W - \sum_{i,j} W^T x_j^T S_{ij} x_j W \tag{13}$$

Where $x_i'$ and $x_j'$ represent the reconstructed samples of $x_i$ and $x_j$, respectively. Let $D_{ii} = \sum_j S_{ji}$, where $D_{ii}$ denotes the metric importance of the ith sample. Thus, we have::

$$\begin{aligned}\frac{1}{2}\sum_{i,j}(x_i^{'}-x_j^{'})^2 S_{ij} &= \sum_i W^T x_i^T D_{ii} x_i W - W^T x^T \\ &= W^T X^T DXW - W^T X^T SXW \\ &= W^T X^T (D-S) XW \\ &= W^T X^T LXW \end{aligned} \tag{14}$$

Where the Laplacian matrix $L = D-S$ . From the result of the above calculation, we can obtain $Tr(W^T X^T LXW)$.

By iteratively solving for the optimal correlation matrix $W$ in the above equation, it is denoted as $W^*$. $W^*$ is expressed as follows:

$$W^* = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ w_{31} & w_{32} & \cdots & w_{3n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nn} \end{bmatrix} \tag{15}$$

Where $w_{ij} = (i,j=1,2,\ldots,n)$represents the weight of the correlation degree between training samples $x_i$ and $x_j$. The k-value corresponding to each sample is determined based on the number of weight values in each column of $W^*$. The optimal k-value for the jth training sample is calculated as follows:

$$k_j = \sum([w_{ij}]+1), i = 1,2,\cdots,n \tag{16}$$

Each $k$ is calculated from the optimal matrix $W^*$. The number of positive values in each column is used to determine the optimal $k_j$. Then, a decision tree is constructed using the training samples and their optimal k-values, and the decision tree is used to find different optimal k-values for new samples.

We calculate the results of the target variable and collaborative variables through dynamic K-Nearest Neighbors (KNN), and implement dynamic KNN interpolation using a linear combination method. This method is suitable for situations with limited computing resources or data constraints, while still achieving reasonable estimation results.

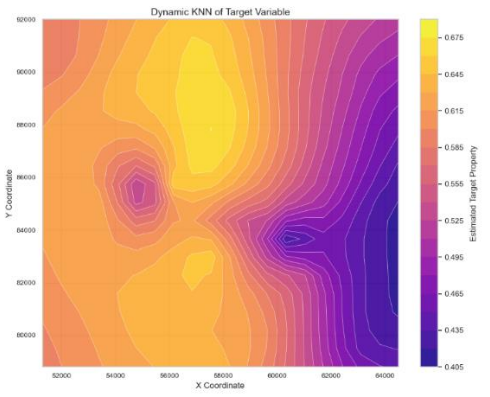

Fig 14 Spatial distribution map after dynamic KNN interpolation

The image employs the "plasma" color scheme to illustrate the spatial distribution trend of the target variable, with a contour map generated through D_KNN interpolation.

Central Region: Yellow areas indicate high target variable values concentrated in the middle section (roughly between 54,000 and 58,000 with Y near 86,000), suggesting potential influences from environmental or geological characteristics.

Edge Regions: Deep blue and purple represent low target variable values on the map's right and left edges, likely affected by local terrain or other factors.

Smooth Transition: The contours transition smoothly from yellow to blue, reflecting a gradual spatial change in values, consistent with geographical data's spatial correlation.

Effectiveness of Interpolation Model: The map demonstrates the spatial variation trend effectively, as the dynamic KNN interpolation model captures the target variable's distribution characteristics accurately within the study area.

# 6. Model Evaluation

## 6.1 Advantages of the Model

**1.Stronger Adaptability:** Dynamic KNN can automatically adjust the K-value based on the local density of the data, better adapting to different density regions within the dataset.

**2.Higher Prediction Accuracy:** By adjusting the K-value, Dynamic KNN can more accurately reflect the true distribution of the data, thereby improving prediction accuracy.

**3.High Flexibility:** The model parameters can be flexibly adjusted according to the characteristics and requirements of the data to adapt to different application scenarios.

**4.Suitable for Multi-class Problems:** Dynamic KNN can effectively handle multi-class problems and is applicable to data with multiple categories.

**5.Robustness:** It has a certain degree of robustness to noisy data and can tolerate outliers in the data to a certain extent.

**6.No Training Required:** The KNN algorithm does not have an explicit training phase and only requires storing the training dataset, resulting in faster modeling speeds.

**7.No Data Input Assumptions:** The algorithm does not require assumptions about data independence, lowering the usage threshold.

**8.Applicable to Non-linear Problems:** The KNN algorithm can effectively handle non-linear problems and is suitable for complex data distributions.

## 6.2 Disadvantages of the Model

**1.High Computational Cost:** On large datasets, the computational load of Dynamic KNN remains significant because it requires calculating the distance between the sample to be classified and each sample in the training set.

**2.High Storage Cost:** The entire training dataset needs to be stored, which can be costly for large-scale datasets.

**3.Influenced by Initial K-value Selection:** Although Dynamic KNN can adjust the K-value, the initial choice of K can still have an impact on model performance.

**4.Sensitive to Outliers:** Despite its robustness, Dynamic KNN can still be significantly affected by outliers in the classification results.

# 7. Model Improvement and Promotion

## 7.1 Improvements of the Model

Improvements to the Dynamic KNN Model [5] primarily focus on enhancing its efficiency, accuracy, and adaptability. Here are some common improvement methods:

**1.Optimizing K-value Selection**

Dynamic Adjustment of K-value: Select the K-value dynamically based on the local density or distribution characteristics of the data. For example, use a smaller K-value in dense regions and a larger K-value in sparse regions.

Cross-validation: Use cross-validation techniques to determine the optimal K-value, reducing the risk of overfitting or underfitting.

**2.Optimizing Distance Metrics**

Weighted Distance: Consider the impact of different features on classification results and assign different weights to different features to optimize distance metrics.

Alternative Distance Metrics: Besides the commonly used Euclidean distance, try different distance metrics such as Manhattan distance, Chebyshev distance, and cosine similarity to accommodate different types of data.

**3.Feature Selection**

Eliminating Irrelevant Attributes: Use feature selection techniques to eliminate attributes unrelated to classification results, reducing computational load and improving model accuracy.

Attribute Weighting: Assign different weights to different attributes to reflect their importance in classification.

## 7.2 Promotion of the Model

The promotion of the Dynamic KNN Model involves multiple aspects, including the expansion of application scenarios, technical optimization and integration, and marketing promotion strategies.

**1.Image Recognition:**

In the field of image recognition, the Dynamic KNN Model can be used for classifying and recognizing objects or scenes in images. By extracting image features and calculating the similarity to known categories, automatic image classification and recognition can be achieved.

**2.Text Classification:**

The Dynamic KNN Model can be applied to text classification tasks, such as news categorization and email filtering. By calculating the similarity between texts, texts can be categorized into the most similar category.

**3.Financial Risk Control:**

In the field of financial risk control, the Dynamic KNN Model can be used to identify abnormal transactions and fraudulent behaviors. By analyzing transaction data and calculating the similarity to normal transactions, potential fraudulent activities can be identified.

**4.Healthcare:**

In the healthcare field, the Dynamic KNN Model can be used for disease prediction and diagnosis. By analyzing patients' medical records, test results, and other data, and calculating the similarity to known cases, doctors can be assisted in diagnosing and treating diseases.

# References

[1]韦新鹏,姚中洋,宝文礼,等.一种基于主动学习克里金模型的证据理论可靠性分析方法[J].机械工程学报,2024,60(02):356-368.
[2]Hyune-Ju K ,Jun L ,Huann-Sheng C , et al.Improved confidence interval for average annual percent change in trend analysis.[J].Statistics in medicine,2017,36(19):3059-3074.
[3]Pham T ,Bui H ,Nguyen M , et al.Towards Multiple-View Nested Graph Convolutional Networks for stable and robust lithium-ion battery forecasting[J].Energy Reports,2024,124777-4793.
[4]Xiang Z ,Yeyin X ,Sai Z , et al.Improved vibration performance of a nonlinear rotor-seal system with modified labyrinth seals by an interpolating database method[J].Tribology International,2024,191109170-.
[5]Charlotte C Z ,Linlin L ,X. S D .Multiplicative Fault Detection and Isolation in Dynamic Systems Using Data-Driven K-Gap Metric based kNN Algorithm[J].IFAC PapersOnLine,2022,55(6):169-174.

# Appendix

1. Problem 1 codes

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import torch
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv, GATConv
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from torch_geometric.utils import add_self_loops
import networkx as nx
from torch_geometric.utils import to_networkx
from torch_geometric.utils import to_scipy_sparse_matrix

# 1. Read the TXT file
with open('Attachment 1\F1_target_variable.txt', 'r') as file:
    lines = file.readlines()

# 2. Extract grid parameters (first 5 lines)
column_span = lines[0].strip().split('：')[1].split(', ')
row_span = lines[1].strip().split('：')[1].split(', ')
column_grid_interval = float(lines[2].strip().split('：')[1])
row_grid_interval = float(lines[3].strip().split('：')[1])
grid_size = lines[4].strip().split('：')[1].split('X')
grid_rows = int(grid_size[0])
grid_cols = int(grid_size[1])
print(f'grid_rows: {grid_rows}, grid_cols: {grid_cols}')

# 3. Extract data values
data_values = []
for line in lines[5:]:
    values = [float(val) for val in line.strip().split()]
    data_values.extend(values)

# 4. Check the number of data points
expected_points = grid_rows * grid_cols    # 266 * 266 = 70756
actual_points = len(data_values)
print(f'Expected data points: {expected_points}, Actual data points: {actual_points}')
```

```
    if actual_points != expected_points:
        print("Warning: Number of data points does not match grid size! Handling
missing data.")
        if actual_points < expected_points:
            # Fill missing data points with NaN
            data_values.extend([np.nan] * (expected_points - actual_points))
        else:
            # Truncate excess data points
            data_values = data_values[:expected_points]

    # 5. Convert to 2D array
    data_grid = np.array(data_values).reshape((grid_rows, grid_cols))

    # 6. Generate X and Y coordinates
    x_start, x_end = float(column_span[0]), float(column_span[1])
    y_start, y_end = float(row_span[0]), float(row_span[1])
    x_coords = np.linspace(x_start, x_end, grid_cols)
    y_coords = np.linspace(y_start, y_end, grid_rows)

    # 7. Ensure grid dimensions match
    assert len(x_coords) == grid_cols, f"Number of X coordinates ({len(x_coords)})
does not match grid_cols ({grid_cols})"
    assert len(y_coords) == grid_rows, f"Number of Y coordinates ({len(y_coords)})
does not match grid_rows ({grid_rows})"

    # 8. Create DataFrame
    X, Y = np.meshgrid(x_coords, y_coords)
    df = pd.DataFrame({
        'X': X.ravel(),
        'Y': Y.ravel(),
        'F1_target': data_grid.ravel()
    })

    # 9. Preview the first few rows of the DataFrame
    print(df.head())

    # 10. Use Seaborn to plot heatmap (interpolate missing NaN values)
    df_interpolated = df.copy()
    df_interpolated['F1_target']                                                    =
df_interpolated['F1_target'].interpolate(method='linear')

    # Convert to 2D array for heatmap
    Z_pred = df_interpolated['F1_target'].values.reshape((grid_rows, grid_cols))
    # Set Seaborn theme
    sns.set_theme(style="whitegrid")

    # Create heatmap
    plt.figure(figsize=(12, 10))
    ax = sns.heatmap(Z_pred, cmap='viridis', xticklabels=False, yticklabels=False,
cbar=True)

    # Add color bar label
    cbar = ax.collections[0].colorbar
    cbar.set_label('F1_target')

    # Set title and axis labels
    plt.title('F1_target_variable Heatmap', fontsize=16)
    plt.xlabel('X Coordinates (Meters)', fontsize=14)
    plt.ylabel('Y Coordinates (Meters)', fontsize=14)

    # Optimize layout
    plt.tight_layout()

    # Display plot
    plt.show()

    # 11. Define sample sizes
    sample_sizes = [10*10, 20*20, 40*40, 60*60, 80*80]  # [100, 400, 1600, 6400,
25600]

    # 12. Initialize dictionary for sampled data
    sampled_data_dict = {}
    for size in sample_sizes:
        # Ensure sample size does not exceed available data points
        valid_data_points = df.dropna(subset=['F1_target']).shape[0]
        if size > valid_data_points:
            print(f"Sample size {size} exceeds available data points
({valid_data_points}). Skipping.")
            continue
        sampled_data = df.dropna(subset=['F1_target']).sample(n=size,
random_state=42)
        sampled_data_dict[size] = sampled_data

    # 13. Define mixed GCN and GAT model
    class MixedGCNGAT(torch.nn.Module):
        def __init__(self, input_dim=3, hidden_dim=64, output_dim=1, gat_heads=8,
dropout_p=0.6):
            super(MixedGCNGAT, self).__init__()

            # First set of GCN layers
            self.gcn1 = GCNConv(input_dim, hidden_dim)
            self.bn1 = torch.nn.BatchNorm1d(hidden_dim)

            # Second set of GCN layers
            self.gcn2 = GCNConv(hidden_dim, hidden_dim)
            self.bn2 = torch.nn.BatchNorm1d(hidden_dim)

            # Third set of GCN layers
            self.gcn3 = GCNConv(hidden_dim, hidden_dim)
            self.bn3 = torch.nn.BatchNorm1d(hidden_dim)

            # First set of GAT layers
            self.gat1   =   GATConv(hidden_dim,   hidden_dim   //   gat_heads,
heads=gat_heads, dropout=dropout_p)
            self.bn4 = torch.nn.BatchNorm1d(hidden_dim)

            # Second set of GAT layers
            self.gat2   =   GATConv(hidden_dim,   hidden_dim   //   gat_heads,
heads=gat_heads, dropout=dropout_p)
            self.bn5 = torch.nn.BatchNorm1d(hidden_dim)

            # Output layer
            self.gat3 = GATConv(hidden_dim, output_dim, heads=1, concat=False,
dropout=dropout_p)

            # Activation function and Dropout
            self.relu = torch.nn.LeakyReLU()
            self.dropout = torch.nn.Dropout(p=dropout_p)

            # Residual connections
            self.residual1 = torch.nn.Linear(hidden_dim, hidden_dim)
            self.residual2 = torch.nn.Linear(hidden_dim, hidden_dim)
            self.residual3 = torch.nn.Linear(hidden_dim, hidden_dim)

        def forward(self, data):
            x, edge_index = data.x, data.edge_index

            # First set of GCN layers + residual connections
            x = self.gcn1(x, edge_index)
            residual = self.residual1(x)
            x = self.bn1(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + residual    # Residual connection

            # Second set of GCN layers + residual connections
            x = self.gcn2(x, edge_index)
            residual = self.residual2(x)
            x = self.bn2(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + residual    # Residual connection

            # GraphSAGE layer
            x = self.gcn3(x, edge_index)
            residual = self.residual3(x)
            x = self.bn3(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + residual    # Residual connection

            # First set of GAT layers
            x = self.gat1(x, edge_index)
            x = self.bn4(x)
            x = self.relu(x)
            x = self.dropout(x)

            # Second set of GAT layers
            x = self.gat2(x, edge_index)
            x = self.bn5(x)
            x = self.relu(x)
            x = self.dropout(x)

            # Output layer
            x = self.gat3(x, edge_index)

            return x

# 14. Define function to create graph with self-loops
def create_graph(sampled_data, grid_rows, grid_cols, k=1):
    """
    Create a graph with k-hop neighbors.

    Parameters:
```

```
    - sampled_data: pandas DataFrame, contains sampled data.
    - grid_rows: int, number of rows in the grid.
    - grid_cols: int, number of columns in the grid.
    - k: int, hop distance.

    Returns:
    - data_graph: torch_geometric.data.Data, the created graph data object.
    """
    # Create node features
    features = sampled_data[['X', 'Y', 'F1_target']].values
    x = torch.tensor(features, dtype=torch.float)

    # Initialize edge list
    edges = set()    # Use set to avoid duplicate edges
    sampled_indices = sampled_data.index.tolist()

    # Build graph edges (k-hop neighborhood)
    for i in range(len(sampled_indices)):
        for j in range(i+1, len(sampled_indices)):
            edges.add((i, j))    # Connect nodes i and j

    edge_index = torch.tensor(list(edges), dtype=torch.long).t().contiguous()

    # Add self loops
    edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))

    data_graph = Data(x=x, edge_index=edge_index)

    return data_graph
# 15. Initialize error results storage dictionary
error_results = {
    'Sample Size': [],
    'Neighbor Distance': [],
    'MSE': [],
    'RMSE': [],
    'MAE': []
}

i = 1

# 16. Training and validation loop
for size in sample_sizes:
    if size not in sampled_data_dict:
        print(f"Sample size {size} was skipped due to insufficient data points.")
```

```
            continue
        sampled_data = sampled_data_dict[size]

        # Split into training and test sets
        train_data,    test_data    =    train_test_split(sampled_data,    test_size=0.2,
random_state=42)

        # Dynamically set k (hop distance) based on sample size
        if size <= 400:
            k = 6
        elif size <= 1600:
            k = 8
        else:
            k = 10

        # Create training and testing graphs
        train_graph = create_graph(train_data, grid_rows, grid_cols, k=k)
        test_graph = create_graph(test_data, grid_rows, grid_cols, k=k)

        # Convert PyG Data object to NetworkX graph
        G = to_networkx(train_graph, to_undirected=True)

        # Plot the graph
        plt.figure(figsize=(12, 12))
        nx.draw(G, node_size=50, with_labels=False, edge_color='gray')
        plt.title('Graph Structure')
        plt.savefig(f'问题 1\Graph Structure {i}')
        plt.show()

        # Assuming edge_index is your edge index
        edge_index = train_graph.edge_index
        num_nodes = train_graph.num_nodes

        # Convert to sparse adjacency matrix
        adj_matrix            =            to_scipy_sparse_matrix(edge_index,
num_nodes=num_nodes).todense()

        # Use Seaborn to plot the heatmap
        plt.figure(figsize=(10, 8))
        sns.heatmap(adj_matrix, cmap='Blues')
        plt.title('Adjacency Matrix Heatmap')
        plt.xlabel('Nodes')
        plt.ylabel('Nodes')
        plt.savefig(f'问题 1\Adjacency Matrix Heatmap {i}')
        plt.show()
        i += 1

        # Print the number of edges to ensure they are correctly added
        print(f'Number of edges in training graph: {train_graph.edge_index.size(1)}')
        print(f'Number of edges in testing graph: {test_graph.edge_index.size(1)}')

        # Initialize model, loss function, and optimizer
        model = MixedGCNGAT()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.005,
weight_decay=5e-4)
        criterion = torch.nn.MSELoss()

        # Train the model
        num_epochs = 200
        for epoch in range(num_epochs):
            model.train()
            optimizer.zero_grad()
            out = model(train_graph)
            loss = criterion(out.squeeze(), train_graph.x[:, 2])
            loss.backward()
            optimizer.step()

            if (epoch + 1) % 50 == 0:
                print(f'Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.6f}')

        # Validate the model
        model.eval()
        with torch.no_grad():
            val_pred = model(test_graph).squeeze().numpy()
            val_true = test_graph.x[:, 2].numpy()
            mse = mean_squared_error(val_true, val_pred)
            rmse = np.sqrt(mse)
            mae = mean_absolute_error(val_true, val_pred)

        # Add predicted values to DataFrame
        df[f'Predicted_F1_target_{size}'] = val_pred
        # Store error results
        error_results['Sample Size'].append(size)
        error_results['Neighbor Distance'].append(k)
        error_results['MSE'].append(mse)
        error_results['RMSE'].append(rmse)
        error_results['MAE'].append(mae)
```

```
        print(f'Sample Size: {size}, Neighbor Distance (k): {k}, MSE: {mse:.6f}, RMSE: {rmse:.6f}, MAE: {mae:.6f}')

    # 17. Create error DataFrame
    error_df = pd.DataFrame(error_results)

    # 18. Use Seaborn to plot error relationship graph
    sns.set(style="whitegrid", font_scale=1.2)

    plt.figure(figsize=(10, 6))
    sns.lineplot(data=error_df, x='Sample Size', y='MSE', marker='o', label='MSE')
    sns.lineplot(data=error_df, x='Sample Size', y='RMSE', marker='s', label='RMSE')
    sns.lineplot(data=error_df, x='Sample Size', y='MAE', marker='^', label='MAE')

    plt.title('Relationship Between Sample Size and Estimation Error', fontsize=16)
    plt.xlabel('Sample Size', fontsize=14)
    plt.ylabel('Error', fontsize=14)
    sns.despine()
    plt.legend(title='Error Metrics')
    plt.tight_layout()
    plt.savefig('问题 1\SampleSize_Error_Relationship_seaborn.png', dpi=300)
    plt.show()

    # 19. Use Seaborn style to plot contour plot
    plt.figure(figsize=(12, 10))
    contour = plt.contourf(X, Y, Z_pred, levels=100, cmap='viridis')
    plt.colorbar(contour, label='F1_target')
    plt.title('F1_target Contour Plot', fontsize=16)
    plt.xlabel('X Coordinates (Meters)', fontsize=14)
    plt.ylabel('Y Coordinates (Meters)', fontsize=14)
    sns.despine()
    plt.tight_layout()
    plt.savefig('问题 1\F1_target_contour_seaborn.png', dpi=300)
    plt.show()

    # 20. Use Seaborn to plot heatmap with adjusted scale label density
    plt.figure(figsize=(12, 10))
    sns.heatmap(Z_pred, cmap='viridis', cbar=True,
                xticklabels=50, yticklabels=50)    # Adjust scale label density
    plt.title('F1_target Heatmap', fontsize=16)
    plt.xlabel('X Coordinates (Meters)', fontsize=14)
    plt.ylabel('Y Coordinates (Meters)', fontsize=14)
    sns.despine()
    plt.tight_layout()
```

```
plt.savefig('问题 1\F1_target_heatmap_seaborn.png', dpi=300)
plt.show()
```

2. problem 2 codes

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import torch
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv, GATConv
import torch.nn.functional as F
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error
from torch_geometric.utils import add_self_loops
import networkx as nx
from torch_geometric.utils import to_scipy_sparse_matrix
from scipy.spatial.distance import pdist, squareform
from scipy.stats import pearsonr
import os
from sklearn.neighbors import NearestNeighbors
from mpl_toolkits.mplot3d import Axes3D
from torch_geometric.utils import subgraph, to_networkx

# Define the base path
base_path = 'Attachment 1'

# List of filenames
filenames = [
    'F1_target_variable.txt',
    'F1_collaborative_variable1.txt',
    'F1_collaborative_variable2.txt',
    'F1_collaborative_variable3.txt',
    'F1_collaborative_variable4.txt'
]

# Read the files
lines_list = []
for filename in filenames:
    file_path = os.path.join(base_path, filename)
    with open(file_path, 'r') as file:
        lines_list.append(file.readlines())

# Unpack the list
```

```
    lines_target, lines_variable1, lines_variable2, lines_variable3, lines_variable4 =
lines_list


    # 2. Extract grid parameters (first 5 lines)
    column_span = lines_target[0].strip().split('： ')[1].split(', ')
    row_span = lines_target[1].strip().split('： ')[1].split(', ')
    column_grid_interval = float(lines_target[2].strip().split('： ')[1])
    row_grid_interval = float(lines_target[3].strip().split('： ')[1])
    grid_size = lines_target[4].strip().split('： ')[1].split('X')
    grid_rows = int(grid_size[0])
    grid_cols = int(grid_size[1])
    print(f'grid_rows: {grid_rows}, grid_cols: {grid_cols}')

    # 3. Extract data values
    data_target_values = []
    data_variable1_values = []
    data_variable2_values = []
    data_variable3_values = []
    data_variable4_values = []

    for line in lines_target[5:]:
        values = [float(val) for val in line.strip().split()]
        data_target_values.extend(values)

    for line in lines_variable1[5:]:
        values = [float(val) for val in line.strip().split()]
        data_variable1_values.extend(values)

    for line in lines_variable2[5:]:
        values = [float(val) for val in line.strip().split()]
        data_variable2_values.extend(values)

    for line in lines_variable3[5:]:
        values = [float(val) for val in line.strip().split()]
        data_variable3_values.extend(values)

    for line in lines_variable4[5:]:
        values = [float(val) for val in line.strip().split()]
        data_variable4_values.extend(values)

    # 4. Check the number of data points
    expected_points = grid_rows * grid_cols    # 266 * 266 = 70756
    actual_target_points = len(data_target_values)
    print(f'Expected data points: {expected_points}, Actual data points:
{actual_target_points}')

    actual_variable1_points = len(data_variable1_values)
    print(f'Expected data points: {expected_points}, Actual data points:
{actual_variable1_points}')

    actual_variable2_points = len(data_variable2_values)
    print(f'Expected data points: {expected_points}, Actual data points:
{actual_variable2_points}')

    actual_variable3_points = len(data_variable3_values)
    print(f'Expected data points: {expected_points}, Actual data points:
{actual_variable3_points}')

    actual_variable4_points = len(data_variable4_values)
    print(f'Expected data points: {expected_points}, Actual data points:
{actual_variable4_points}')

    # Function to handle missing data
    def handle_missing_data(data_values, expected_points):
        actual_points = len(data_values)
        if actual_points != expected_points:
            print(f"Expected data points: {expected_points}, Actual data points:
{actual_points}")
            if actual_points < expected_points:
                # Fill missing data points with NaN
                data_values.extend([np.nan] * (expected_points - actual_points))
            else:
                # Truncate excess data points
                data_values = data_values[:expected_points]
        return data_values

    # Handle missing data for all variables
    data_target_values = handle_missing_data(data_target_values, expected_points)
    data_variable1_values = handle_missing_data(data_variable1_values,
expected_points)
    data_variable2_values = handle_missing_data(data_variable2_values,
expected_points)
    data_variable3_values = handle_missing_data(data_variable3_values,
expected_points)
    data_variable4_values = handle_missing_data(data_variable4_values,
expected_points)
```

```
    # 5. Convert to 2D arrays
    data_target_grid = np.array(data_target_values).reshape((grid_rows, grid_cols))
    data_variable1_grid    =    np.array(data_variable1_values).reshape((grid_rows,
grid_cols))
    data_variable2_grid    =    np.array(data_variable2_values).reshape((grid_rows,
grid_cols))
    data_variable3_grid    =    np.array(data_variable3_values).reshape((grid_rows,
grid_cols))
    data_variable4_grid    =    np.array(data_variable4_values).reshape((grid_rows,
grid_cols))

    # 6. Generate X and Y coordinates
    x_start, x_end = float(column_span[0]), float(column_span[1])
    y_start, y_end = float(row_span[0]), float(row_span[1])
    x_coords = np.linspace(x_start, x_end, grid_cols)
    y_coords = np.linspace(y_start, y_end, grid_rows)

    # 7. Ensure grid size matches
    assert len(x_coords) == grid_cols, f"Number of X coordinates ({len(x_coords)})
does not match grid_cols ({grid_cols})"
    assert len(y_coords) == grid_rows, f"Number of Y coordinates ({len(y_coords)})
does not match grid_rows ({grid_rows})"

    # 8. Create DataFrame
    X, Y = np.meshgrid(x_coords, y_coords)
    df = pd.DataFrame({
        'X': X.ravel(),
        'Y': Y.ravel(),
        'F1_collaborative_variable1': data_variable1_grid.ravel(),
        'F1_collaborative_variable2': data_variable2_grid.ravel(),
        'F1_collaborative_variable3': data_variable3_grid.ravel(),
        'F1_collaborative_variable4': data_variable4_grid.ravel(),
        'F1_target': data_target_grid.ravel()
    })

    # Assume collaborative variables are stored in the first four columns of the
DataFrame
    variables = ['F1_collaborative_variable1', 'F1_collaborative_variable2',
                 'F1_collaborative_variable3',
'F1_collaborative_variable4','F1_target']

    # Calculate the Pearson correlation coefficient between the collaborative variables
    correlation_matrix = df[variables].corr()
```

```
print("Correlation matrix between collaborative variables:")
print(correlation_matrix)
correlation_matrix.to_csv('问题 2/correlation_matrix.csv')

# Set Seaborn theme
sns.set_theme(style="whitegrid")

# Create heatmap
plt.figure(figsize=(10, 8))
ax = sns.heatmap(
    correlation_matrix,
    cmap='coolwarm',
    annot=True,
    fmt=".2f",
    xticklabels=variables,
    yticklabels=variables,
    cbar=True,
    square=True,
    linewidths=.5,
    linecolor='white'
)

# Set colorbar label
cbar = ax.collections[0].colorbar
cbar.set_label('Correlation Coefficient', fontsize=12)

# Set title and axis labels
plt.title('Correlation Matrix of Variables', fontsize=16)
plt.xlabel('Variables', fontsize=14)
plt.ylabel('Variables', fontsize=14)

# Adjust tick labels
plt.xticks(rotation=45, ha='right', fontsize=12)
plt.yticks(rotation=0, fontsize=12)

# Optimize layout
plt.tight_layout()

# Display chart
plt.savefig('问题 2/Correlation Matrix.png')
plt.show()

# Create an undirected graph
G = nx.Graph()
```

```
    # Add nodes
    for var in variables:
        G.add_node(var)

    # Add edges with weight as the absolute value of the correlation
    for i in variables:
        for j in variables:
            if i != j:
                G.add_edge(i, j, weight=abs(correlation_matrix.loc[i, j]))

    # Get the edge weights for visualization
    edges, weights = zip(*nx.get_edge_attributes(G, 'weight').items())

    # Set plotting parameters
    plt.figure(figsize=(12, 10))
    pos = nx.spring_layout(G, seed=42)   # Use Fruchterman-Reingold layout
algorithm

    # Draw nodes
    nx.draw_networkx_nodes(G, pos, node_size=700, node_color='skyblue')

    # Draw edges
    nx.draw_networkx_edges(G, pos, edge_color='gray', width=[w * 5 for w in
weights])

    # Draw node labels
    nx.draw_networkx_labels(G, pos, font_size=12, font_family='sans-serif')

    # Add edge labels (optional)
    edge_labels = nx.get_edge_attributes(G, 'weight')
    edge_labels = {k: f"{v:.2f}" for k, v in edge_labels.items()}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=10)

    # Set title
    plt.title('Correlation Graph of Variables', fontsize=16)

    # Remove axes
    plt.axis('off')

    # Optimize layout
    plt.tight_layout()

    # Save and display the chart
    plt.savefig('问题 2/correlation_graph.png', dpi=300)
    plt.show()

    # Calculate centrality measures
    degree_centrality = nx.degree_centrality(G)
    betweenness_centrality = nx.betweenness_centrality(G)
    eigenvector_centrality = nx.eigenvector_centrality(G)

    # Create a DataFrame for centrality measures
    centrality_df = pd.DataFrame({
        'Variable': list(degree_centrality.keys()),
        'Degree Centrality': list(degree_centrality.values()),
        'Betweenness Centrality': list(betweenness_centrality.values()),
        'Eigenvector Centrality': list(eigenvector_centrality.values())
    })

    # Print centrality measures
    print("Centrality Measures:")
    print(centrality_df)

    # Extract correlation of target variable with collaborative variables
    target_correlations = correlation_matrix['F1_target'].drop('F1_target').abs()

    # Merge centrality measures with correlation
    centrality_df = centrality_df[centrality_df['Variable'] != 'F1_target']
    centrality_df['Correlation with Target'] = target_correlations.values

    # Sort by correlation with target and select the top two collaborative variables
    top_variables = centrality_df.sort_values(by='Correlation with Target',
ascending=False).head(2)

    print("Selected Collaborative Variables:")
    print(top_variables[['Variable', 'Correlation with Target']])

    # Plot scatter plot
    plt.figure(figsize=(10, 8))
    sns.scatterplot(
        data=centrality_df,
        x='Degree Centrality',
        y='Correlation with Target',
        hue='Variable',
        s=100,
        palette='Set2'
    )
    # Set title and axis labels
    plt.title('Degree Centrality vs Correlation with Target', fontsize=16)
    plt.xlabel('Degree Centrality', fontsize=14)
    plt.ylabel('Absolute Correlation with Target', fontsize=14)

    # Adjust legend
    plt.legend(title='Variable', fontsize=12, title_fontsize=12)

    # Optimize layout
    plt.tight_layout()

    # Save and display chart
    plt.savefig('问题 2/centrality_correlation_scatter.png', dpi=300)
    plt.show()

    # 1. Calculate correlation between collaborative variables and target variable
    variables = ['F1_collaborative_variable1', 'F1_collaborative_variable2',
                 'F1_collaborative_variable3', 'F1_collaborative_variable4']
    correlations = df[variables +
['F1_target']].corr()['F1_target'].drop('F1_target').abs()

    # 2. Select the top 2 collaborative variables with the highest correlation
    top2_vars = correlations.sort_values(ascending=False).head(2).index.tolist()
    print(f"The top two collaborative variables with the highest correlation to the
target variable are: {top2_vars}")

    # 3. Extract features and target variables
    features = df[top2_vars].values
    targets = df['F1_target'].values

    # 4. Define the mixed GCN and GAT model
    class MixedGCNGAT(torch.nn.Module):
        def __init__(self, input_dim=2, hidden_dim=64, output_dim=1, gat_heads=8,
dropout_p=0.5):
            super(MixedGCNGAT, self).__init__()

            # First GCN layer
            self.gcn1 = GCNConv(input_dim, hidden_dim)
            self.bn1 = torch.nn.BatchNorm1d(hidden_dim)

            # Second GCN layer
            self.gcn2 = GCNConv(hidden_dim, hidden_dim)
            self.bn2 = torch.nn.BatchNorm1d(hidden_dim)
            # Third GCN layer
            self.gcn3 = GCNConv(hidden_dim, hidden_dim)
            self.bn3 = torch.nn.BatchNorm1d(hidden_dim)

            # First GAT layer
            self.gat1 = GATConv(hidden_dim, hidden_dim // gat_heads,
heads=gat_heads, dropout=dropout_p)
            self.bn4 = torch.nn.BatchNorm1d(hidden_dim)

            # Second GAT layer
            self.gat2 = GATConv(hidden_dim, hidden_dim // gat_heads,
heads=gat_heads, dropout=dropout_p)
            self.bn5 = torch.nn.BatchNorm1d(hidden_dim)

            # Output layer
            self.gat3 = GATConv(hidden_dim, output_dim, heads=1, concat=False,
dropout=dropout_p)

            # Activation function and Dropout
            self.relu = torch.nn.LeakyReLU()
            self.dropout = torch.nn.Dropout(p=dropout_p)

            # Residual connections
            self.residual1 = torch.nn.Linear(input_dim, hidden_dim)  # Modify
here
            self.residual2 = torch.nn.Linear(hidden_dim, hidden_dim)
            self.residual3 = torch.nn.Linear(hidden_dim, hidden_dim)

        def forward(self, data):
            x, edge_index = data.x, data.edge_index

            # First GCN layer + Residual connection
            residual = x  # x's shape is (num_nodes, input_dim)
            x = self.gcn1(x, edge_index)  # x's shape becomes (num_nodes,
hidden_dim)
            x = self.bn1(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + self.residual1(residual)  # residual1(residual) output shape is
(num_nodes, hidden_dim)

            # Second GCN layer + Residual connection
            residual = x
            x = self.gcn2(x, edge_index)
            x = self.bn2(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + self.residual2(residual)

            # Third GCN layer + Residual connection
            residual = x
            x = self.gcn3(x, edge_index)
            x = self.bn3(x)
            x = self.relu(x)
            x = self.dropout(x)
            x = x + self.residual3(residual)

            # First GAT layer
            x = self.gat1(x, edge_index)
            x = self.bn4(x)
            x = self.relu(x)
            x = self.dropout(x)

            # Second GAT layer
            x = self.gat2(x, edge_index)
            x = self.bn5(x)
            x = self.relu(x)
            x = self.dropout(x)

            # Output layer
            x = self.gat3(x, edge_index)

            return x


    # 5. Define the function to create a graph with self-loop edges
    def create_graph(features, targets, k=5):
        """
        Create a graph dynamically using k-nearest neighbors based on node features.

        Parameters:
        - features: numpy array, node feature matrix.
        - targets: numpy array, target variable array.
        - k: int, number of neighbors.

        Returns:
        - data_graph: torch_geometric.data.Data, constructed graph data object.
```

```
        """
        num_nodes = features.shape[0]
        nbrs = NearestNeighbors(n_neighbors=k+1, algorithm='ball_tree').fit(features)
        distances, indices = nbrs.kneighbors(features)

        # Initialize edge list
        edge_index = []
        edge_attr = []

        for i in range(num_nodes):
            for j in range(1, k+1):    # indices[i][0] is the node itself, skip it
                neighbor = indices[i][j]
                edge_index.append([i, neighbor])
                edge_attr.append(distances[i][j])

        edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()
        edge_attr = torch.tensor(edge_attr, dtype=torch.float)

        x = torch.tensor(features, dtype=torch.float)
        y = torch.tensor(targets, dtype=torch.float).unsqueeze(1)

        data_graph = Data(x=x, edge_index=edge_index, y=y)

        return data_graph


    # 6. Initialize error results dictionary
    error_results = {
        'Sample Size': [],
        'Neighbor Distance': [],
        'MSE': [],
        'RMSE': [],
        'MAE': []
    }


    # 7. Training and validation loop
    # Define different k values
    k_values = [2, 4, 6, 8, 10]

    # Initialize error results dictionary
    error_results = {
        'Neighbor Distance (k)': [],
        'MSE': [],
        'RMSE': [],
        'MAE': []
    }

    # Split the dataset into train and test (outside the loop)
    num_nodes = features.shape[0]
    indices = np.arange(num_nodes)
    train_indices,     test_indices     =     train_test_split(indices,     test_size=0.2,
random_state=42)
    train_mask = torch.zeros(num_nodes, dtype=torch.bool)
    test_mask = torch.zeros(num_nodes, dtype=torch.bool)
    train_mask[train_indices] = True
    test_mask[test_indices] = True


    for k in k_values:
        print(f"Processing k={k}")

        # Create graph data object
        data_graph = create_graph(features, targets, k=k)

        data_graph.train_mask = train_mask
        data_graph.test_mask = test_mask

        # Initialize model, loss function, and optimizer
        input_dim = features.shape[1]
        model = MixedGCNGAT(input_dim=input_dim)
        optimizer     =     torch.optim.Adam(model.parameters(),     lr=0.005,
weight_decay=5e-4)
        criterion = torch.nn.MSELoss()

        # Train the model
        # Define training parameters
        num_epochs = 300
        update_interval = 1  # Update graph structure every few epochs

        for epoch in range(num_epochs):
            if epoch % update_interval == 0:
                print(f"Epoch {epoch+1}: Updating graph structure...")
                data_graph  =  create_graph(features,  targets,  k=k)   #  Choose
appropriate k value

                # Split train and test sets (can remain unchanged or dynamically
update)
                num_nodes = data_graph.num_nodes
                indices = np.arange(num_nodes)
                train_indices, test_indices = train_test_split(indices, test_size=0.2,
random_state=42)
                train_mask = torch.zeros(num_nodes, dtype=torch.bool)
                test_mask = torch.zeros(num_nodes, dtype=torch.bool)
                train_mask[train_indices] = True
                test_mask[test_indices] = True
                data_graph.train_mask = train_mask
                data_graph.test_mask = test_mask

            # Train the model
            model.train()
            optimizer.zero_grad()
            out = model(data_graph)
            loss         =          criterion(out[data_graph.train_mask].squeeze(),
data_graph.y[data_graph.train_mask].squeeze())
            loss.backward()
            optimizer.step()

            if (epoch+1) % 50 == 0:
                print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.6f}')

        # Validate the model
        model.eval()
        with torch.no_grad():
            out = model(data_graph)
            predictions = out.squeeze().numpy()
            val_pred = out[data_graph.test_mask].squeeze().numpy()
            val_true = data_graph.y[data_graph.test_mask].squeeze().numpy()
            mse = mean_squared_error(val_true, val_pred)
            rmse = np.sqrt(mse)
            mae = mean_absolute_error(val_true, val_pred)

        # Add predicted values to DataFrame
        df[f'Predicted_F1_target_k_{k}'] = predictions

        # Store error results
        error_results['Neighbor Distance (k)'].append(k)
        error_results['MSE'].append(mse)
        error_results['RMSE'].append(rmse)
        error_results['MAE'].append(mae)
```

```
        print(f'Neighbor Distance (k={k}): MSE={mse:.4f}, RMSE={rmse:.4f}, 
MAE={mae:.4f}')

    # 17. Create error DataFrame
    error_df = pd.DataFrame(error_results)

    # 18. Plot error relationship using Seaborn
    sns.set(style="whitegrid", font_scale=1.2)

    plt.figure(figsize=(10, 6))
    sns.lineplot(data=error_df, x='Neighbor Distance (k)', y='MSE', marker='o', 
label='MSE')
    sns.lineplot(data=error_df, x='Neighbor Distance (k)', y='RMSE', marker='s', 
label='RMSE')
    sns.lineplot(data=error_df, x='Neighbor Distance (k)', y='MAE', marker='^', 
label='MAE')

    plt.title('Effect of Neighbor Distance (k) on Estimation Error', fontsize=16)
    plt.xlabel('Neighbor Distance (k)', fontsize=14)
    plt.ylabel('Error', fontsize=14)
    sns.despine()
    plt.legend(title='Error Metrics')
    plt.tight_layout()
    plt.savefig('问题 2/k_vs_error.png', dpi=300)
    plt.show()

    # Draw the graph structure for each k value
    for k in k_values:
        print(f"Drawing graph for k={k}")
        data_graph = create_graph(features, targets, k=k)

        # Convert PyG Data object to NetworkX graph
        G = to_networkx(data_graph, to_undirected=True, node_attrs=['x'])

        plt.figure(figsize=(12, 12))
        pos = nx.spring_layout(G, seed=42)
        nx.draw(G, pos, node_size=10, with_labels=False, edge_color='gray')
        plt.title(f'Graph Structure for k={k}')
        plt.tight_layout()
        plt.savefig(f'问题 2/graph_structure_k_{k}.png', dpi=300)
        plt.show()


    # For each k value, calculate residuals and plot them
    for k in k_values:

        residuals = df['F1_target'] - df[f'Predicted_F1_target_k_{k}']
        df['Residuals'] = residuals

        # Plot residual heatmap
        Z_residuals = residuals.values.reshape((grid_rows, grid_cols))
        plt.figure(figsize=(12, 10))
        sns.heatmap(Z_residuals, cmap='coolwarm', cbar=True,
                    xticklabels=50, yticklabels=50, center=0)
        plt.title('Residuals Heatmap', fontsize=16)
        plt.xlabel('X Coordinates (Meters)', fontsize=14)
        plt.ylabel('Y Coordinates (Meters)', fontsize=14)
        sns.despine()
        plt.tight_layout()
        plt.savefig(f'问题 2/Residuals_heatmap_{k}.png', dpi=300)
        plt.show()


        # Generate grid data for plotting
        Z_pred    =    df[f'Predicted_F1_target_k_{k}'].values.reshape((grid_rows,
grid_cols))

        # Get X and Y coordinates
        X = df['X'].values.reshape((grid_rows, grid_cols))
        Y = df['Y'].values.reshape((grid_rows, grid_cols))

        # 19. Plot contour plot of predicted values using Seaborn style
        plt.figure(figsize=(12, 10))
        contour = plt.contourf(X, Y, Z_pred, levels=100, cmap='viridis')
        plt.colorbar(contour, label='Predicted F1_target')
        plt.title('Predicted F1_target Contour Plot', fontsize=16)
        plt.xlabel('X Coordinates (Meters)', fontsize=14)
        plt.ylabel('Y Coordinates (Meters)', fontsize=14)
        sns.despine()
        plt.tight_layout()
        plt.savefig('问题 2/Predicted_F1_target_contour_seaborn_{k}.png', dpi=300)
        plt.show()

        # 20. Plot heatmap of predicted values with adjusted scale label density
        plt.figure(figsize=(12, 10))
        sns.heatmap(Z_pred, cmap='viridis', cbar=True,
                    xticklabels=50, yticklabels=50)
        plt.title('Predicted F1_target Heatmap', fontsize=16)
        plt.xlabel('X Coordinates (Meters)', fontsize=14)
        plt.ylabel('Y Coordinates (Meters)', fontsize=14)
        sns.despine()
        plt.tight_layout()
        plt.savefig(' 问 题 2/Predicted_F1_target_heatmap_seaborn_{k}.png',
dpi=300)
        plt.show()


    # Set the device
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f'Using device: {device}')

    # Assuming features and targets are already defined and loaded
    # Replace with actual data loading code
    # For example:
    # features = np.load('features.npy')    # Example
    # targets = np.load('targets.npy')       # Example
    # Make sure features is a [num_nodes, num_features] shaped numpy array
    # targets is a [num_nodes] shaped numpy array

    # Example data (replace with actual data as necessary)
    num_nodes = 10000    # Example number of nodes
    num_features = 5     # Example feature dimension
    features = np.random.rand(num_nodes, num_features).astype(np.float32)
    targets = np.random.rand(num_nodes).astype(np.float32)

    # Define create_graph function
    def create_graph(features, targets, k=5):
        """
        Dynamically create a graph based on k-nearest neighbors of node features.

        Parameters:
        - features: numpy array, node feature matrix.
        - targets: numpy array, target variable array.
        - k: int, number of neighbors.

        Returns:
        - data_graph: torch_geometric.data.Data, constructed graph data object.
        """
        num_nodes = features.shape[0]
        nbrs                =                NearestNeighbors(n_neighbors=k+1,
algorithm='ball_tree').fit(features)
        distances, indices = nbrs.kneighbors(features)
        # Initialize edge list
        edge_index = []
        edge_attr = []

        for i in range(num_nodes):
            for j in range(1, k+1):    # indices[i][0] is the node itself, skip it
                neighbor = indices[i][j]
                edge_index.append([i, neighbor])
                edge_attr.append(distances[i][j])

        edge_index = torch.tensor(edge_index, dtype=torch.long).t().contiguous()
        edge_attr = torch.tensor(edge_attr, dtype=torch.float)

        x = torch.tensor(features, dtype=torch.float)
        y = torch.tensor(targets, dtype=torch.float).unsqueeze(1)

        # Move data to device
        x = x.to(device)
        edge_index = edge_index.to(device)
        y = y.to(device)

        data_graph = Data(x=x, edge_index=edge_index, y=y)

        return data_graph

    # Define generate_graph_at_time function
    def generate_graph_at_time(t):
        """
        Generate the corresponding graph at time t. Here, different k values are treated as time points.
        """
        k_values = [2, 4, 6, 8, 10]
        k = k_values[t % len(k_values)]    # Cycle through k values
        data_graph = create_graph(features, targets, k=k)
        return data_graph, k

    # Create 3D plot
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')

    time_steps = [0, 1, 2, 3, 4]    # Corresponding to different time points

    for idx, t in enumerate(time_steps):
        data_graph, k = generate_graph_at_time(t)

        # Sample the graph to reduce the number of nodes
        num_samples = 1000    # Adjust as needed
        if data_graph.num_nodes < num_samples:
            sampled_indices = np.random.choice(data_graph.num_nodes,
data_graph.num_nodes, replace=False)
        else:
            sampled_indices = np.random.choice(data_graph.num_nodes,
num_samples, replace=False)
        sampled_indices = torch.tensor(sampled_indices,
dtype=torch.long).to(device)

        # Extract the subgraph
        # Move sampled_indices and edge_index to CPU for subgraph extraction
        sampled_indices_cpu = sampled_indices.to('cpu')
        data_graph_cpu = data_graph.to('cpu')

        edge_index_sub, _ = subgraph(sampled_indices_cpu,
data_graph_cpu.edge_index, relabel_nodes=True)
        x_sub = data_graph_cpu.x[sampled_indices_cpu]
        subgraph_data = Data(x=x_sub, edge_index=edge_index_sub).to('cpu')

        # Convert the subgraph to a NetworkX graph
        G = to_networkx(subgraph_data, to_undirected=True)

        # Get the 3D coordinates of nodes
        pos = nx.spring_layout(G, dim=3, seed=42)
        xs = [pos[n][0] for n in G.nodes()]
        ys = [pos[n][1] for n in G.nodes()]
        zs = [t] * len(G.nodes())    # Use time as the Z-axis coordinate

        # Plot the scatter plot
        ax.scatter(xs, ys, zs, s=20, alpha=0.6, label=f'k={k}')

    # Set axis labels
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Time')

    # Set title
    plt.title('Dynamic Graph in 3D Space-Time')

    # Set legend, avoid duplicate labels
```

```
handles, labels = ax.get_legend_handles_labels()
unique_labels = dict(zip(labels, handles))
ax.legend(unique_labels.values(), unique_labels.keys())

# Optimize layout
plt.tight_layout()

# Save and show the plot
plt.savefig('问题 2/dynamic_graph_3d.png', dpi=300)
plt.show()
```

3. problem 3 codes

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pykrige.ok import OrdinaryKriging
from scipy.spatial import distance
from sklearn.metrics import mean_squared_error, mean_absolute_error
import matplotlib

# Set TkAgg backend
matplotlib.use("TkAgg")


# Load data
def load_data():
    target_variable = np.load("F1_target_variable.npy")
    collaborative_1 = np.load("F1_collaborative_variable1.npy")
    collaborative_4 = np.load("F1_collaborative_variable4.npy")
    return target_variable, collaborative_1, collaborative_4


# Random uniform sampling
def random_sampling(data, sampling_ratio):
    np.random.seed(42)
    total_points = data.size
    sample_size = int(total_points * sampling_ratio)
    sampled_indices = np.random.choice(total_points, sample_size, 
replace=False)
    sampled_values = data.flatten()[sampled_indices]
    sampled_coords = np.array(np.unravel_index(sampled_indices, 
data.shape)).T
    return sampled_coords, sampled_values
```

```
# D-KNN Interpolation
def D_KNN_interpolation(coords, values, grid_x, grid_y, variogram_model="linear"):
    # Ensure coordinates are of type float
    coords = coords.astype(np.float64)
    grid_x = grid_x.astype(np.float64)
    grid_y = grid_y.astype(np.float64)

    OK = OrdinaryKriging(
        coords[:, 0],
        coords[:, 1],
        values,
        variogram_model=variogram_model,
        verbose=False,
        enable_plotting=False,
    )
    z, _ = OK.execute("grid", grid_x, grid_y)
    return z


# Inverse Distance Weighting (IDW) Interpolation
def idw_interpolation(coords, values, grid_x, grid_y, p=2):
    grid_points = np.array(np.meshgrid(grid_x, grid_y)).T.reshape(-1, 2)
    z = np.zeros(grid_points.shape[0])
    for i, point in enumerate(grid_points):
        distances = distance.cdist([point], coords).flatten()
        weights = 1 / np.power(distances + 1e-12, p)    # Avoid division by zero
        z[i] = np.sum(weights * values) / np.sum(weights)
    return z.reshape(grid_x.size, grid_y.size)


# Error evaluation
def evaluate_performance(true_values, predicted_values):
    mse = mean_squared_error(true_values, predicted_values)
    mae = mean_absolute_error(true_values, predicted_values)
    max_error = np.max(np.abs(true_values - predicted_values))
    return mse, mae, max_error


# Main function
def main():
    # Load data
    target_variable, collaborative_1, collaborative_4 = load_data()
```

```
        grid_x = np.arange(target_variable.shape[0])
        grid_y = np.arange(target_variable.shape[1])

        # Define sampling ratios
        sampling_ratios = [0.01, 0.05, 0.1]

        # Store results
        results = []

        for sampling_ratio in sampling_ratios:
            print(f"Sampling ratio: {sampling_ratio * 100}%")

            # Random sampling
            sampled_coords, sampled_values = random_sampling(target_variable,
sampling_ratio)

            # D-KNN interpolation
            D_KNN_result          =          D_KNN_interpolation(sampled_coords,
sampled_values, grid_x, grid_y)

            # IDW interpolation
            idw_result   =   idw_interpolation(sampled_coords,   sampled_values,
grid_x, grid_y)

            # True values (excluding sampled points)
            mask = np.ones(target_variable.shape, dtype=bool)
            mask[sampled_coords[:, 0], sampled_coords[:, 1]] = False
            true_values = target_variable[mask]
            predicted_D_KNN = D_KNN_result[mask]
            predicted_idw = idw_result[mask]

            # Error calculation
            D_KNN_mse,        D_KNN_mae,        D_KNN_max_error        =
evaluate_performance(true_values, predicted_D_KNN)
            idw_mse,          idw_mae,          idw_max_error          =
evaluate_performance(true_values, predicted_idw)

            # Record results
            results.append(
                [sampling_ratio,    "D_KNN",    D_KNN_mse,    D_KNN_mae,
D_KNN_max_error]
            )
            results.append([sampling_ratio,    "IDW",    idw_mse,    idw_mae,
idw_max_error])
```

```
                print(f"D_KNN MSE: {D_KNN_mse:.4f}, MAE: {D_KNN_mae:.4f},
Max Error: {D_KNN_max_error:.4f}")
                print(f"IDW MSE: {idw_mse:.4f}, MAE: {idw_mae:.4f}, Max Error:
{idw_max_error:.4f}")

        # Convert to DataFrame
        results_df = pd.DataFrame(results, columns=["Sampling Ratio", "Method",
"MSE", "MAE", "Max Error"])

        # Plot comparison of metrics
        metrics = ["MSE", "MAE", "Max Error"]
        fig, axes = plt.subplots(1, 3, figsize=(18, 5))

        for idx, metric in enumerate(metrics):
            ax = axes[idx]
            for method in ["D_KNN", "IDW"]:
                method_results = results_df[results_df["Method"] == method]
                ax.plot(
                    method_results["Sampling Ratio"] * 100,
                    method_results[metric],
                    marker='*',
                    label=method
                )
            ax.set_xlabel("Sampling Ratio (%)", fontsize=12)
            ax.set_ylabel(metric, fontsize=12)
            ax.set_title(f"{metric} Comparison", fontsize=14)
            ax.legend()
            ax.grid(True)

        plt.tight_layout()
        plt.show()

    if __name__ == "__main__":
        main()
```

4. problem 4 codes

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns
```

```
    # Set matplotlib to use the TkAgg backend for display
    matplotlib.use('TkAgg')
    sns.set_theme(style="whitegrid")

    # Define the data reading function for collaborative variables
    def read_collaborative_data(file_path):
        """Read data from a collaborative variable file and convert it to a 266x266
matrix format."""
        with open(file_path, 'r') as file:
            lines = file.readlines()
            data = []
            for line in lines[6:]:    # Start reading numerical data from line 7
                if line.strip():    # Skip empty lines
                    try:
                        row = list(map(float, line.strip().split()))
                        data.extend(row)
                    except ValueError as e:
                        print(f"Parsing error, skipping line: {line.strip()} - Error:
{e}")
                        continue

            # Check data length to ensure it matches a 266x266 matrix
            if len(data) == 266 * 266:
                matrix_data = np.array(data).reshape(266, 266)
                return matrix_data
            else:
                print(f"Data length mismatch: {len(data)}. Expected 70756
elements (266x266).")
                return np.array([])

    # Folder path for Attachment 2
    folder_path = '问题 4'

    # List of file names for collaborative variables
    collaborative_file_names = ['F2_collaborative_variable1.txt',
'F2_collaborative_variable2.txt',
                            'F2_collaborative_variable3.txt',
'F2_collaborative_variable4.txt']

    # Load target variable from Excel file
    target_file_path = os.path.join(folder_path, 'F2_target_variable.xlsx')
    try:
        target_df = pd.read_excel(target_file_path)
        print("Target variable data loaded successfully.")
    except FileNotFoundError:
        print(f"{target_file_path} not found. Please check the path.")
        exit()

    # Print the first few rows to verify
    print(target_df.head())

    # Load collaborative variables
    data_dict_problem4 = {}
    for file_name in collaborative_file_names:
        file_path = os.path.join(folder_path, file_name)
        data = read_collaborative_data(file_path)
        if data.size > 0:
            data_dict_problem4[file_name] = data   # Store processed data in the
dictionary
            # Save each matrix as a .npy file with a descriptive name
            np.save(f'{file_name.split(".")[0]}_problem4.npy', data)
            print(f"Saved {file_name} as {file_name.split('.')[0]}_problem4.npy")
        else:
            print(f"{file_name} failed to load, skipping this file.")

    # Use Seaborn to plot heatmaps of the collaborative data
    if all(data.shape == (266, 266) for data in data_dict_problem4.values()):
        plt.figure(figsize=(12, 12))
        for i, (file_name, matrix_data) in enumerate(data_dict_problem4.items(), 1):
            plt.subplot(2, 2, i)
            # Use Seaborn to plot the heatmap
            sns.heatmap(matrix_data, cmap='viridis', cbar=True)
            plt.title(file_name)
            plt.axis('off')   # Hide the axis

        plt.tight_layout()
        plt.show()
    else:
        print("Data loading failed. Please check the file contents and format.")

    import numpy as np
    import pandas as pd
    import matplotlib.pyplot as plt
    from pykrige.ok import OrdinaryKriging
    import os
    from scipy.ndimage import zoom
    import matplotlib
    import seaborn as sns
```

```
sns.set_theme(style="whitegrid")

# Set matplotlib to use the TkAgg backend for display
matplotlib.use('TkAgg')

# Folder path for Attachment 2
folder_path = 'I:/竞赛/数维杯/1 数维国际 B/代码/Attachment 2'

# Load target variable from Excel file
file_path = os.path.join(folder_path, 'F2_target_variable.xlsx')

# Load target data
try:
    target_data = pd.read_excel(file_path)
    print("File loaded successfully.")
except FileNotFoundError:
    print("File not found. Please check the path.")
    exit()

# Extract coordinates and target values
target_coords = target_data[['X-Coordinate', 'Y-Coordinate']].values
target_values = target_data['Target Property'].values

# Downsample data by randomly selecting 10% of points for interpolation
sample_indices = np.random.choice(len(target_coords), size=int(0.1 * len(target_coords)), replace=False)
sampled_coords = target_coords[sample_indices]
sampled_values = target_values[sample_indices]

# Scale coordinates to reduce memory usage
sampled_coords_scaled = sampled_coords / 1000.0  # Scale down coordinates

# Set a low-resolution grid (20x20)
grid_x = np.linspace(sampled_coords_scaled[:, 0].min(), sampled_coords_scaled[:, 0].max(), 20)
grid_y = np.linspace(sampled_coords_scaled[:, 1].min(), sampled_coords_scaled[:, 1].max(), 20)
grid_x, grid_y = np.meshgrid(grid_x, grid_y)

# Set up dynamic KNN interpolation
variogram_model = 'spherical'  # Using spherical model

# Perform dynamic KNN interpolation
OK = OrdinaryKriging(
        sampled_coords_scaled[:, 0], sampled_coords_scaled[:, 1], sampled_values,
        variogram_model=variogram_model, verbose=False, enable_plotting=False
    )

    # Execute interpolation with high resolution and downsample
    high_res_z, ss = OK.execute("grid", grid_x.flatten(), grid_y.flatten())  # Create
high-res interpolation
    high_res_z = high_res_z.reshape((400, 400))  # Assuming high_res_z was
generated at 400x400

    # Downsample to match the grid size 20x20
    z_downsampled = zoom(high_res_z, (20/400, 20/400))

    # Plot the contour map with new colormap
    plt.figure(figsize=(10, 8))
    contour = plt.contourf(grid_x * 1000, grid_y * 1000, z_downsampled, levels=20,
cmap="cividis", alpha=0.9)  # Scale back to original coordinates
    cbar = plt.colorbar(contour)
    cbar.set_label("Estimated Target Property", fontsize=12, labelpad=10)
    plt.contour(grid_x * 1000, grid_y * 1000, z_downsampled, levels=20,
colors='white', linewidths=0.7, alpha=0.6)
    plt.xlabel("X Coordinate", fontsize=12)
    plt.ylabel("Y Coordinate", fontsize=12)
    plt.title("Kriging Interpolation of Target Variable (Low Resolution with
Sampling)", fontsize=14)
    plt.tick_params(labelsize=10)
    plt.tight_layout()
    plt.show()

    # Set Seaborn style
    sns.set_theme(style="whitegrid")

    # Load data section remains the same
    # Set file path

    # Load target data
    try:
        target_data = pd.read_excel(file_path)
        print("File loaded successfully.")
    except FileNotFoundError:
        print("File not found. Please check the path.")
        exit()

    # Extract coordinates and target values
    target_coords = target_data[['X-Coordinate', 'Y-Coordinate']].values
    target_values = target_data['Target Property'].values

    # Downsample data by randomly selecting 10% of points for interpolation
    sample_indices = np.random.choice(len(target_coords), size=int(0.1 *
len(target_coords)), replace=False)
    sampled_coords = target_coords[sample_indices]
    sampled_values = target_values[sample_indices]

    # Scale coordinates to reduce memory usage
    sampled_coords_scaled = sampled_coords / 1000.0    # Scale down coordinates

    # Set a low-resolution grid (20x20)
    grid_x = np.linspace(sampled_coords_scaled[:, 0].min(),
sampled_coords_scaled[:, 0].max(), 20)
    grid_y = np.linspace(sampled_coords_scaled[:, 1].min(),
sampled_coords_scaled[:, 1].max(), 20)
    grid_x, grid_y = np.meshgrid(grid_x, grid_y)

    # Set up dynamic KNN interpolation
    variogram_model = 'spherical'    # Using spherical model

    # Perform dynamic KNN interpolation
    OK = OrdinaryKriging(
        sampled_coords_scaled[:, 0], sampled_coords_scaled[:, 1], sampled_values,
        variogram_model=variogram_model, verbose=False, enable_plotting=False
    )

    # Execute interpolation with high resolution and downsample
    # For demonstration, we'll directly interpolate on the low-resolution grid
    z_downsampled, ss = OK.execute("grid", grid_x[0, :], grid_y[:, 0])

    # Plot the contour map using Matplotlib functions
    plt.figure(figsize=(10, 8))

    # Plot filled contour map
    contourf = plt.contourf(grid_x * 1000, grid_y * 1000, z_downsampled, levels=20,
cmap="plasma", alpha=0.9)

    # Add color bar
    cbar = plt.colorbar(contourf)
    cbar.set_label("Estimated Target Property", fontsize=12, labelpad=10)

    # Plot contour lines
    contour = plt.contour(grid_x * 1000, grid_y * 1000, z_downsampled, levels=20,
colors='white', linewidths=0.7, alpha=0.6)

    # Set axis labels and title
    plt.xlabel("X Coordinate", fontsize=12)
    plt.ylabel("Y Coordinate", fontsize=12)
    plt.title("Dynamic KNN of Target Variable", fontsize=14)
    plt.tick_params(labelsize=10)
    plt.tight_layout()
    plt.show()
```