GraphLearning_CodingAssignment

June 25, 2024

1 Coding Assignment - "Graph Learning"

1.1 1. Preparation

1.1.1 1.1 Libraries

[1]:
```
import numpy as np
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.mixture import GaussianMixture

# We will use networx objects to store empircial graphs, local datasets and␣
↪models
import networkx as nx
from sklearn.neighbors import kneighbors_graph
from numpy import linalg as LA
```

1.1.2 1.2 Helper functions

[2]:
```
# The function generates a scatter plot of nodes (=FMI stations) using
# latitude and longitude as coordinates.
def plotFMI(G_FMI):

    # Get the coordinates of the stations.
    coords = np.array([G_FMI.nodes[node]['coord'] for node in G_FMI.nodes])

    # Draw nodes
    for node in G_FMI.nodes:
        plt.scatter(coords[node,1], coords[node,0], color='black', s=4,␣
↪zorder=5)  # zorder ensures nodes are on top of edges
        plt.text(coords[node,1]+0.1, coords[node,0]+0.2, str(node), fontsize=8,␣
↪ha='center', va='center', color='black', fontweight='bold')
    # Draw edges
    for edge in G_FMI.edges:
```

```
        plt.plot([coords[edge[0],1],coords[edge[1],1]],␣
↪[coords[edge[0],0],coords[edge[1],0]], linestyle='-', color='gray', alpha=0.
↪5)

    plt.xlabel('longitude')
    plt.ylabel('latitude')
    plt.title('FMI stations')
    plt.show()

# The function below extracts a feature and label from each row
# of dataframe df. Each row is expected to hold a FMI weather
# measurement with cols "Latitude", "Longitude", "temp", "Timestamp".
# Returns numpy arrays X, y.
def ExtractFeatureMatrixLabelVector(data):
    nrfeatures = 7
    nrdatapoints = len(data)

    # We build the feature matrix X (each of its rows hold the features of a␣
↪data point)
    # and the label vector y (whose entries hold the labels of data points).
    X = np.zeros((nrdatapoints, nrfeatures))
    y = np.zeros((nrdatapoints, 1))

    # Iterate over all rows in dataframe and create corresponding feature␣
↪vector and label.
    for ind in range(nrdatapoints):
        # Latitude of FMI station, normalized by 100.
        lat = float(data['Latitude'].iloc[ind])/100
        # Longitude of FMI station, normalized by 100.
        lon = float(data['Longitude'].iloc[ind])/100
        # Temperature value of the data point.
        tmp = data['temp'].iloc[ind]
        # Read the date and time of the temperature measurement.
        date_object = datetime.strptime(data['Timestamp'].iloc[ind], '%Y-%m-%d␣
↪%H:%M:%S')
        # Extract year, month, day, hour, and minute. Normalize these values
        # to ensure that the features are in range [0,1].
        year = float(date_object.year)/2025
        month = float(date_object.month)/13
        day = float(date_object.day)/32
        hour = float(date_object.hour)/25
        minute = float(date_object.minute)/61

        # Store the data point's features and a label.
        X[ind,:] = [lat, lon, year, month, day, hour, minute]
        y[ind,:] = tmp
```

```
    return X, y
```

1.2 2. Data

1.2.1 2.1 Dataset

[ ]:
```
# Import the weather measurements.
data = pd.read_csv('Assignment_MLBasicsData.csv')

# We consider each temperature measurement (=a row in dataframe) as a
# separate data point.
# Get the numbers of data points and the unique stations.
num_stations = len(data.name.unique())
num_datapoints = len(data)
```

1.2.2 2.2 Features and labels

[ ]:
```
# We next build the feature matrix X (each of its rows hold the features of a␣
↪data point)
# and the label vector y (whose entries hold the labels of data points).
X, y = ExtractFeatureMatrixLabelVector(data)

print(f"The created feature matrix contains {np.shape(X)[0]} entries of {np.
↪shape(X)[1]} features each.")
print(f"The created label vector contains {np.shape(y)[0]} measurements.")
```

1.2.3 2.3 Empirical graph

[ ]:
```
# Create a networkX graph
G_FMI = nx.Graph()

# Add a one node per station
G_FMI.add_nodes_from(range(0, num_stations))

for i, station in enumerate(data.name.unique()):
    # Extract data of a certain station
    station_data = data[data.name==station]

    # Extract features and labels
    X_node, y_node = ExtractFeatureMatrixLabelVector(station_data)

    localsamplesize = len(y_node)
    G_FMI.nodes[i]['samplesize'] = localsamplesize # The number of measurements␣
↪of the i-th weather station
    G_FMI.nodes[i]['name'] = station # The name of the i-th weather station
```

```
    G_FMI.nodes[i]['coord'] = np.array([station_data.Latitude.unique()[0],␣
 ↪station_data.Longitude.unique()[0]]) # The coordinates of the i-th weather␣
 ↪station
    G_FMI.nodes[i]['X'] = X_node # The feature matrix for local dataset at node␣
 ↪i
    G_FMI.nodes[i]['y'] = y_node  # The  label vector for local dataset at node␣
 ↪i
    G_FMI.nodes[i]['z'] = None # The representation vector for local dataset at␣
 ↪node i

# Visualize the empirical graph.
plotFMI(G_FMI)
```

1.3 3. The discrepancy measure

1.3.1 3.1 Helper function

[ ]:
```
# Inputs:
#    1. graph_FMI (class: networkx.classes.graph.Graph) - a graph to which the␣
 ↪edges will be added.
#    2. node_degree (class: int) - the minimum number of neighbors (the minimum␣
 ↪node degree).
# Outputs:
#    1. graph (class: networkx.classes.graph.Graph) - a graph with added edges.
def add_edges(graph_FMI, node_degree):
    graph = graph_FMI.copy()

    for node in graph.nodes:

        # TODO: Extract the representation vector of the node.
        # z_node =
        raise NotImplementedError

        # Create storages for discrepancies and the corresponding neighbors.
        d_mins = np.full(shape=node_degree, fill_value=1e10)
        edges = np.full(shape=(node_degree, 2), fill_value=(node, -1))

        for potential_neighbor in graph.nodes:
            if potential_neighbor != node:

                # TODO: Extract the representation vecotr of the potential␣
 ↪neighbor.
                # z_neighbor =
                raise NotImplementedError

                # TODO: Calculate the discrepancy.
                # d =
```

```
                raise NotImplementedError

                # TODO: Find the max discrepancy so far.
                #       Also, find its index to access the
                #       corresponding neighbor later.
                # d_max_idx =
                # d_max =
                raise NotImplementedError

                if d < d_max:
                    # TODO: Store the calculated discrepancy and
                    #       the corresponding neighbor.
                    # d_mins[d_max_idx] =
                    # edges[d_max_idx][1] =
                    raise NotImplementedError

        # Add edges from the given pairs of connected nodes.
        graph.add_edges_from(edges)

    return graph
```

1.3.2 3.2 Student task #1 - The average temperature

[ ]:
```
def add_edges_avg_temp(graph_FMI, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    ####################TODO####################
    # TODO: 1. Create the representation vector for each node.
    #       2. Add the edges based on the representation vectors.
    raise NotImplementedError

    return graph

# Visualize an example graph with
# minimum node degree = 1.
plotFMI(add_edges_avg_temp(G_FMI, 1))
```

1.3.3 3.3 Student task #2 - The difference in GMM parameters

[ ]:
```
def add_edges_GMM_param(graph_FMI, GMM_seed, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    # Define the number components for the GMM.
    n_components = 2
```

```
    ####################TODO####################
    # TODO: 1. Create the representation vector for each node (see "NOTE").
    #       2. Add the edges based on the representation vectors.
    # NOTE: Use the same GMM structure as in the "FL Flavors" assignment.
    #       You can copy-paste your previous implementation.
    raise NotImplementedError

    # Add edges.
    graph = add_edges(graph, n_neighbors)

    return graph

# Visualize an example graph.
plotFMI(add_edges_GMM_param(G_FMI, 4740, 1))
```

1.3.4 3.4 Student task #3 - The gradient of the average squared error loss

```
[ ]: def add_edges_gradient_loss(graph_FMI, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    ####################TODO####################
    # TODO: 1. Create the representation vector for each node (see "NOTE").
    #       2. Add the edges based on the representation vectors.
    # NOTE: 1. Fit a linear regression to the whole dataset
    #          and extract the model's parameters.
    #       2. Calculate the gradient of the average squared error loss for␣
    ↪each node
    #          according to the Section 7.5 in the Lecture Notes .
    raise NotImplementedError

    # Add edges.
    graph = add_edges(graph, n_neighbors)

    return graph

# Visualize an example graph.
plotFMI(add_edges_gradient_loss(G_FMI, 1))
```

1.4 4. Model

1.4.1 4.1 FedGD (TODO)

[ ]:
```python
def FedGD(graph_FMI, split_seed):
    graph = graph_FMI.copy()

    # Define hyperparameters.
    max_iter = 1000 # The number of gradient steps.
    alpha = 0.5 # Alpha parameter.
    l_rate = 0.1 # The learning rate.

    # Create the storages for the training and validation errors.
    num_stations = len(graph.nodes)
    train_errors = np.zeros(num_stations)
    val_errors = np.zeros(num_stations)

    # TODO: Use your previous implementation of
    #       the FedGD algorithm.
    #       See coding assignment "FL Algorithms".
    # HINT: 1. Split the local datasets into training and validation sets.
    #       2. Initialize all weight vectors with zeros.
    #       3. Perform FedGD on the local training sets.
    #       4. Compute and store the training and validation errors
    #          for each node.
    raise NotImplementedError

    # Output the average training and validation errors.
    return np.mean(train_errors), np.mean(val_errors)
```

1.4.2 4.2 Test connectivity

[ ]:
```python
# Define the random seed for
# add_edges_GMM_param function.
seed = 4740

for num_neighbors in range(1, 11):
    G_FMI_1 = add_edges_avg_temp(G_FMI, num_neighbors)
    G_FMI_2 = add_edges_GMM_param(G_FMI, seed, num_neighbors)
    G_FMI_3 = add_edges_gradient_loss(G_FMI, num_neighbors)

    # Print the results.
    print(f"The minimum number of neighbors is {num_neighbors}")
    print(f"G_FMI_1 is connected: {nx.is_connected(G_FMI_1)}")
    print(f"G_FMI_2 is connected: {nx.is_connected(G_FMI_2)}")
    print(f"G_FMI_3 is connected: {nx.is_connected(G_FMI_3)}\n")
```

1.4.3 4.3 FedGD errors

[ ]:
```python
# Define the random seed for
# add_edges_GMM_param and FedGD functions.
seed = 4740

# The minimum number of neighbors to connect with.
num_neighbors = 1

# Add edges.
G_FMI_1 = add_edges_avg_temp(G_FMI, num_neighbors)
G_FMI_2 = add_edges_GMM_param(G_FMI, seed, num_neighbors)
G_FMI_3 = add_edges_gradient_loss(G_FMI, num_neighbors)

# Apply the FedGD algorithm.
G_FMI_1_train_error, G_FMI_1_val_error = FedGD(G_FMI_1, seed)
G_FMI_2_train_error, G_FMI_2_val_error = FedGD(G_FMI_2, seed)
G_FMI_3_train_error, G_FMI_3_val_error = FedGD(G_FMI_3, seed)

# Print the results.
print(f"The seed is {seed}")
print(f"The average training error for G_FMI_1: {G_FMI_1_train_error}\nThe␣
↪average validation error for G_FMI_1: {G_FMI_1_val_error}\n")
print(f"The average training error for G_FMI_2: {G_FMI_2_train_error}\nThe␣
↪average validation error for G_FMI_2: {G_FMI_2_val_error}\n")
print(f"The average training error for G_FMI_3: {G_FMI_3_train_error}\nThe␣
↪average validation error for G_FMI_3: {G_FMI_3_val_error}\n\n")
```

[ ]:
```python

```

GraphLearning_RefSol

June 25, 2024

1 Reference Solution for Coding Assignment "Graph Learning"

1.1 1. Preparation

1.1.1 1.1 Libraries

[1]:
```
import numpy as np
import pandas as pd
from datetime import datetime
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.mixture import GaussianMixture

# We will use networx objects to store empircial graphs, local datasets and␣
 ↪models
import networkx as nx
from sklearn.neighbors import kneighbors_graph
from numpy import linalg as LA
```

1.1.2 1.2 Helper functions

[2]:
```
# The function generates a scatter plot of nodes (=FMI stations) using
# latitude and longitude as coordinates.
def plotFMI(G_FMI):

    # Get the coordinates of the stations.
    coords = np.array([G_FMI.nodes[node]['coord'] for node in G_FMI.nodes])

    # Draw nodes
    for node in G_FMI.nodes:
        plt.scatter(coords[node,1], coords[node,0], color='black', s=4,␣
 ↪zorder=5)  # zorder ensures nodes are on top of edges
        plt.text(coords[node,1]+0.1, coords[node,0]+0.2, str(node), fontsize=8,␣
 ↪ha='center', va='center', color='black', fontweight='bold')
    # Draw edges
    for edge in G_FMI.edges:
```

```
        plt.plot([coords[edge[0],1],coords[edge[1],1]],␣
↪[coords[edge[0],0],coords[edge[1],0]], linestyle='-', color='gray', alpha=0.
↪5)

    plt.xlabel('longitude')
    plt.ylabel('latitude')
    plt.title('FMI stations')
    plt.show()

# The function below extracts a feature and label from each row
# of dataframe df. Each row is expected to hold a FMI weather
# measurement with cols "Latitude", "Longitude", "temp", "Timestamp".
# Returns numpy arrays X, y.
def ExtractFeatureMatrixLabelVector(data):
    nrfeatures = 7
    nrdatapoints = len(data)

    # We build the feature matrix X (each of its rows hold the features of a␣
↪data point)
    # and the label vector y (whose entries hold the labels of data points).
    X = np.zeros((nrdatapoints, nrfeatures))
    y = np.zeros((nrdatapoints, 1))

    # Iterate over all rows in dataframe and create corresponding feature␣
↪vector and label.
    for ind in range(nrdatapoints):
        # Latitude of FMI station, normalized by 100.
        lat = float(data['Latitude'].iloc[ind])/100
        # Longitude of FMI station, normalized by 100.
        lon = float(data['Longitude'].iloc[ind])/100
        # Temperature value of the data point.
        tmp = data['temp'].iloc[ind]
        # Read the date and time of the temperature measurement.
        date_object = datetime.strptime(data['Timestamp'].iloc[ind], '%Y-%m-%d␣
↪%H:%M:%S')
        # Extract year, month, day, hour, and minute. Normalize these values
        # to ensure that the features are in range [0,1].
        year = float(date_object.year)/2025
        month = float(date_object.month)/13
        day = float(date_object.day)/32
        hour = float(date_object.hour)/25
        minute = float(date_object.minute)/61

        # Store the data point's features and a label.
        X[ind,:] = [lat, lon, year, month, day, hour, minute]
        y[ind,:] = tmp
```

```
    return X, y
```

1.2 2. Data

1.2.1 2.1 Dataset

[3]:
```
# Import the weather measurements.
data = pd.read_csv('Assignment_MLBasicsData.csv')

# We consider each temperature measurement (=a row in dataframe) as a
# separate data point.
# Get the numbers of data points and the unique stations.
num_stations = len(data.name.unique())
num_datapoints = len(data)
```

1.2.2 2.2 Features and labels

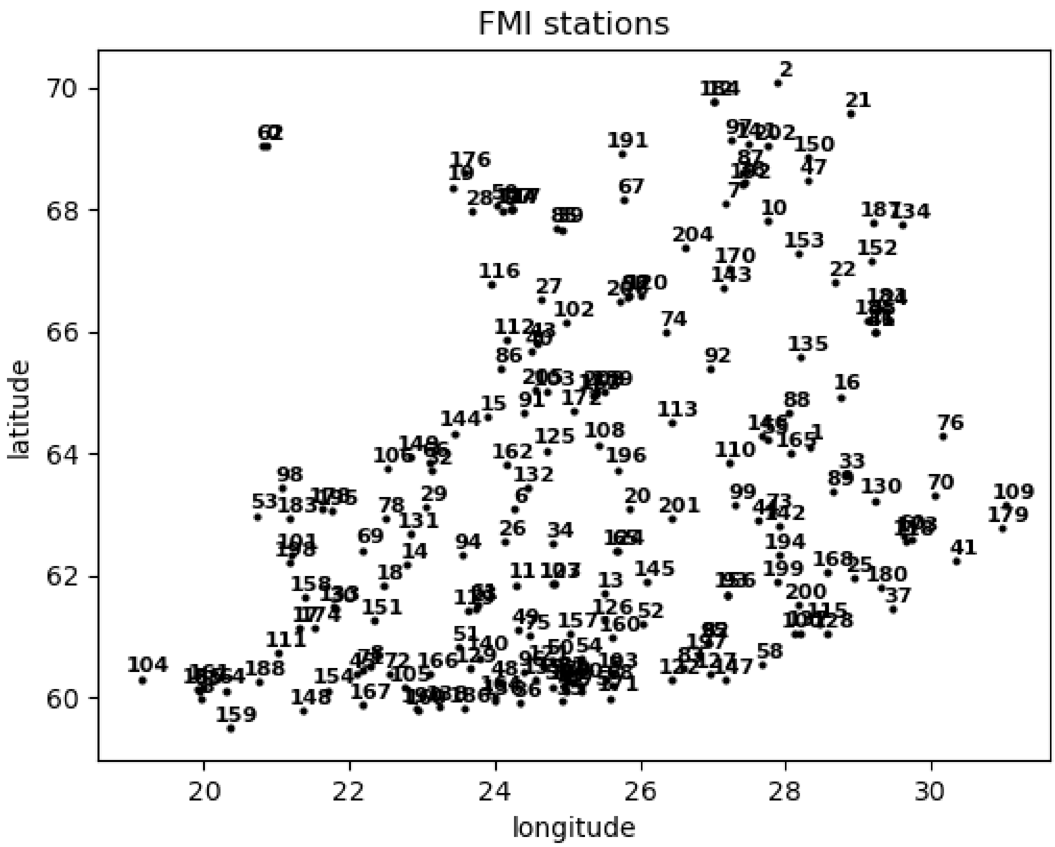
[4]:
```
# We next build the feature matrix X (each of its rows hold the features of a␣
↪data point)
# and the label vector y (whose entries hold the labels of data points).
X, y = ExtractFeatureMatrixLabelVector(data)

print(f"The created feature matrix contains {np.shape(X)[0]} entries of {np.
↪shape(X)[1]} features each.")
print(f"The created label vector contains {np.shape(y)[0]} measurements.")
```

```
The created feature matrix contains 19768 entries of 7 features each.
The created label vector contains 19768 measurements.
```

1.2.3 2.3 Empirical graph

[5]:
```
# Create a networkX graph
G_FMI = nx.Graph()

# Add a one node per station
G_FMI.add_nodes_from(range(0, num_stations))

for i, station in enumerate(data.name.unique()):
    # Extract data of a certain station
    station_data = data[data.name==station]

    # Extract features and labels
    X_node, y_node = ExtractFeatureMatrixLabelVector(station_data)

    localsamplesize = len(y_node)
    G_FMI.nodes[i]['samplesize'] = localsamplesize # The number of measurements␣
↪of the i-th weather station
```

```
    G_FMI.nodes[i]['name'] = station # The name of the i-th weather station
    G_FMI.nodes[i]['coord'] = np.array([station_data.Latitude.unique()[0], station_data.Longitude.unique()[0]]) # The coordinates of the i-th weather station
    G_FMI.nodes[i]['X'] = X_node # The feature matrix for local dataset at node i
    G_FMI.nodes[i]['y'] = y_node  # The  label vector for local dataset at node i
    G_FMI.nodes[i]['z'] = None # The representation vector for local dataset at node i

# Visualize the empirical graph.
plotFMI(G_FMI)
```



1.3 3. The discrepancy measure

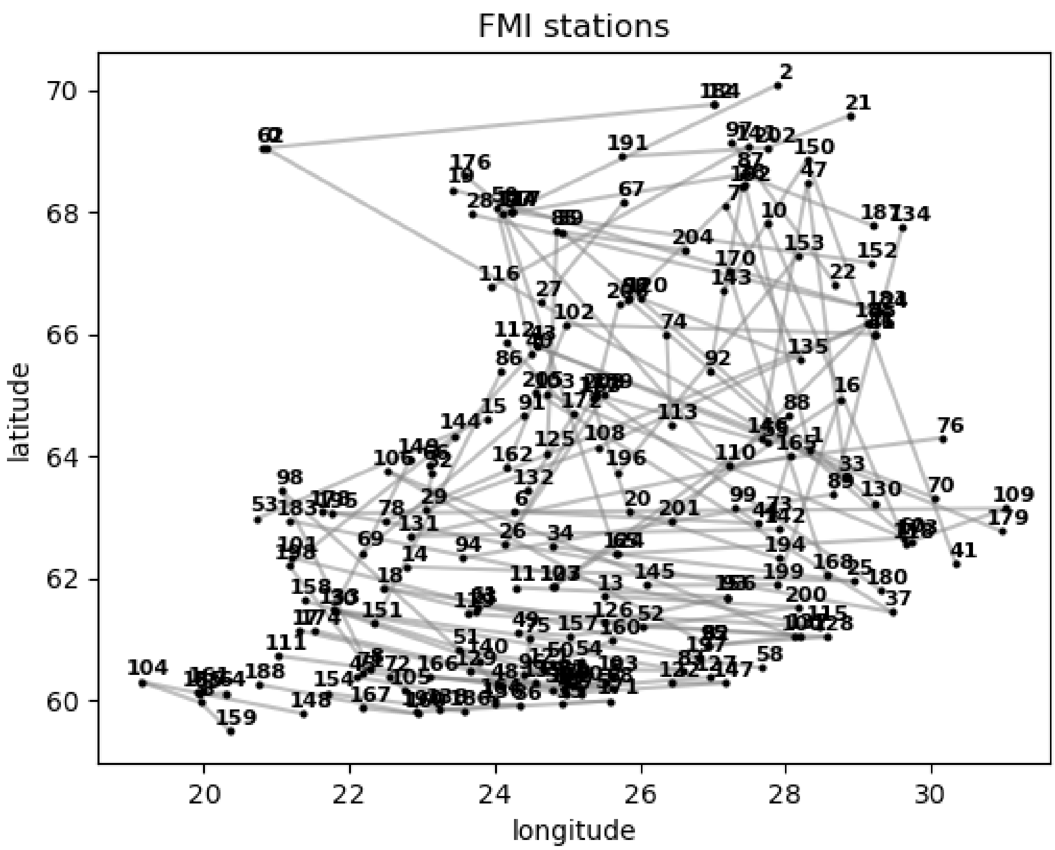
1.3.1 3.1 Helper function

[6]:
```
def add_edges(graph_FMI, node_degree):
    graph = graph_FMI.copy()

    for node in graph.nodes:

        z_node = graph.nodes[node]['z']

        # Create storages for discrepancies and the corresponding neighbors.
        d_mins = np.full(shape=node_degree, fill_value=1e10)
        edges = np.full(shape=(node_degree, 2), fill_value=(node, -1))

        for potential_neighbor in graph.nodes:
            if potential_neighbor != node:
                z_neighbor = graph.nodes[potential_neighbor]['z']
                d = LA.norm(z_node - z_neighbor)

                # Find the max discrepancy so far.
                d_max_idx = np.argmax(d_mins)
                d_max = d_mins[d_max_idx]

                if d < d_max:
                    d_mins[d_max_idx] = d
                    edges[d_max_idx][1] = potential_neighbor

        # print(f"Node {node} has neighbors {[edges[neighbor][1] for neighbor␣
↪in range(node_degree)]}")
        graph.add_edges_from(edges)

    return graph
```

1.3.2 3.2 Student task #1 - The average temperature

[7]:
```
def add_edges_avg_temp(graph_FMI, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    # Create the representation vector for each node.
    for node in graph.nodes:
        avg_temp = np.mean(graph.nodes[node]['y'])
        graph.nodes[node]['z'] = avg_temp

    # Add edges.
    graph = add_edges(graph, n_neighbors)
```

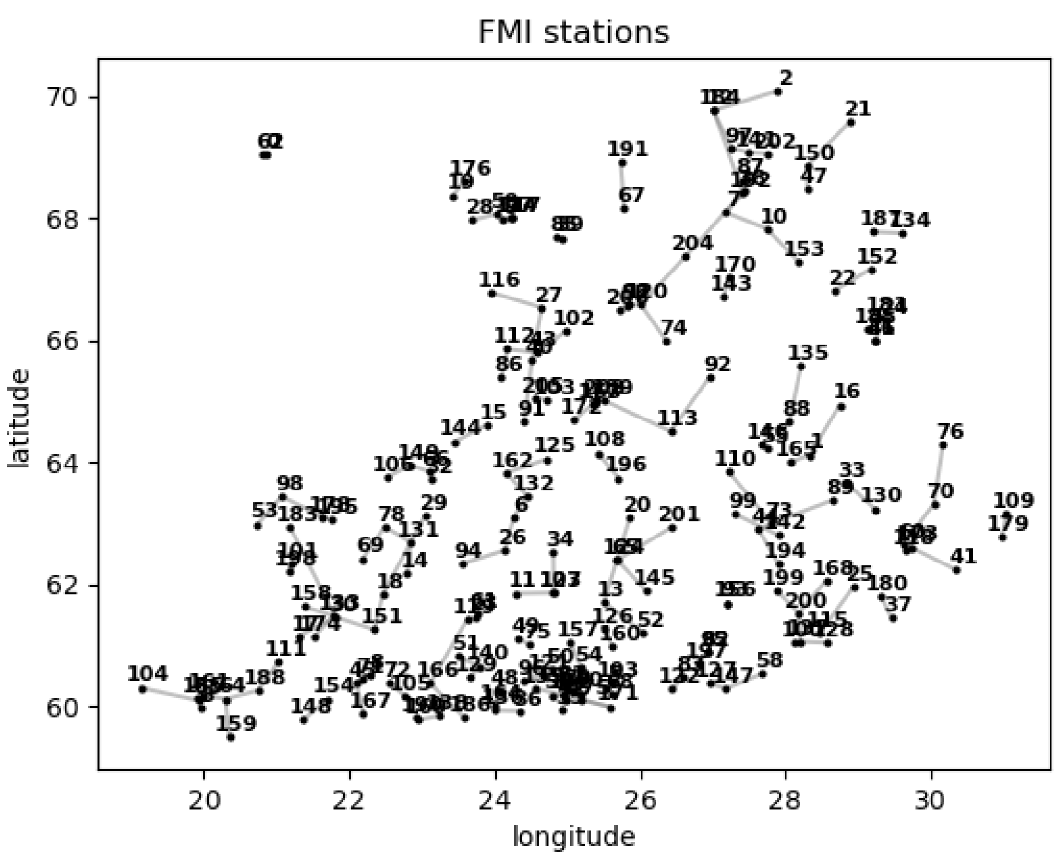
```python
    return graph

# Visualize an example graph.
plotFMI(add_edges_avg_temp(G_FMI, 1))
```



1.3.3 3.3 Student task #2 - The difference in GMM parameters

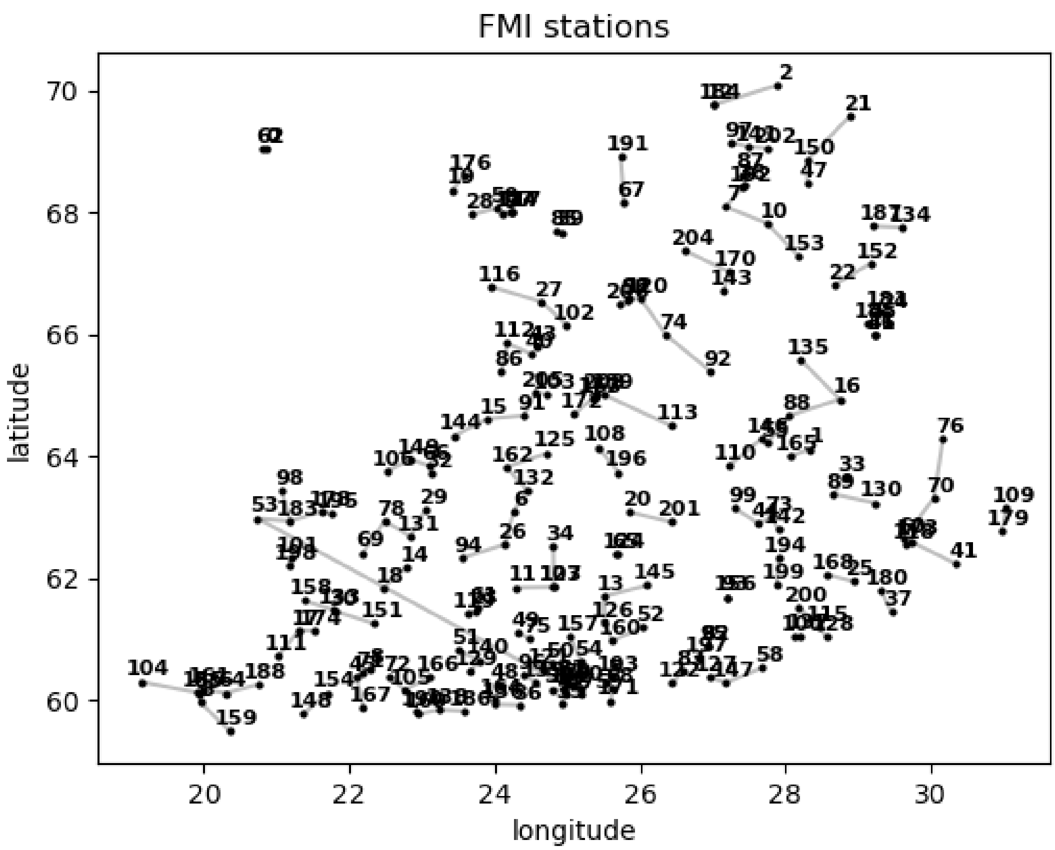
[8]:
```python
def add_edges_GMM_param(graph_FMI, GMM_seed, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    # Define the number components for the GMM.
    n_components = 2

    # Fit the nodes to the GMM model and extract the model's parameters.
    for node in graph.nodes():

        # Extract node's features.
        node_X = graph.nodes[node]['X']
```

```python
        # Fit GMM.
        gmm = GaussianMixture(n_components=n_components, random_state = GMM_seed)
        gmm.fit(node_X)

        # Get the parameters of the GMM (mean vectors, covariance matricies, and component weights).
        gmm_params = np.concatenate((np.concatenate((gmm.means_.ravel(), gmm.covariances_.ravel())), gmm.weights_))

        # Assign GMM parameters to the node.
        graph.nodes[node]['z'] = gmm_params

    # Add edges.
    graph = add_edges(graph, n_neighbors)

    return graph

# Visualize an example graph.
plotFMI(add_edges_GMM_param(G_FMI, 4740, 1))
```



1.3.4 3.4 Student task #3 - The gradient of the average squared error loss

[9]:
```
def add_edges_gradient_loss(graph_FMI, n_neighbors):
    # Copy the nodes to a new graph.
    graph = graph_FMI.copy()

    # Define and fit the Linear regression.
    linear_reg = LinearRegression()
    linear_reg.fit(X, y)

    # Extract the weight vector.
    w_hat = linear_reg.coef_

    # Calculate the average squared error loss.
    for node in graph.nodes:
        node_X = graph.nodes[node]['X']
        node_y = graph.nodes[node]['y']
        m = graph.nodes[node]['samplesize']
        loss = (-2/m) * node_X.T.dot(node_y - node_X.dot(w_hat.T))
        graph.nodes[node]['z'] = loss

    # Add edges.
    graph = add_edges(graph, n_neighbors)

    return graph

# Visualize an example graph.
plotFMI(add_edges_gradient_loss(G_FMI, 1))
```



1.4 4. Model

1.4.1 4.1 FedGD

[10]:
```python
def FedGD(graph_FMI, split_seed):
    graph = graph_FMI.copy()

    for node in graph.nodes:
        # Split the dataset into training and validation set.
        X_train, X_val, y_train, y_val = train_test_split(graph.nodes[node]['X'], graph.nodes[node]['y'], test_size=0.2, random_state=split_seed)

        graph.nodes[node]['X_train'] = X_train # The training feature matrix for local dataset at node i
        graph.nodes[node]['y_train'] = y_train  # The training label vector for local dataset at node i
        graph.nodes[node]['X_val'] = X_val # The training feature matrix for local dataset at node i
```

```
        graph.nodes[node]['y_val'] = y_val  # The training label vector for␣
↪local dataset at node i

    # Initialize all weight vectors with zeros
    for station in graph.nodes:
        graph.nodes[station]['weights'] = np.zeros((7, 1))

    # Define hyperparameters.
    max_iter = 1000
    alpha = 0.5
    l_rate = 0.1
    num_stations = len(graph.nodes)

    for i in range(max_iter):
        # Iterate over all nodes.
        for current_node in graph.nodes:

            # Extract the training data from the current node.
            X_train = graph.nodes[current_node]['X_train']
            y_train = graph.nodes[current_node]['y_train']
            w_current = graph.nodes[current_node]['weights']
            training_size = len(y_train)

            # Compute the first term of the Equation 5.9.
            term_1 = (2/training_size) * X_train.T.dot(y_train - X_train.
↪dot(w_current))
            # Compute the second term of the Equation 5.9
            # by receiving neighbors' weight vectors.
            term_2 = 0
            neighbors = list(graph.neighbors(current_node))
            for neighbor in neighbors:
                w_neighbor = graph.nodes[neighbor]['weights']
                term_2 += w_neighbor - w_current
            term_2 *= 2*alpha
            # Equation 5.8
            w_updated = w_current + l_rate * (term_1 + term_2)

            # Update the current weight vector but do not overwrite the
            # "weights" attribute as we need to do all updates synchronously, i.
↪e.,
            # using the previous local params

            graph.nodes[current_node]['newweights'] = w_updated

        # after computing the new localparmas for each node, we now update
        # the node attribute 'weights' for all nodes
        for node_id in graph.nodes:
```

```
            graph.nodes[node_id]['weights'] = graph.nodes[node_id]['newweights']

    # Create the storages for the training and validation errors.
    train_errors = np.zeros(num_stations)
    val_errors = np.zeros(num_stations)

    # Iterate over all nodes.
    for station in graph.nodes:
        # Extract the data of the current node.
        X_train = graph.nodes[station]['X_train']
        y_train = graph.nodes[station]['y_train']
        X_val = graph.nodes[station]['X_val']
        y_val = graph.nodes[station]['y_val']
        w = graph.nodes[station]['weights']

        # Compute and store the training and validation errors.
        train_errors[station] = mean_squared_error(y_train, X_train.dot(w))
        val_errors[station] = mean_squared_error(y_val, X_val.dot(w))

    # Output the average training and validation errors.
    return np.mean(train_errors), np.mean(val_errors)
```

1.4.2 4.2 Test connectivity

[13]:
```
# Define the random seed for
# add_edges_GMM_param function.
seed = 4740

for num_neighbors in range(1, 11):
    G_FMI_1 = add_edges_avg_temp(G_FMI, num_neighbors)
    G_FMI_2 = add_edges_GMM_param(G_FMI, seed, num_neighbors)
    G_FMI_3 = add_edges_gradient_loss(G_FMI, num_neighbors)

    # Print the results.
    print(f"The minimum number of neighbors is {num_neighbors}")
    print(f"G_FMI_1 is connected: {nx.is_connected(G_FMI_1)}")
    print(f"G_FMI_2 is connected: {nx.is_connected(G_FMI_2)}")
    print(f"G_FMI_3 is connected: {nx.is_connected(G_FMI_3)}\n")
```

```
The minimum number of neighbors is 1
G_FMI_1 is connected: False
G_FMI_2 is connected: False
G_FMI_3 is connected: False

The minimum number of neighbors is 2
G_FMI_1 is connected: False
G_FMI_2 is connected: False
```

```
G_FMI_3 is connected: False

The minimum number of neighbors is 3
G_FMI_1 is connected: False
G_FMI_2 is connected: False
G_FMI_3 is connected: False

The minimum number of neighbors is 4
G_FMI_1 is connected: False
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 5
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 6
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 7
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 8
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 9
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True

The minimum number of neighbors is 10
G_FMI_1 is connected: True
G_FMI_2 is connected: False
G_FMI_3 is connected: True
```

1.4.3 4.3 FedGD errors

[14]:
```
# Define the random seed for
# add_edges_GMM_param and FedGD functions.
seed = 4740

# The minimum number of neighbors to connect with.
num_neighbors = 1

# Add edges.
G_FMI_1 = add_edges_avg_temp(G_FMI, num_neighbors)
G_FMI_2 = add_edges_GMM_param(G_FMI, seed, num_neighbors)
G_FMI_3 = add_edges_gradient_loss(G_FMI, num_neighbors)

# Apply the FedGD algorithm.
G_FMI_1_train_error, G_FMI_1_val_error = FedGD(G_FMI_1, seed)
G_FMI_2_train_error, G_FMI_2_val_error = FedGD(G_FMI_2, seed)
G_FMI_3_train_error, G_FMI_3_val_error = FedGD(G_FMI_3, seed)

# Print the results.
print(f"The seed is {seed}")
print(f"The average training error for G_FMI_1: {G_FMI_1_train_error}\nThe␣
↪average validation error for G_FMI_1: {G_FMI_1_val_error}\n")
print(f"The average training error for G_FMI_2: {G_FMI_2_train_error}\nThe␣
↪average validation error for G_FMI_2: {G_FMI_2_val_error}\n")
print(f"The average training error for G_FMI_3: {G_FMI_3_train_error}\nThe␣
↪average validation error for G_FMI_3: {G_FMI_3_val_error}\n\n")
```

```
The seed is 4740
The average training error for G_FMI_1: 20.68635695529578
The average validation error for G_FMI_1: 21.386552008386015

The average training error for G_FMI_2: 20.201786964135945
The average validation error for G_FMI_2: 20.892958171150042

The average training error for G_FMI_3: 20.052132233485334
The average validation error for G_FMI_3: 20.64577295712873
```

[ ]:
```
```