FLAlgorithms_CodingAssignment

June 25, 2024

1 Coding Assignment "FL Algorithms"

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.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# 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):
    coords = [G_FMI.nodes[i]['coord'] for i in range(num_stations)]
    df_coords = pd.DataFrame(coords,columns=['latitude','longitude'])
    coords = np.hstack((df_coords["latitude"].to_numpy().
↪reshape(-1,1),df_coords["longitude"].to_numpy().reshape(-1,1)))
    # Create a plot
    fig, ax = plt.subplots()
    # Draw nodes
    for node in G_FMI.nodes:
        ax.scatter(coords[node,1], coords[node,0], color='black', s=4,␣
↪zorder=5)  # zorder ensures nodes are on top of edges
    # Add labels
    for node in G_FMI.nodes:
        ax.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:
        ax.plot([coords[edge[0],1],coords[edge[1],1]],␣
 ↪[coords[edge[0],0],coords[edge[1],0]], linestyle='-', color='gray')

    ax.set_xlabel('longitude')
    ax.set_ylabel('latitude')
    ax.set_title('FMI stations')
    plt.show()


# The function connects each FMI station with
# the nearest neighbours.
def add_edges(graph, numneighbors=4):
    coords = [graph.nodes[i]['coord'] for i in range(num_stations)]
    df_coords = pd.DataFrame(coords,columns=['latitude','longitude'])
    coords = np.hstack((df_coords["latitude"].to_numpy().
 ↪reshape(-1,1),df_coords["longitude"].to_numpy().reshape(-1,1)))
    A = kneighbors_graph(coords, numneighbors, mode='connectivity',␣
 ↪include_self=False)
    nrnodes = len(graph.nodes)
    for iter_i in range(nrnodes):
        for iter_ii in range(nrnodes):
            if iter_i != iter_ii :
                if A[iter_i,iter_ii]> 0 :
                    graph.add_edge(iter_i, iter_ii)
    return graph

# 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 ExtractFeaureMatrixLabvelVector(data):
    nrfeatures = 7
    nrdatapoints = len(data)
    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
        X[ind,:] = [lat, lon, year, month, day, hour, minute]
        y[ind,:] = tmp

    return X, y
```

1.2 2 Data

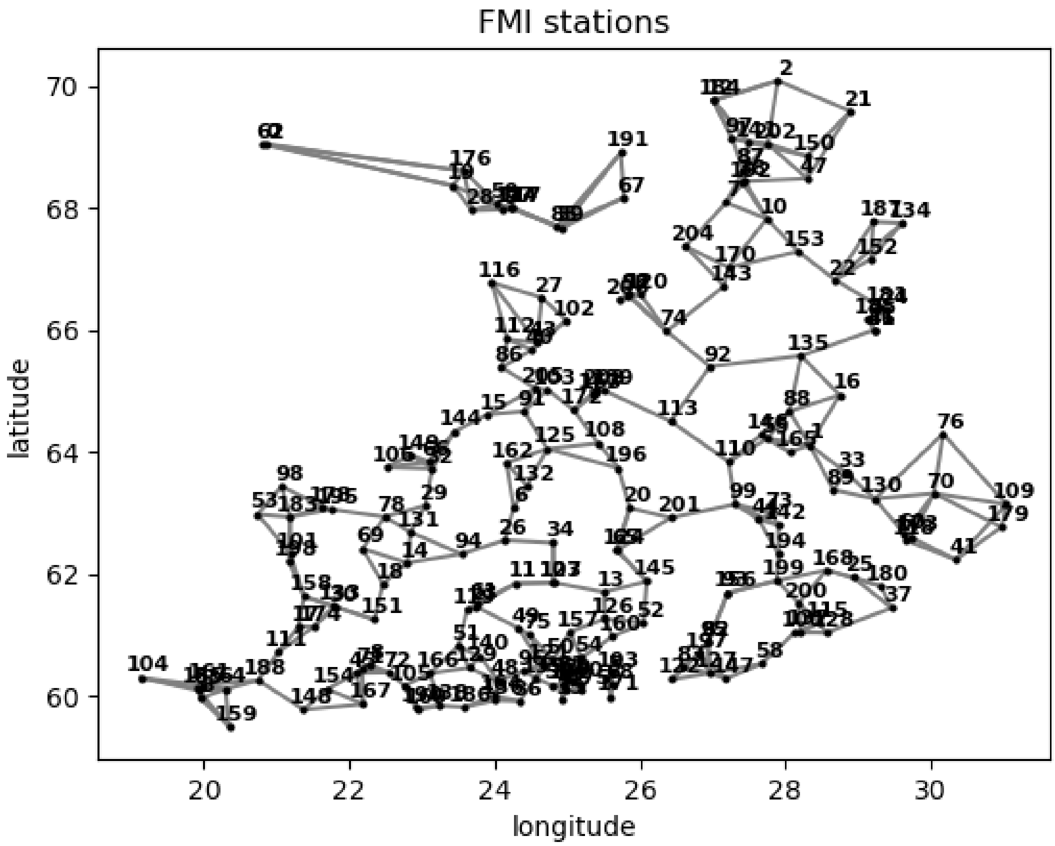
1.2.1 2.1 Dataset

[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 Empirical graph

[4]:
```
# Define the random seed and the fraction of validation set for
# the train_test_split() function.
test_size = 0.2
seed = 1

#############
# In what follows, we
# 1. construct the empirical graph G_FMI as a networkx.Graph() object,
# 2. add a single node for each station,
# 3. for each node add the following attributes:
#   'samplesize' - The number of measurements of the i-th weather station,
#   'name' - The name of the i-th weather station,
#   'coord' - The coordinates of the i-th weather station,
#   'X_train', 'y_train', 'X_val', and 'y_val' - the training and validation␣
↪data,
#   'weights' - the i-th node's model parameters.
```

```
# 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, y = ExtractFeaureMatrixLabvelVector(station_data)

    # Split the dataset into training and validation set.
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=1)

    localsamplesize = len(y)
    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'] = (station_data.Latitude.unique()[0], station_data.
 ↪Longitude.unique()[0]) # The coordinates of the i-th weather station
    G_FMI.nodes[i]['X_train'] = X_train # The training feature matrix for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['y_train'] = y_train  # The training label vector for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['X_val'] = X_val # The training feature matrix for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['y_val'] = y_val  # The training label vector for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['weights'] = np.zeros((7,1))

# Add edges between each station and its nearest neighbors.
# NOTE: the node degree might be different for different nodes.
numneighbors = 3
G_FMI = add_edges(G_FMI, numneighbors=numneighbors)

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



1.3 3. Model

1.3.1 3.1 Student task #1 - FedGD

[5]:
```
# Define hyperparameters.
max_iter = 1000
alpha = 0.5
l_rate = 0.1

###################TODO####################
# TODO: Implement a FedGD algorithm (Algorithm 3 from the Lecture Notes).

raise NotImplementedError
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[5], line 9
      4 l_rate = 0.1
      6 ###################TODO####################
      7 # TODO: Implement a FedGD algorithm (Algorithm 3 from the Lecture Notes).
```

```
----> 9 raise NotImplementedError

NotImplementedError:
```

[6]:
```
# Create the storages for the training and validation errors.
train_errors = np.zeros(num_stations)
val_errors = np.zeros(num_stations)

###################TODO####################
# TODO: Calculate the average (over all nodes) training and validation errors.
# Note: Use mean_squared_error() method for calculating local errors.

raise NotImplementedError

# Output the average training and validation errors.
print("The average training error:", np.mean(train_errors))
print("The average validation error:", np.mean(val_errors))
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[6], line 9
      3 val_errors = np.zeros(num_stations)
      5 ###################TODO####################
      6 # TODO: Calculate the average (over all nodes) training and validation␣
 ↪errors.
      7 # Note: Use mean_squared_error() method for calculating local errors.
----> 9 raise NotImplementedError
     11 # Output the average training and validation errors.
     12 print("The average training error:", np.mean(train_errors))

NotImplementedError:
```

1.3.2 3.2 Student task #2 - FedSGD

[7]:
```
# Define hyperparameters.
max_iter = 1000
alpha = 0.5
l_rate = 0.1
batch_size = 10

###################TODO####################
# TODO: Implement a FedSGD algorithm (Algorithm 4 from the Lecture Notes).

raise NotImplementedError
```

```
---------------------------------------------------------------------------
```

```
NotImplementedError                       Traceback (most recent call last)
Cell In[7], line 10
      5 batch_size = 10
      7 ####################TODO####################
      8 # TODO: Implement a FedSGD algorithm (Algorithm 4 from the Lecture␣
  ↪Notes).
---> 10 raise NotImplementedError

NotImplementedError:
```

[8]:

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

####################TODO####################
# TODO: Calculate the average (over all nodes) training and validation errors.
# Note: Use mean_squared_error() method for calculating local errors.

raise NotImplementedError

# Output the average training and validation errors.
print("The average training error:", np.mean(train_errors))
print("The average validation error:", np.mean(val_errors))
```

```
---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
Cell In[8], line 9
      3 val_errors = np.zeros(num_stations)
      5 ####################TODO####################
      6 # TODO: Calculate the average (over all nodes) training and validation␣
  ↪errors.
      7 # Note: Use mean_squared_error() method for calculating local errors.
----> 9 raise NotImplementedError
     11 # Output the average training and validation errors.
     12 print("The average training error:", np.mean(train_errors))

NotImplementedError:
```

[ ]:

FLAlgorithms_RefSol

June 25, 2024

1 Reference Solution for Assignment "FL Algorithms"

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.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

# 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):
    coords = [G_FMI.nodes[i]['coord'] for i in range(num_stations)]
    df_coords = pd.DataFrame(coords,columns=['latitude','longitude'])
    coords = np.hstack((df_coords["latitude"].to_numpy().
 ↪reshape(-1,1),df_coords["longitude"].to_numpy().reshape(-1,1)))
    # Create a plot
    fig, ax = plt.subplots()
    # Draw nodes
    for node in G_FMI.nodes:
        ax.scatter(coords[node,1], coords[node,0], color='black', s=4,␣
 ↪zorder=5)  # zorder ensures nodes are on top of edges
    # Add labels
    for node in G_FMI.nodes:
        ax.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:
        ax.plot([coords[edge[0],1],coords[edge[1],1]],␣
↪[coords[edge[0],0],coords[edge[1],0]], linestyle='-', color='gray')

    ax.set_xlabel('longitude')
    ax.set_ylabel('latitude')
    ax.set_title('FMI stations')
    plt.show()


# The function connects each FMI station with
# the nearest neighbours.
def add_edges(graph, numneighbors=4):
    coords = [graph.nodes[i]['coord'] for i in range(num_stations)]
    df_coords = pd.DataFrame(coords,columns=['latitude','longitude'])
    coords = np.hstack((df_coords["latitude"].to_numpy().
↪reshape(-1,1),df_coords["longitude"].to_numpy().reshape(-1,1)))
    A = kneighbors_graph(coords, numneighbors, mode='connectivity',␣
↪include_self=False)
    nrnodes = len(graph.nodes)
    for iter_i in range(nrnodes):
        for iter_ii in range(nrnodes):
            if iter_i != iter_ii :
                if A[iter_i,iter_ii]> 0 :
                    graph.add_edge(iter_i, iter_ii)
    return graph

# 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 ExtractFeaureMatrixLabvelVector(data):
    nrfeatures = 7
    nrdatapoints = len(data)
    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
        X[ind,:] = [lat, lon, year, month, day, hour, minute]
        y[ind,:] = tmp

    return X, y
```

1.2 2 Data

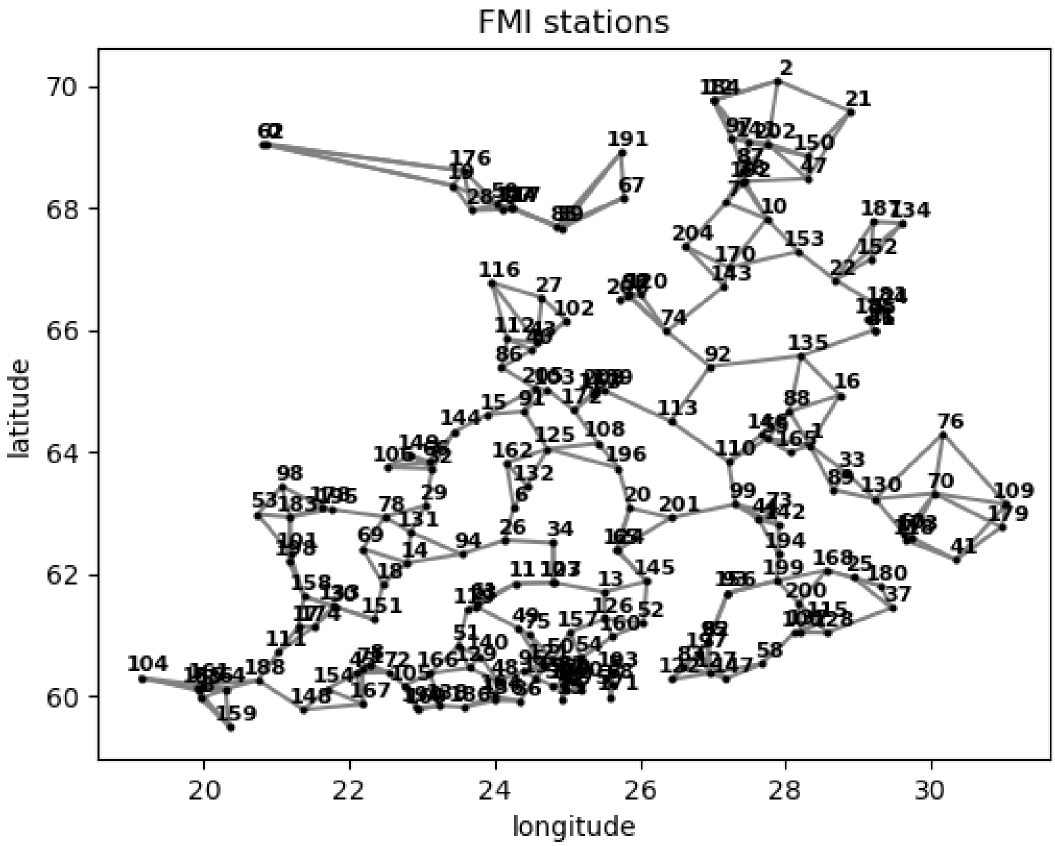
1.2.1 2.1 Dataset

[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 Empirical graph

[4]:
```
# 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, y = ExtractFeaureMatrixLabvelVector(station_data)

    # Split the dataset into training and validation set.
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
↪random_state=1)
```

```
    localsamplesize = len(y)
    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'] = (station_data.Latitude.unique()[0], station_data.
 ↪Longitude.unique()[0]) # The coordinates of the i-th weather station
    G_FMI.nodes[i]['X_train'] = X_train # The training feature matrix for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['y_train'] = y_train  # The training label vector for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['X_val'] = X_val # The training feature matrix for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['y_val'] = y_val  # The training label vector for local␣
 ↪dataset at node i
    G_FMI.nodes[i]['weights'] = np.zeros((7,1))

# Add edges between each station and its nearest neighbors.
# NOTE: the node degree might be different for different nodes.
numneighbors = 3
G_FMI = add_edges(G_FMI, numneighbors=numneighbors)

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



1.3 3. Model

1.3.1 3.1 Student task #1 - FedGD

[5]:
```
# Initialize all weight vectors with zeros
for station in G_FMI.nodes:
    G_FMI.nodes[station]['weights'] = np.zeros((X_train.shape[1], 1))
```

[6]:
```
# Define hyperparameters.
max_iter = 1000
alpha = 0.5
l_rate = 0.1

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

        # Extract the training data from the current node.
        X_train = G_FMI.nodes[current_node]['X_train']
        y_train = G_FMI.nodes[current_node]['y_train']
```

```
        w_current = G_FMI.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(G_FMI.neighbors(current_node))
        for neighbor in neighbors:
            w_neighbor = G_FMI.nodes[neighbor]['weights']
            term_2 += w_neighbor - w_current
        term_2 *= 2*alpha
        # Equation 5.9
        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

        G_FMI.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 G_FMI.nodes:
        G_FMI.nodes[node_id]['weights'] = G_FMI.nodes[node_id]['newweights']
```

[7]:
```
# 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 G_FMI.nodes:
    # Extract the data of the current node.
    X_train = G_FMI.nodes[station]['X_train']
    y_train = G_FMI.nodes[station]['y_train']
    X_val = G_FMI.nodes[station]['X_val']
    y_val = G_FMI.nodes[station]['y_val']
    w = G_FMI.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.
print("The average training error:", np.mean(train_errors))
```

```
print("The average validation error:", np.mean(val_errors))
```

```
The average training error: 21.065277715408104
The average validation error: 21.336245897848386
```

1.3.2 3.2 Student task #2 - FedSGD

[8]:
```
# Initialize all weight vectors with zeros
# we add another node attribute "curr_batch_start" that points to the first␣
↪index of the next
# batch of data points

for station in G_FMI.nodes:
    G_FMI.nodes[station]['weights'] = np.zeros((X_train.shape[1], 1))   #
    G_FMI.nodes[station]['curr_batch_start'] = 0
```

[9]:
```
# Define hyperparameters.
max_iter = 1000
alpha = 0.5
l_rate = 0.1
batch_size = 10

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

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

        # Compute the first term of the Equation 5.11.

        curr_batch_start = G_FMI.nodes[current_node]['curr_batch_start']
      #  print(curr_batch_start)
        # Get the batched features and labels
        X_train_batch = X_train[curr_batch_start:(curr_batch_start+batch_size)]
        y_train_batch = y_train[curr_batch_start:(curr_batch_start+batch_size)]

        # update batch start for the next iteration
        curr_batch_start = curr_batch_start + batch_size
        # check if batch start would be outside the training set
        if curr_batch_start >= training_size:
            curr_batch_start =0  # if next batch exceeds training set size␣
↪start over from first datapoint
        G_FMI.nodes[current_node]['curr_batch_start']=curr_batch_start
```

```
        term_1 = (2/batch_size) * X_train_batch.T.dot(y_train_batch -␣
↪X_train_batch.dot(w_current))

        # Compute the second term of the Equation 5.11
        # by receiving neighbors' weight vectors.
        term_2 = 0
        neighbors = list(G_FMI.neighbors(current_node))
        for neighbor in neighbors:
            w_neighbor = G_FMI.nodes[neighbor]['weights']
            term_2 += w_neighbor - w_current
        term_2 *= 2*alpha
        # Equation 5.11
        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

        G_FMI.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 G_FMI.nodes:
        G_FMI.nodes[node_id]['weights'] = G_FMI.nodes[node_id]['newweights']
```

[10]:
```
# 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 G_FMI.nodes:
    # Extract the data of the current node.
    X_train = G_FMI.nodes[station]['X_train']
    y_train = G_FMI.nodes[station]['y_train']
    X_val = G_FMI.nodes[station]['X_val']
    y_val = G_FMI.nodes[station]['y_val']
    w = G_FMI.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.
print("The average training error:", np.mean(train_errors))
print("The average validation error:", np.mean(val_errors))
```

```
The average training error: 21.911684628667366
The average validation error: 22.07873980583682
```

[ ]:

```

```