Homework Assignment 2

Yihan Wang

November 5, 2024

1 Problem 1: Feature-to-Graph Conversion

The objective is to construct a near-neighbor graph $G(V, E)$ from feature data X in a d-dimensional metric space, where the mapping from X to V is one-to-one, and E preserves spatial proximity to some extent.

1(a) Dataset Gathering and Preparation

In this part, we gathered two feature datasets to use in the construction of a near-neighbor graph:

- **Dataset 1:** The *Iris dataset*, containing 150 data points, each with 4-dimensional features representing sepal and petal characteristics of various flower species.
- **Dataset 2:** A *randomly generated dataset* with 1000 data points in a 5-dimensional feature space, created to meet the requirement of having at least two datasets.

The following MATLAB code snippet was used to load and generate these datasets:

Listing 1: Loading and Generating Datasets

```
% Dataset 1: Load Iris Data (150 points, 4D space)
load fisheriris;
X1 = meas;  % Iris feature data

% Dataset 2: Generate 1000 random points in 5D space
n = 1000; d = 5;
X2 = randn(n, d);  % Randomly generated dataset
```

Figures 1 and 2 show the initial transformation of these datasets into k-Nearest Neighbor (k-NN) graphs with $k = 5$. These figures validate that both datasets have been appropriately processed into k-NN graphs, setting up the groundwork for the remaining parts of the analysis.

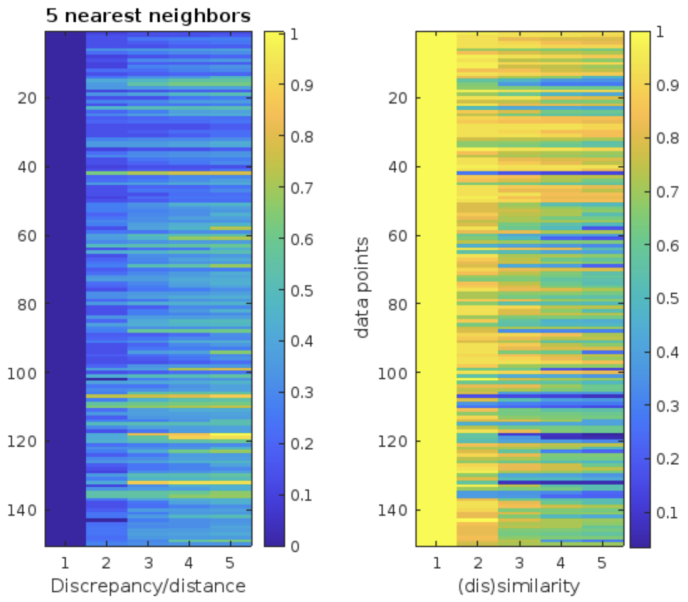


Figure 1: 5 nearest neighbors for Iris dataset (k-NN graph)

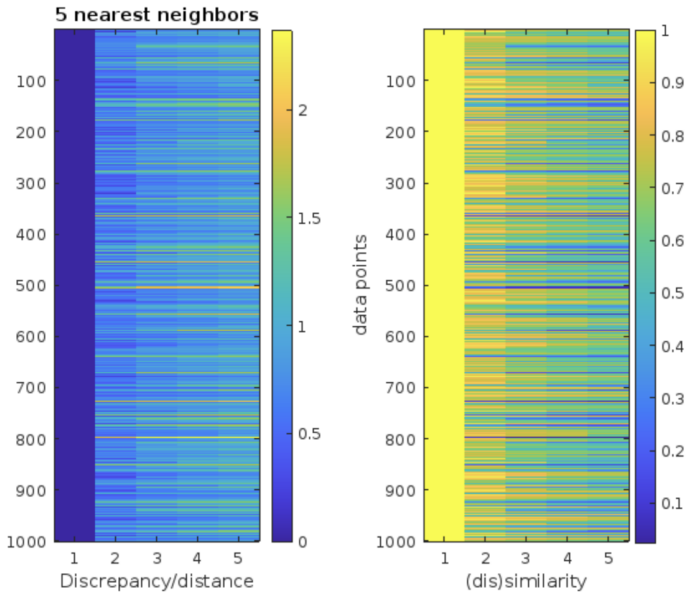


Figure 2: 5 nearest neighbors for Random dataset (k-NN graph)

1(b) Constructing a k-NN Graph

In this part, we constructed a k-Nearest Neighbor (k-NN) graph for both datasets using a specified number of nearest neighbors, k. The k-NN graph was constructed to ensure each point in the dataset is connected to its k closest neighbors, creating a sparse graph that preserves local spatial relationships.

Listing 2: Constructing k-NN Graphs

run demo_KNN_analysis.m

```
% Part (a): Load both datasets
% Dataset 1: Load Iris Data (150 points, 4D space)
load fisheriris;
X1 = meas;  % Iris feature data

% Dataset 2: Generate 1000 random points in 5D space
n = 1000; d = 5;
X2 = randn(n, d);  % Randomly generated dataset

fprintf('Loaded both datasets: Iris and Random Data.\n');

% Part (b): Construct a k-NN Graph for both datasets

% Step 1: Set the number of nearest neighbors
k = input('Specify the number of nearest neighbors k: ');
fprintf('k = %d selected\n', k);  % Confirmation message

% Step 2: Compute pairwise distances for each dataset
fprintf('Computing pairwise distances...\n');
D1 = pdist2(X1, X1);  % Iris dataset
D2 = pdist2(X2, X2);  % Random dataset

% Step 3: Sort distances and get indices of k-nearest neighbors
fprintf('Sorting distances to find k-nearest
neighbors...\n');
[~, idx1] = sort(D1, 2);  % For Iris dataset
[~, idx2] = sort(D2, 2);  % For Random dataset

% Step 4: Initialize adjacency matrices and construct
symmetric k-NN graphs
fprintf('Constructing adjacency matrices...\n');
A_knn1 = zeros(size(D1));  % For Iris dataset
A_knn2 = zeros(size(D2));  % For Random dataset

% Construct symmetric k-NN graph for Iris dataset
for i = 1:size(D1, 1)
    A_knn1(i, idx1(i, 2:k+1)) = 1;  % Exclude self-loop
end
```

```
A_knn1 = max(A_knn1, A_knn1'); % Make the matrix symmetric
fprintf('k-NN graph constructed for Iris dataset.\n');

% Construct symmetric k-NN graph for Random dataset
for i = 1:size(D2, 1)
    A_knn2(i, idx2(i, 2:k+1)) = 1;
end
A_knn2 = max(A_knn2, A_knn2'); % Make the matrix symmetric
fprintf('k-NN graph constructed for Random dataset.\n');

% Step 5: Visualize the k-NN graphs (Optional)
G_knn1 = graph(A_knn1);
fprintf('Visualizing k-NN graph for Iris dataset...\n');
figure; plot(G_knn1);
title('k-NN Graph for Iris Dataset');

G_knn2 = graph(A_knn2);
fprintf('Visualizing k-NN graph for Random
dataset...\n');
figure; plot(G_knn2);
title('k-NN Graph for Random Dataset');
```

Figures 3 and 4 below illustrate the k-NN graph for each dataset.

1(c) Converting Pairwise Distances to Pairwise Similarity Scores

In this part, we converted the pairwise distances into pairwise similarity scores for both datasets using a Gaussian (RBF) kernel. This transformation allows us to interpret the distance data as similarity scores, where closer points have higher similarity values. The Gaussian kernel formula used is:

$$S_{ij} = \exp\left(-\frac{D_{ij}^2}{2\sigma^2}\right)$$

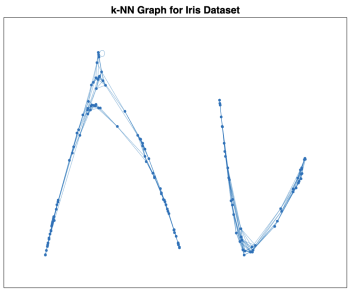
where D_{ij} is the distance between points i and j, and σ is a kernel width parameter that controls the similarity decay with distance.

The MATLAB code below shows the implementation of the Gaussian kernel to compute similarity scores for both datasets:

Listing 3: Converting Pairwise Distances to Similarity Scores

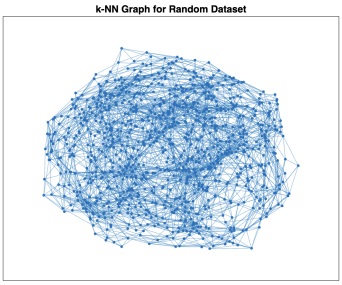
```
% Part (c): Convert pairwise distances to pairwise
similarity scores

% Parameters for Gaussian (RBF) kernel
sigma = 1.0; % Kernel width parameter (can be adjusted)
```

see knn_graph_construction.m line 32-38



run demo_KNN_analysis.m

Figure 3: k-NN Graph for Iris Dataset



run demo_KNN_analysis.m

Figure 4: k-NN Graph for Random Dataset

Figure 5: Side-by-side view of k-NN Graphs for Iris and Random Datasets.

```
% Step 1: Convert distances to similarities for Iris dataset
S_knn1 = exp(-D1.^2 / (2 * sigma^2)); % Gaussian
kernel for Iris dataset
S_knn1 = S_knn1 .* A_knn1; % Apply k-NN mask to retain
only nearest neighbors

% Step 2: Convert distances to similarities for Random dataset
S_knn2 = exp(-D2.^2 / (2 * sigma^2)); % Gaussian
kernel for Random dataset
S_knn2 = S_knn2 .* A_knn2; % Apply k-NN mask to retain
only nearest neighbors

% Visualize the similarity matrices (Optional)
figure;
imagesc(S_knn1);
colorbar;
title('Pairwise Similarity Scores for Iris Dataset
(Gaussian Kernel)');
```

```
figure;
imagesc(S_knn2);
colorbar;
title('Pairwise Similarity Scores for Random Dataset
(Gaussian Kernel)');
```

Figures 6 and 7 below show the resulting similarity matrices for the Iris and Random datasets, respectively. In the Iris dataset, clusters are visible, indicating high similarity among groups of points. In the Random dataset, similarity scores are more uniformly low, as expected due to random distribution.

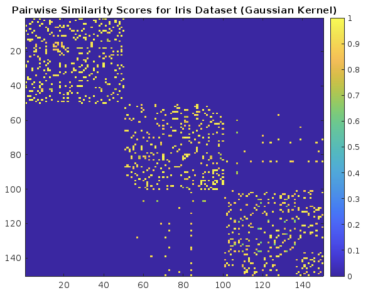


Figure 6: Pairwise Similarity Scores for Iris Dataset (Gaussian Kernel)

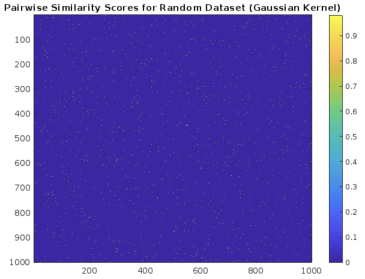


Figure 7: Pairwise Similarity Scores for Random Dataset (Gaussian Kernel)

1(d) Degree Distribution and Variation with Different k Values

trivial that d_out = k, so examing d_in

In this part, we analyze the degree distribution of the k-NN graphs created from the Iris and Random datasets for different values of k (5, 10, and 20). As k increases, each node has more neighbors, resulting in a shift in the degree distribution. This section examines how the degree distribution changes with different values of k.

The MATLAB code below shows the process of constructing k-NN graphs with different k values and plotting the resulting degree distributions.

Listing 4: Degree Distribution Analysis for Different Values of k

```
% Part (d): Analyze Degree Distribution for Different
Values of k

% Define values of k to test
k_values = [5, 10, 20];  % Example k values to observe
changes in degree distribution

% Loop over each k value to compute and plot degree distribution
for k_idx = 1:length(k_values)
    k = k_values(k_idx);
    fprintf('Analyzing degree distribution for k = %d\n', k);

    % Step 1: Construct k-NN graph for Iris dataset with current k
    [~, idx1] = sort(D1, 2);  % For Iris dataset
    A_knn1 = zeros(size(D1));
    for i = 1:size(D1, 1)
        A_knn1(i, idx1(i, 2:k+1)) = 1;  % Exclude self-loop
    end
    A_knn1 = max(A_knn1, A_knn1');  % Make symmetric
    G_knn1 = graph(A_knn1);

    % Step 2: Construct k-NN graph for Random dataset with current k
    [~, idx2] = sort(D2, 2);  % For Random dataset
    A_knn2 = zeros(size(D2));
    for i = 1:size(D2, 1)
        A_knn2(i, idx2(i, 2:k+1)) = 1;
    end
    A_knn2 = max(A_knn2, A_knn2');  % Make symmetric
    G_knn2 = graph(A_knn2);

    % Step 3: Calculate degree distribution for each dataset
    degrees_iris = degree(G_knn1);
    degrees_random = degree(G_knn2);

    % Step 4: Plot degree distribution for Iris dataset
    figure;
    histogram(degrees_iris, 'Normalization', 'pdf');
    title(sprintf('Degree Distribution for Iris Dataset
    (k = %d)', k));
    xlabel('Degree');
    ylabel('Probability Density');

    % Step 5: Plot degree distribution for Random dataset
    figure;
```

Construct A using demo_KNN_analysis, or just go by trivial symmertrizing bring in more neighbors

```
    histogram(degrees_random, 'Normalization', 'pdf');
    title(sprintf('Degree Distribution for Random
----Dataset (k=%d)', k));
    xlabel('Degree');
    ylabel('Probability Density');
end
```

Figures 14 to 19 below show the degree distributions for both the Iris and Random datasets at different values of k.

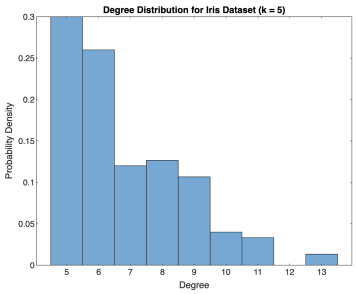


Figure 8: Degree Distribution for Iris Dataset (k = 5)

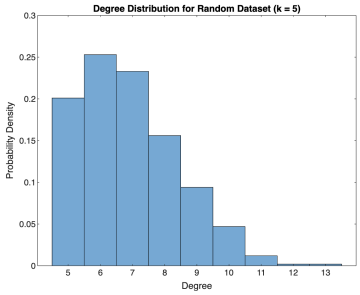


Figure 9: Degree Distribution for Random Dataset (k = 5)

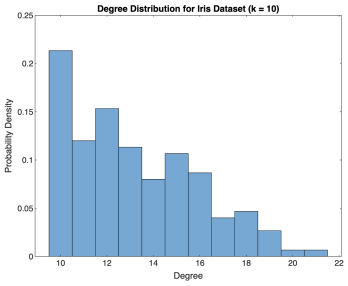


Figure 10: Degree Distribution for Iris Dataset (k = 10)

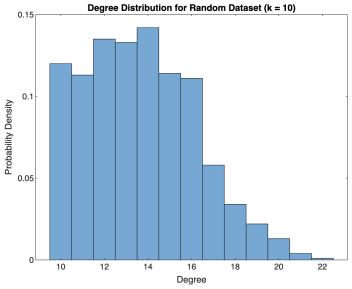


Figure 11: Degree Distribution for Random Dataset (k = 10)

Part (e): Observing Distributions of Additional Measures

In this section, we analyze the clustering coefficient distribution for the Iris and Random datasets at different values of k (e.g., $k = 5$, $k = 10$, and $k = 20$). The clustering coefficient provides insights into the degree to which nodes in the graph tend to cluster together.

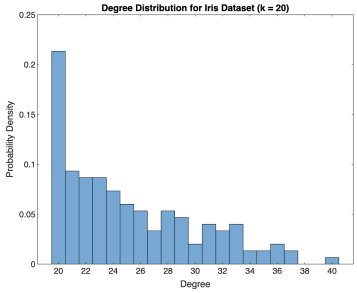


Figure 12: Degree Distribution for Iris Dataset (k = 20)

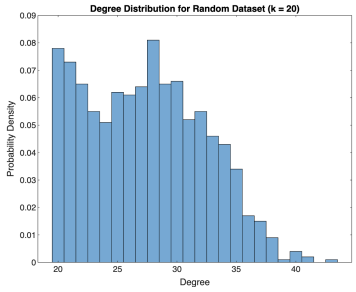


Figure 13: Degree Distribution for Random Dataset (k = 20)

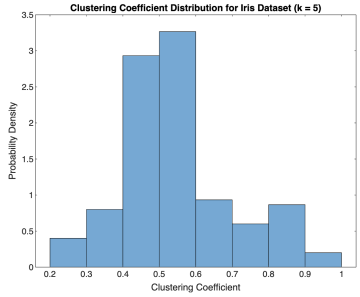


Figure 14: Clustering Coefficient Distribution for Iris Dataset ($k = 5$)

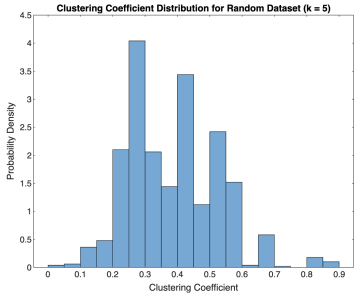


Figure 15: Clustering Coefficient Distribution for Random Dataset ($k = 5$)

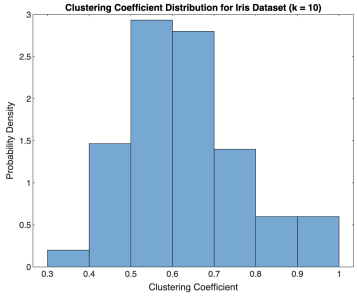


Figure 16: Clustering Coefficient Distribution for Iris Dataset ($k = 10$)

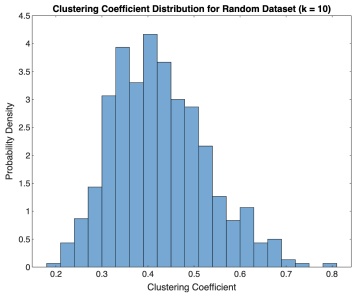


Figure 17: Clustering Coefficient Distribution for Random Dataset ($k = 10$)

1(f) Converting Categorical Data to Numerical Format

In this section, we convert categorical data into numerical format using one-hot encoding. One-hot encoding is a common method for representing categorical

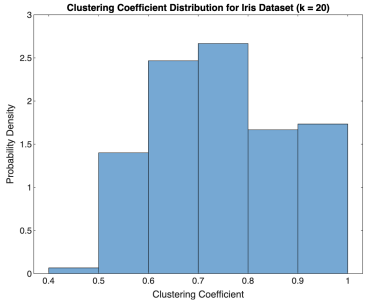


Figure 18: Clustering Coefficient Distribution for Iris Dataset ($k = 20$)

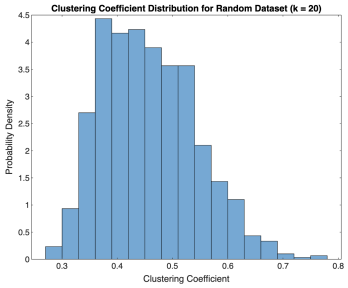


Figure 19: Clustering Coefficient Distribution for Random Dataset ($k = 20$)

variables, where each category is represented by a unique binary vector. Additionally, we show how to perform label encoding, which converts categories into unique integer values.

The MATLAB code below demonstrates both one-hot encoding and label encoding for a sample categorical dataset.

Listing 5: Converting Categorical Data to Numerical Format

```
% Example Part (f): Convert Categorical Data into Numerical Format

% Sample categorical data (replace this with actual
feature data if available)
% Here, assume we have a simple array of categorical
data for demonstration.
categorical_data = ["red", "blue", "green", "blue",
"green", "red", "blue"];

% Step 1: Convert categorical data to categorical type in MATLAB
categorical_data = categorical(categorical_data);

% Step 2: One-Hot Encoding
% Find the unique categories and create a one-hot
encoded matrix
categories = categories(categorical_data);
numCategories = numel(categories);
numDataPoints = numel(categorical_data);
oneHotEncoded = zeros(numDataPoints, numCategories);

% Assign 1s in the appropriate locations to create the
one-hot encoded format
for i = 1:numDataPoints
    categoryIdx = find(categories ==
```

```
    categorical_data(i));
    oneHotEncoded(i, categoryIdx) = 1;
end

% Display the resulting one-hot encoded matrix
disp('One-Hot Encoded Matrix:');
disp(oneHotEncoded);
disp('Categories (in column order):');
disp(categories);

% Optional: If you need a label-encoded format (integer
encoding)
labelEncoded = double(categorical_data);

% Display the label encoded data
disp('Label Encoded Data:');
disp(labelEncoded);
```

In the code above:

- **One-Hot Encoding**: The categorical data is first converted to MATLAB's 'categorical' type. A one-hot encoded matrix is then created by assigning a binary vector to each category.
- **Label Encoding**: As an alternative, label encoding converts each category to a unique integer value.

Problem 2: Vertex-to-Vector Encoding and Embedding

The objective is to map vertices in a graph $G(V, E)$, weighted or unweighted, into a vector set X in a metric space to approximate adjacency preservation in pairwise distances.

(a) Graph Selection and Construction

We use two graphs based on the Iris dataset: use demo_dim_reduction.m

- **Unweighted Graph**: Created as a 5-Nearest Neighbors (k-NN) graph.
- **Weighted Graph**: Using Gaussian similarity on pairwise distances.

The following MATLAB code loads the dataset and constructs both graphs:

Listing 6: Constructing Unweighted and Weighted Graphs

```
% Load Iris dataset for graph construction
get_iris_data;
X = meas;

% Unweighted k-NN graph with k = 5
```

```
k = 5;
D_unweighted = pdist2(X, X);
A_unweighted = zeros(size(D_unweighted));
[~, idx_unweighted] = sort(D_unweighted, 2);
for i = 1:size(D_unweighted, 1)
    A_unweighted(i, idx_unweighted(i, 2:k+1)) = 1;
end
A_unweighted = max(A_unweighted, A_unweighted'); %
Symmetric matrix

% Weighted graph using Gaussian similarity with sigma = 1
sigma = 1.0;
A_weighted = exp(-D_unweighted.^2 / (2 * sigma^2));
A_weighted(A_weighted < 0.1) = 0; % Sparsify graph
```

use demo_dim_reduction.m

(b) Spectral Embedding with Normalized Laplacian

We calculate a spectral embedding for each graph using the normalized Laplacian matrix with an embedding dimension $d = 3$.

Listing 7: Spectral Embedding with Normalized Laplacian

```
% Spectral embedding for unweighted graph
d = 3;
sigma_eigs = 1e-5; % Stability shift for eigs
D_unweighted_deg = diag(sum(A_unweighted, 2));
L_unweighted = D_unweighted_deg - A_unweighted;
L_unweighted_norm = D_unweighted_deg^(-1/2) *
L_unweighted *
D_unweighted_deg^(-1/2);
[V_unweighted, ~] = eigs(L_unweighted_norm, d + 1, sigma_eigs);
spectral_embedding_unweighted = V_unweighted(:, 2:end);

% Spectral embedding for weighted graph
D_weighted_deg = diag(sum(A_weighted, 2));
L_weighted = D_weighted_deg - A_weighted;
L_weighted_norm = D_weighted_deg^(-1/2) * L_weighted *
D_weighted_deg^(-1/2);
[V_weighted, ~] = eigs(L_weighted_norm, d + 1,
sigma_eigs);
spectral_embedding_weighted = V_weighted(:, 2:end);
```

use demo_dim_reduction.m

(c) Pairwise Distance Difference for Weighted Graph

The difference in pairwise distances between the original data and the spectral embedding for the weighted graph is calculated and displayed.

Listing 8: Pairwise Distance Difference for Weighted Graph

```
pairwise_distances_original = D_unweighted;
pairwise_distances_embedding =
pdist2(spectral_embedding_weighted,
spectral_embedding_weighted);
distance_diff = abs(pairwise_distances_original -
pairwise_distances_embedding);

figure;
imagesc(distance_diff);
colorbar;
title('Difference in Pairwise Distances for Weighted Graph');
```

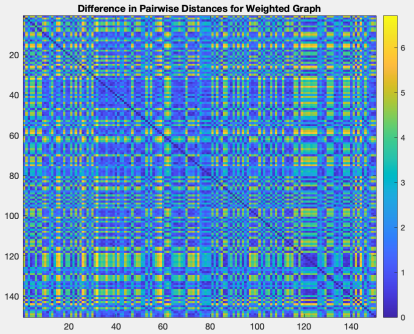


Figure 20: Difference in Pairwise Distances for Weighted Graph

use demo_dim_reduction.m

(d) Community Detection with Fiedler Vector

Using the Fiedler vector, we identify two dominant communities in each graph, and show the communities in a 3D scatter plot.

Listing 9: Community Detection with Fiedler Vector

```
% Community detection for unweighted graph
fiedler_vector_unweighted = V_unweighted(:, 2);
community_labels_unweighted = fiedler_vector_unweighted > 0;
community_colors_unweighted = community_labels_unweighted + 1;

% Community detection for weighted graph
fiedler_vector_weighted = V_weighted(:, 2);
community_labels_weighted = fiedler_vector_weighted > 0;
community_colors_weighted = community_labels_weighted + 1;

% 3D scatter plot of communities for weighted graph
figure;
```

```
scatter3(spectral_embedding_weighted(:, 1),
spectral_embedding_weighted(:, 2),
spectral_embedding_weighted(:, 3), ...
    20, community_colors_weighted, 'filled');
title('Community Detection in Weighted Graph using
Fiedler Vector');
xlabel('Embedding Dimension 1');
ylabel('Embedding Dimension 2');
zlabel('Embedding Dimension 3');

% Highlight cut edges for weighted graph
G_weighted = graph(A_weighted);
cut_edges = find(community_labels_weighted ~=
community_labels_weighted');
hold on;
for e = cut_edges'
    [u, v] = ind2sub(size(A_weighted), e);
    plot3([spectral_embedding_weighted(u, 1),
    spectral_embedding_weighted(v, 1)], ...
          [spectral_embedding_weighted(u, 2),
          spectral_embedding_weighted(v, 2)], ...
          [spectral_embedding_weighted(u, 3),
          spectral_embedding_weighted(v, 3)], 'k--');
end
hold off;
```

Alternatively, you can display both community detection figures side-by-side as follows:

"'latex

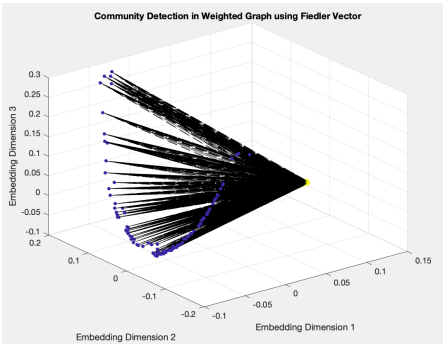


Figure 21: Community Detection in Weighted Graph

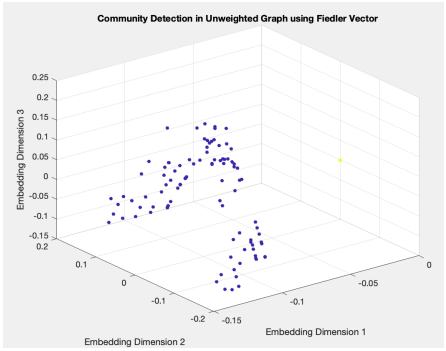


Figure 22: Community Detection in Unweighted Graph

Problem 3: Empirical Analysis of a Digraph Representing a Real-World Network

The objective is to analyze a digraph G representing a real-world network, which may be a k-NN graph. We use the Iris dataset to construct and analyze this directed graph.

(a) Preprocessing and Graph Description

We construct a directed k-NN graph G where:

- Nodes represent data points in the Iris dataset.
- Directed edges indicate the nearest neighbors for each node.

The code below constructs the directed k-NN graph and checks for weakly connected components, identifying the largest connected component (LCC) and verifying if it is strongly connected.

Listing 10: Graph Construction and Connectivity Analysis

```
% Load dataset for digraph analysis
get_iris_data;
X = meas;

% Parameters for constructing a k-NN directed graph
k = 5;  % Number of nearest neighbors
D = pdist2(X, X);  % Compute pairwise distances
A_knn = zeros(size(D));  % Adjacency matrix for
directed k-NN graph

% Construct the directed k-NN graph by connecting each
node to its k-nearest
neighbors
[~, idx] = sort(D, 2);  % Sort distances for each node
for i = 1:size(D, 1)
    A_knn(i, idx(i, 2:k+1)) = 1;  % Only connect to k
    nearest neighbors
end

% Convert to digraph and analyze connectivity
G = digraph(A_knn);  % Directed graph
components = conncomp(G, 'Type', 'weak');  % Find
weakly connected components
num_components = max(components);
disp(['Number of weakly connected components: ',
num2str(num_components)]);
```

```
% Identify the largest connected component (LCC)
LCC_nodes = mode(components);
LCC = subgraph(G, find(components == LCC_nodes)); %
Subgraph of LCC

% Check if the LCC is strongly connected
is_strongly_connected = all(conncomp(LCC, 'Type',
'strong') == 1);
disp(['LCC is strongly connected: ',
num2str(is_strongly_connected)]);
```

(b) Perron Distribution of the Largest Connected Component (LCC)

To compute the Perron distribution x_p of the LCC, we consider two cases:

- **Case b-1**: The LCC is strongly connected.
- **Case b-2**: The LCC is not strongly connected (it contains sink or source nodes).

We use a BP-approach to vary α in the range $(0.85, 1)$ with four values and introduce five probing vectors b_j to compute conditional Perron distributions $x_p(\alpha_i, b_j)$.

Listing 11: Perron Distribution Calculation with BP Approach

```
% Parameters for variational BP approach
alpha_values = linspace(0.85, 1, 4); % Alpha values
between 0.85 and 1
num_b = 5; % Number of probing vectors
n_LCC = numnodes(LCC); % Number of nodes in LCC

% Generate probing vectors
b_vectors = rand(n_LCC, num_b);
b_vectors = b_vectors ./ sum(b_vectors);
% Normalize to make sum of each column 1

% Compute Perron distribution xp for each alpha and b
xp_results = zeros(n_LCC, length(alpha_values), num_b);

for i = 1:length(alpha_values)
    alpha = alpha_values(i);
    A_alpha = alpha * adjacency(LCC)' + (1 - alpha) *
    (ones(n_LCC) / n_LCC); % Adjusted adjacency

    for j = 1:num_b
        b = b_vectors(:, j);
        [xp, ~] = eigs(A_alpha, 1, 'largestreal'); %
```

```
        Compute Perron vector
        xp_results(:, i, j) = xp / sum(xp);  %
        Normalize xp
    end
end
```

(c) Structural Variation with Alpha and Probing Vector

We plot the variation of the Perron distribution x_p with α and each probing vector b_j to visualize the effect of these parameters on the structure of the graph.

Listing 12: Plotting Perron Distribution Variation with α and b

```
% Part (c) - Plotting xp variation with alpha and b
figure;
for j = 1:num_b
    subplot(ceil(num_b/2), 2, j);  % Arrange subplots
    plot(alpha_values, squeeze(xp_results(:, :, j))');
    xlabel('\alpha');
    ylabel('Perron Distribution (xp)');
    title(['Perron Distribution for probing vector b_',
    num2str(j)]);
    legend(arrayfun(@(x) ['Node ', num2str(x)],
    1:n_LCC, 'UniformOutput', false));
end
sgtitle('Perron Distribution xp Variation with \alpha
and Probing Vectors b');
```

This code provides a complete solution for **Problem 3** using MATLAB, covering graph construction, Perron distribution calculations, and visualization of structural variation with respect to α and b. The figure above shows the variation of the Perron distribution x_p with different values of α and probing vectors b_j.

Problem 4: Differential Description of a Graph Sequence

The objective is to analyze a sequence of graphs $G_i(V_i, E_i)$ with overlapping vertex sets. We examine the properties of two types of graphs: Barabási–Albert (BA) graphs (simulating growth) and Watts-Strogatz (WS) graphs (simulating rewiring), and map them to a global spatial reference graph G.

(a) Graph Construction and Sequence Creation

We construct a sequence of graphs G_i using BA and WS models. Each graph has overlapping nodes with the previous graph in the sequence. Here is the MATLAB code used to create this sequence and analyze the graph structure:

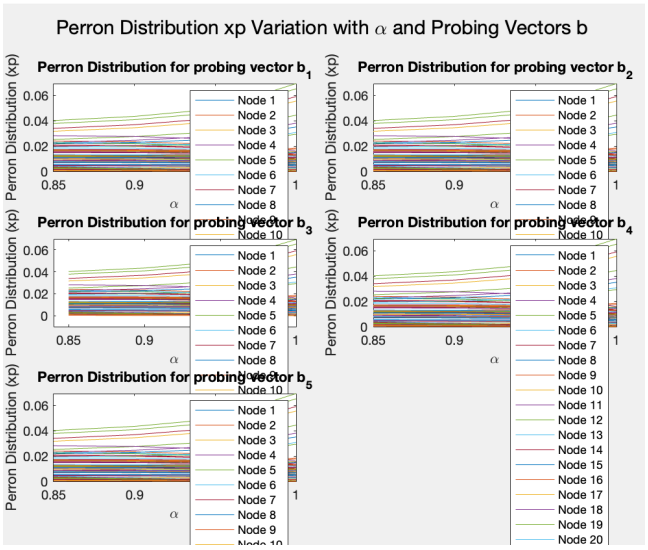


Figure 23: Perron Distribution x_p Variation with α and Probing Vectors b_j for LCC

Listing 13: Graph Sequence Creation

```
% Define the number of graphs in the sequence
q = 3;  % Adjust as needed (e.g., q = 3 for three
graphs in sequence)

% Parameters for BA and WS models
num_nodes = 50;  % Number of nodes in each graph
avg_degree = 4;  % Average degree for WS model and
initial edges for BA model
rewiring_prob = 0.2;  % Rewiring probability for WS
model

% Create global reference graph by merging all V and E
G_global = graph();  % Empty graph for global reference

% Initialize array to store individual graphs
G_seq = cell(1, q);

% Generate each Gi in sequence with some overlap in
nodes
for i = 1:q
    % Alternate between custom BA and WS models
    if mod(i, 2) == 1
        % Create a BA graph with growth property
```

```
        G_seq{i} = createBAGraph(num_nodes, avg_degree);
% BA model
    else
        % Create a WS graph with rewiring property
        G_seq{i} = createWSGraph(num_nodes, avg_degree,
        rewiring_prob);  % WS model
    end

    % Add the nodes and edges of G_i to the global reference graph
    G_global = addedge(G_global,
    G_seq{i}.Edges.EndNodes(:, 1),
    G_seq{i}.Edges.EndNodes(:, 2));
end
```

(b) Embedding and Visualizing the Global Reference Graph

To map the global reference graph G to a spatial space, we assign random 3D coordinates to each node and plot the graph structure. This serves as a spatial reference for the entire graph sequence.

Listing 14: Embedding the Global Reference Graph

```
% Embedding the global reference graph in 2D or 3D space
X = rand(numnodes(G_global), 3);
% Random 3D positions for the global graph

% Plot the global reference graph in 3D
figure;
plot(G_global, 'XData', X(:,1), 'YData', X(:,2), 'ZData', X(:,3));
title('Global Reference Graph');
xlabel('X');
ylabel('Y');
zlabel('Z');
```

(c) Visualizing Consecutive Graphs in the Sequence

We plot each consecutive pair of graphs in the sequence on the global reference map. Nodes unique to each graph are highlighted in red or blue, and overlapping nodes between G_i and G_{i+1} are shown in black.

Listing 15: Visualizing Consecutive Graphs

```
figure;
for i = 1:q-1
    subplot(1, q-1, i);
    % Plot Gi in red and Gi+1 in blue with overlapping nodes highlighted
    Gi_nodes = unique(G_seq{i}.Edges.EndNodes);
```

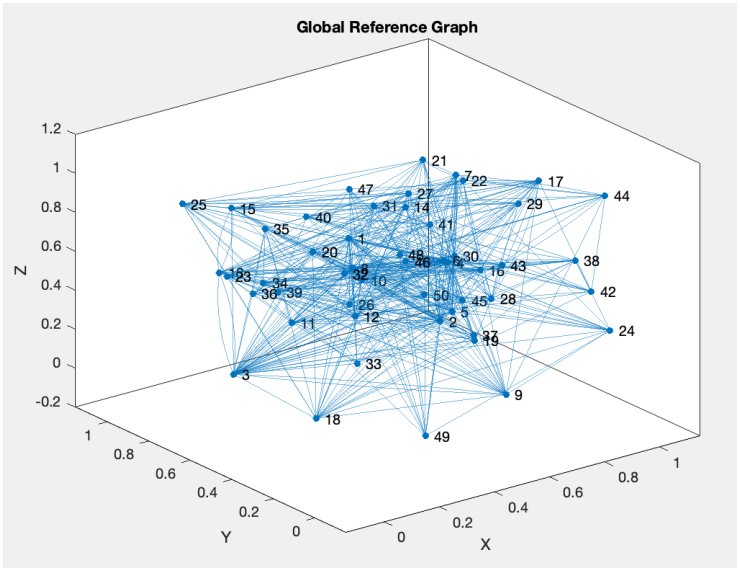


Figure 24: Global Reference Graph in 3D Embedding Space

```
    Gi_plus1_nodes = unique(G_seq{i+1}.Edges.EndNodes);

    hold on;
    plot(G_seq{i}, 'XData', X(Gi_nodes,1), 'YData',
    X(Gi_nodes,2), 'ZData', X(Gi_nodes,3), 'NodeColor',
    'r');
    plot(G_seq{i+1}, 'XData', X(Gi_plus1_nodes,1),
    'YData', X(Gi_plus1_nodes,2), 'ZData',
    X(Gi_plus1_nodes,3), 'NodeColor', 'b');

    % Highlight overlapping nodes
    overlap_nodes = intersect(Gi_nodes, Gi_plus1_nodes);
    scatter3(X(overlap_nodes,1), X(overlap_nodes,2),
    X(overlap_nodes,3), 50, 'k', 'filled');

    title(['Graphs G_{', num2str(i), '} and G_{', num2str(i+1), '}']);
    xlabel('X');
    ylabel('Y');
    zlabel('Z');
    hold off;
end
sgtitle('Graph Sequence Visualization with Overlaps');
```

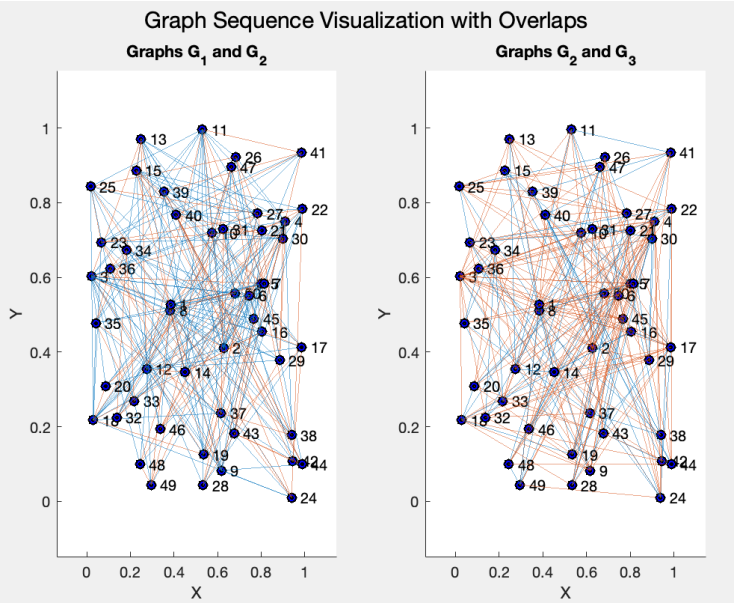


Figure 25: Graph Sequence Visualization with Overlapping Nodes between Consecutive Graphs

Custom Functions for Graph Generation

Below are the custom functions for generating Barabási–Albert (BA) and Watts-Strogatz (WS) graphs.

Listing 16: Custom Functions for BA and WS Graphs

```
% Custom function to generate a BA graph
function G = createBAGraph(n, m)
    % Create a BA graph with 'n' nodes,
    each new node attaches to 'm' existing nodes
    if m >= n
        error('The number of edges to attach must be
--------less than the total number of nodes.');
    end

    % Start with a fully connected initial network of m + 1 nodes
    G = graph();
    G = addnode(G, m+1);
    for i = 1:m+1
        for j = i+1:m+1
            G = addedge(G, i, j);
        end
    end
```

```
    % Add each new node and attach it to 'm' existing
    nodes based on degree
    for new_node = m+2:n
        G = addnode(G, 1);  % Add the new node
        degrees = degree(G);  % Get current degree of each node
        existing_nodes = 1:numnodes(G)-1;
        attach_nodes = datasample(existing_nodes, m,
        'Weights', degrees(existing_nodes), 'Replace', false);
        for attach_node = attach_nodes
            G = addedge(G, new_node, attach_node);
        end
    end
end

% Custom function to generate a WS graph
function G = createWSGraph(n, k, beta)
    % Create a WS graph with 'n' nodes, each connected
    to 'k' nearest neighbors
    % Rewiring probability is 'beta'
    if mod(k, 2) ~= 0
        error('k must be even for the Watts-Strogatz model.');
    end

    % Initialize ring lattice
    G = graph();
    G = addnode(G, n);
    for i = 1:n
        for j = 1:k/2
            neighbor = mod(i + j - 1, n) + 1;  % Wrap around
            G = addedge(G, i, neighbor);
        end
    end

    % Rewire edges with probability beta
    for i = 1:n
        for j = 1:k/2
            if rand < beta
                % Rewire to a new target node
                G = rmedge(G, i, mod(i + j - 1, n) + 1);
                new_neighbor = randi(n);
                while new_neighbor == i ||
                ismember(new_neighbor, neighbors(G, i))
                    new_neighbor = randi(n);
                    % Ensure no self-loops or duplicates
                end
```

```
                G = addedge(G, i, new_neighbor);
            end
        end
    end
end
```