BOĞAZIÇI UNIVERSITY

NONLINEAR MODELS IN OPERATIONS RESEARCH
IE 440

Final Assignment

Author:
M. Akın Elden

4 January 2020

Department of Industrial Engineering
Boğaziçi University

1 Introduction

The project is implemented using Python as the programming language. Complete code file and data files are sent with the report.

The source code used to import required dependencies:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

2 Clustering

Both K-means batch and K-means online algorithms starts with random codebook locations. In each iteration, assigns points to the closest codebook and then tries to minimize an error function (z) by centering the codebooks to their assigned points. In SOM algorithm, both the closest codebook and its neighbors are moved toward the point and quantity of movement is calculated by the degree of neighborhood which determined by a continuous function (Gaussian Kernel).

The source code used to read the cluster data:

```
cluster_data = pd.read_csv('IE440Final19ClusteringData.txt', sep='\t', header=0);

cluster_centers = cluster_data.copy()[0:0]
patterns = np.array(cluster_data)[:,:-1]
```

The source code used to split the data into clusters:

```
def splitClusters(x, w, b):
    data = pd.DataFrame(x.reshape(-1,2), columns=["x", "y"])
    clusters = []
    centers = []
    for i in range(len(w)):
        clusters.append(data[b[i] == 1])
        centers.append(pd.DataFrame(w[i].reshape(1,2), columns=["x","y"]))
    return clusters, centers
```

The source code used to find the best model by running multiple times:

```
def findBestTrial(patterns, cluster_number, clustering_function, trial_number=10000,
    seed=440):
    np.random.seed(seed)
    z = np.inf
    b = []
    w = []
    x = []
    for i in range(trial_number):
        x_t, b_t, w_t, z_t = clustering_function(patterns, cluster_number)
        if z_t < z:
            z = z_t
```

```
            b = b_t
            w = w_t
            x = x_t
    return x, b, w, z
```

Function for plotting the clusters:

```
def plotClusters(dataArrays, clusterPoints, colors, title, figsize=[6.4, 4.8], name="
    graph"):
    if len(dataArrays) != len(clusterPoints) or len(dataArrays) != len(colors):
        print("Data, cluster points and colors size doesn't match!")
        return
    plt.figure(figsize=figsize)
    for i in range(len(dataArrays)):
        plt.scatter(dataArrays[i]['x'], dataArrays[i]['y'], color=colors[i], marker=".",
             alpha=0.6)
        plt.scatter(clusterPoints[i]['x'], clusterPoints[i]['y'], color=colors[i],
            marker="p", s=200, edgecolors="black")
    plt.title(title)
    plt.savefig("{0}.png".format(name))
```

K-Means (batch)

The K-Means batch algorithm experimented for K=5, 10 and 15. Each experiment is run for 1000 times (since 10000 times running would take hours to complete), and the best result is shown.

The source code used to implement K-Means batch algorithm:

```
def KMeans_batch(patterns, cluster_number):
    I = cluster_number
    P = np.size(patterns,0)
    K = np.size(patterns,1)
    t = 1
    z_prev = np.inf
    randIndices = np.random.choice(range(P), I, replace=False)
    w_t = np.copy(patterns)[randIndices]
    b_prev = np.zeros((I,P))
    while(True):
        b_t = np.zeros((I,P))
        for p in range(P):
            ip = np.linalg.norm(patterns[p] - w_t, axis=1).argmin()
            b_t[ip,p] = 1
        z_t = 0
        assg_indices = np.argwhere(b_t > 0.5)
        for i in assg_indices:
            z_t += np.sum( (patterns[i[1]] - w_t[i[0]])**2 )
        w_tp = np.copy(w_t)
        for i in range(I):
            for k in range(K):
                w_tp[i,k] = (b_t[i]*patterns[:,k]).sum() / b_t[i].sum()
        if z_t >= z_prev:
            break
        t += 1
```

```
        z_prev = z_t
        b_prev = b_t
        w_t = w_tp
    return patterns, b_prev, w_t, z_prev
```

K=5 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 5, KMeans_batch, trial_number = 1000, seed=440)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue"]
plotClusters(clusters, centers, colors, "5-Means Batch Clustering", name="batch_5")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 1204.5725910455437

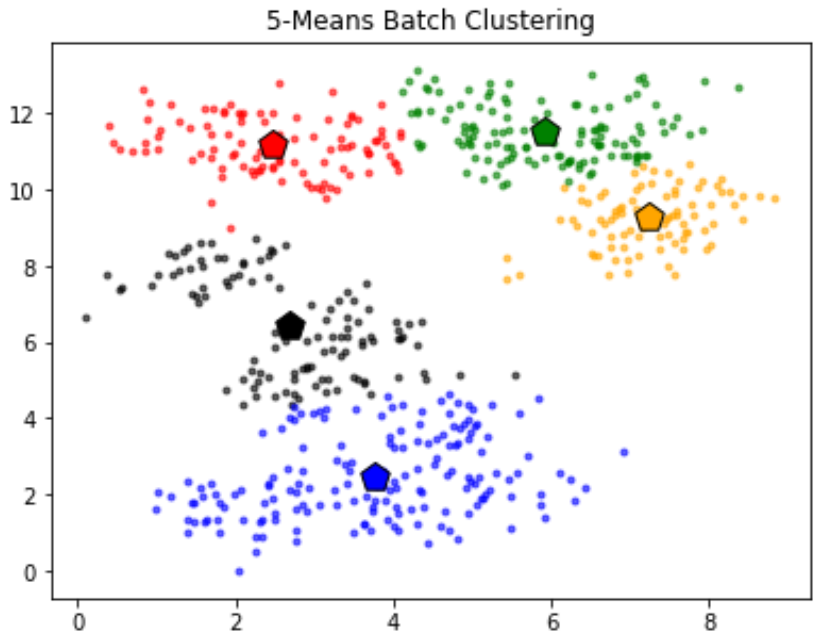

Figure 1: Scatter plot of K=5 batch algorithm

K=10 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 10, KMeans_batch, trial_number=1000, seed=72)
clusters, centers = splitClusters(x, w, b)
```

```
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink"]
plotClusters(clusters, centers, colors, "10-Means Batch Clustering", name="batch_10")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 502.2076811426707

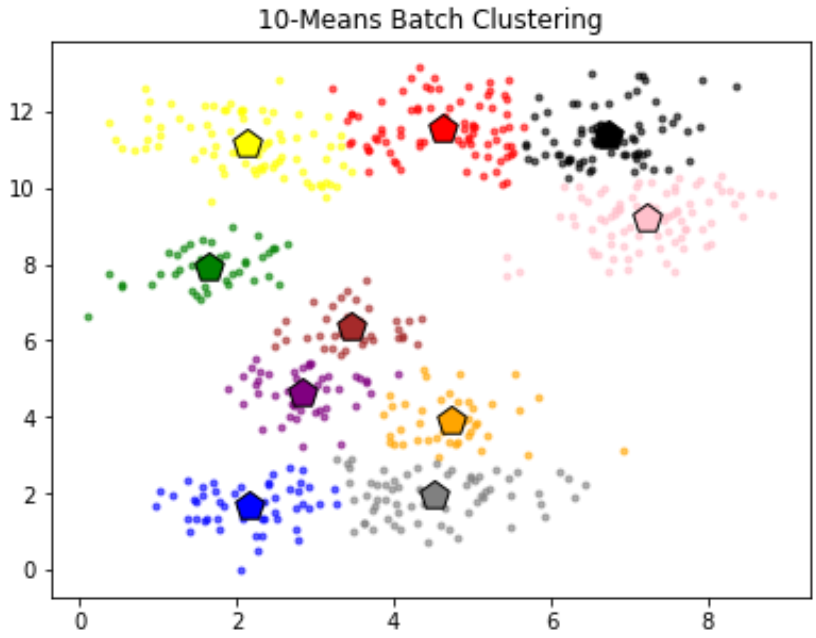

Figure 2: Scatter plot of K=10 batch algorithm

K=15 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 15, KMeans_batch, trial_number=1000, seed=50)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink", "cyan", "magenta", "lightgreen", "navy","crimson"]
plotClusters(clusters, centers, colors, "15-Means Batch Clustering", name="batch_15")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 315.4562129066001

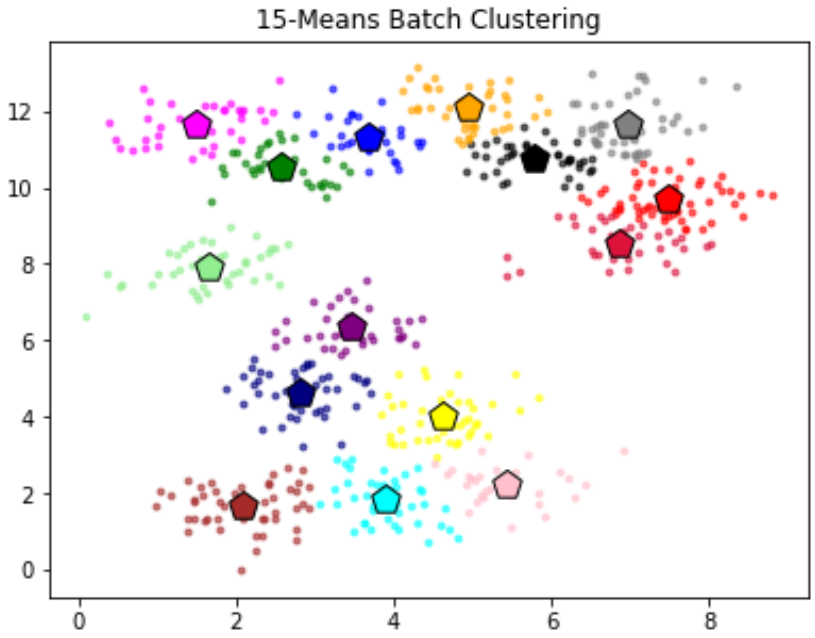

Figure 3: Scatter plot of K=15 batch algorithm

K-Means (online)

The K-Means online algorithm experimented for K=5, 10 and 15. Each experiment is run for 1000 times (since 10000 times running would take hours to complete), and the best result is shown.

The source code used to implement K-Means online algorithm:

```
def KMeans_online(patterns, cluster_number, alpha = 0.8, moment = 0.9, epsilon=0.01):
    I = cluster_number
    P = np.size(patterns,0)
    K = np.size(patterns,1)
    t = 0
    z = np.inf
    randIndices = np.random.choice(range(P), I, replace=False)
    w = np.copy(patterns)[randIndices]
    b = np.zeros((I,P))
    x = np.copy(patterns)
    while(alpha > epsilon):
        np.random.shuffle(x)
        b = np.zeros((I,P))
        w_tp = np.copy(w)
        for p in range(P):
            ip = np.linalg.norm(x[p] - w, axis=1).argmin()
            b[ip,p] = 1
            delta_w_ip = alpha * (x[p] - w[ip])
            w_tp[ip] = w[ip] + delta_w_ip
        z = 0
```

```
        assg_indices = np.argwhere(b > 0.5)
        for i in assg_indices:
            z += np.sum( (x[i[1]] - w[i[0]])**2 )
        alpha = moment * alpha
        t += 1
        w = w_tp
    return x, b, w, z
```

K=5 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 5, KMeans_online, trial_number = 1000, seed=440)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue"]
plotClusters(clusters, centers, colors, "5-Means Online Clustering", name="online_5")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 1214.5236656269215

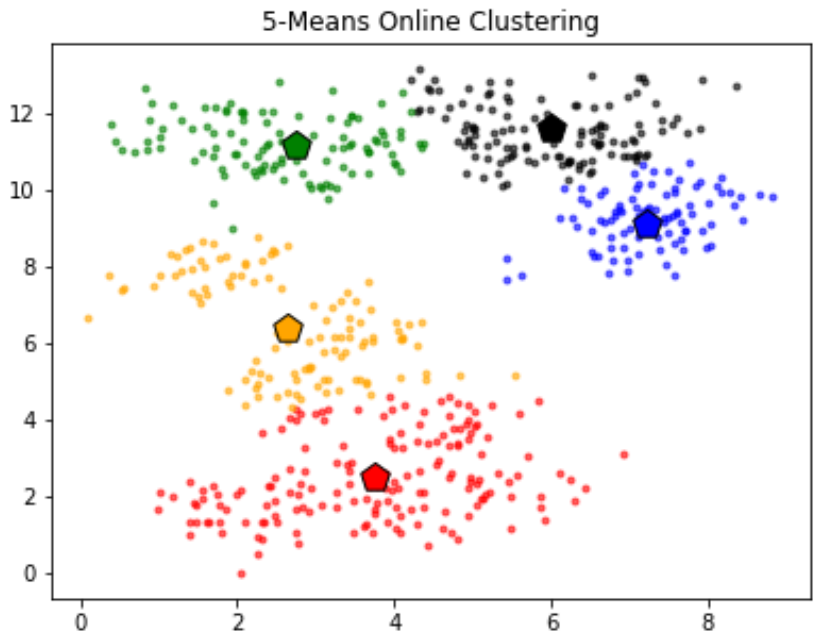

Figure 4: Scatter plot of K=5 online algorithm

K=10 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 10, KMeans_online, trial_number = 1000, seed=72)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink"]
plotClusters(clusters, centers, colors, "10-Means Online Clustering", name="online_10")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 511.6593649213377

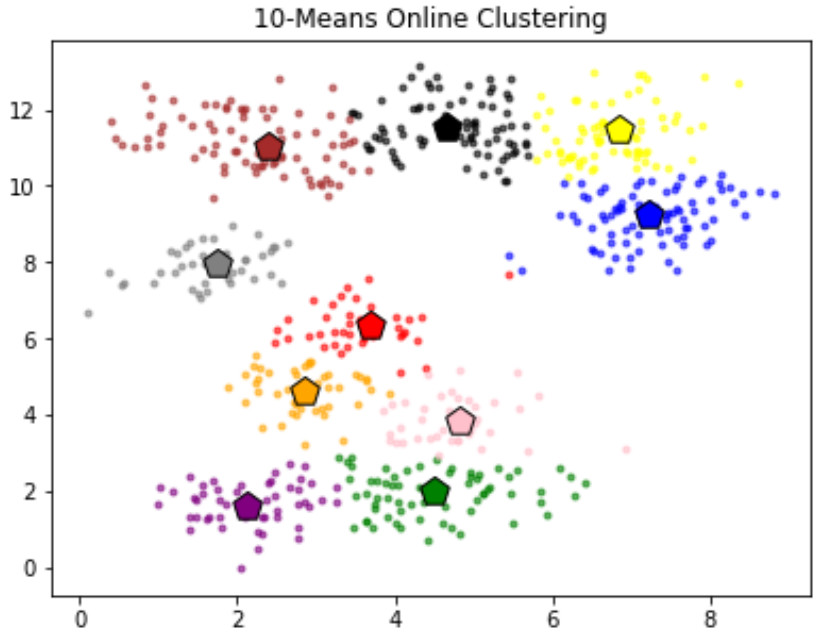

Figure 5: Scatter plot of K=10 online algorithm

K=15 Experiment

Source code:

```
x, b, w, z = findBestTrial(patterns, 15, KMeans_online, trial_number=1000, seed=50)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink", "cyan", "magenta", "lightgreen", "navy","crimson"]
plotClusters(clusters, centers, colors, "15-Means Online Clustering", name="online_15")
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 328.0408479032981

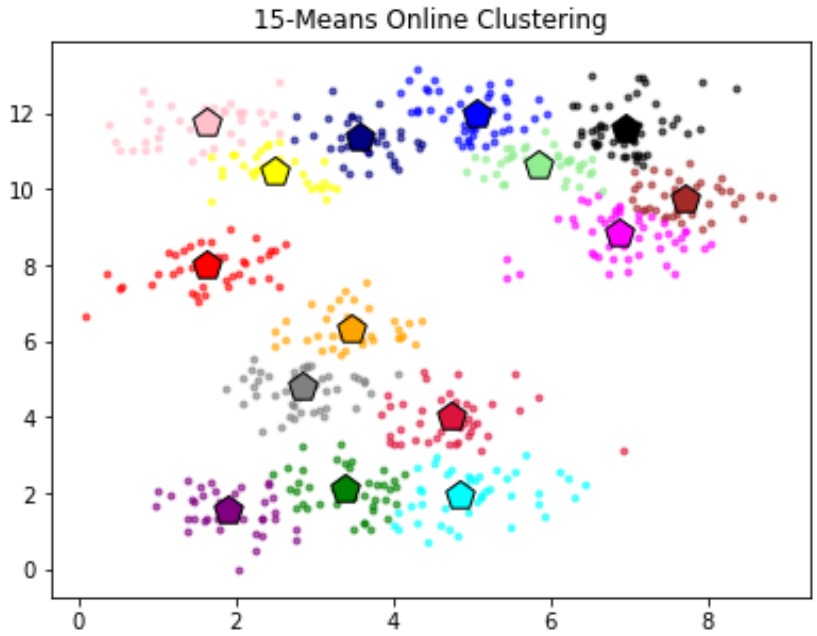

Figure 6: Scatter plot of K=15 online algorithm

SOM

The SOM algorithm experimented for K=5, 10 and 15. Algorithm is tested with different seed values and best results are shown here. The number of iterations is determined as $500 * I$ where I is the number of codebooks(K), and learning rate is decreased slowly for the first 1000 iterations, and then it decreased at a higher rate. Also σ is initialized large enough to cover all codebooks.

The source code used to implement SOM algorithm:

```
def SOM(patterns, cluster_number, alpha_init=0.95, beta=0.8, seed=440):
    np.random.seed(seed)
    I = cluster_number
    P = np.size(patterns,0)
    K = np.size(patterns,1)
    t = 0
    randIndices = np.random.choice(range(P), I, replace=False)
    w = np.copy(patterns)[randIndices]
    x = np.copy(patterns)
    alpha = alpha_init
    sigma = (x[:,0].max() - x[:,0].min()) / 2
    while(t < 500 * I):
        np.random.shuffle(x)
        for p in range(P):
            ip = np.linalg.norm(x[p] - w, axis=1).argmin()
            for i in range(I):
                gaussian_i = np.exp(-np.linalg.norm(w[i] - w[ip])**2 / sigma**2)
```

```
                delta_w_i = alpha * gaussian_i * (x[p] - w[i])
                w[i] += delta_w_i
        sigma = max(1e-8, beta * sigma) # To avoid divide by zero error
        if t < 1000:
            alpha = alpha_init * (1 - t / 2000)
        else:
            alpha = 0.999**t
        t += 1
    return x, w
```

Since the result of which point is assigned to which neuron and error value is not determined during the algorithm, below functions are used for that purpose:

```
def findClusterClasses(x, w):
    I = np.size(w, 0)
    P = np.size(x, 0)
    b = np.zeros((I,P))
    for p in range(P):
        ip = np.linalg.norm(x[p] - w, axis=1).argmin()
        b[ip,p] = 1
    return b

def calculateError(b, x, w):
    z = 0
    assg_indices = np.argwhere(b > 0.5)
    for i in assg_indices:
        z += np.sum( (x[i[1]] - w[i[0]])**2 )
    return z
```

K=5 Experiment

Source code:

```
x, w = SOM(patterns, 5, seed=24)
b = findClusterClasses(x, w)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue"]
plotClusters(clusters, centers, colors, "SOM with 5 Neurons Clustering", name="
    som_cluster_5")
z = calculateError(b, x, w)
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 1216.8968743643009

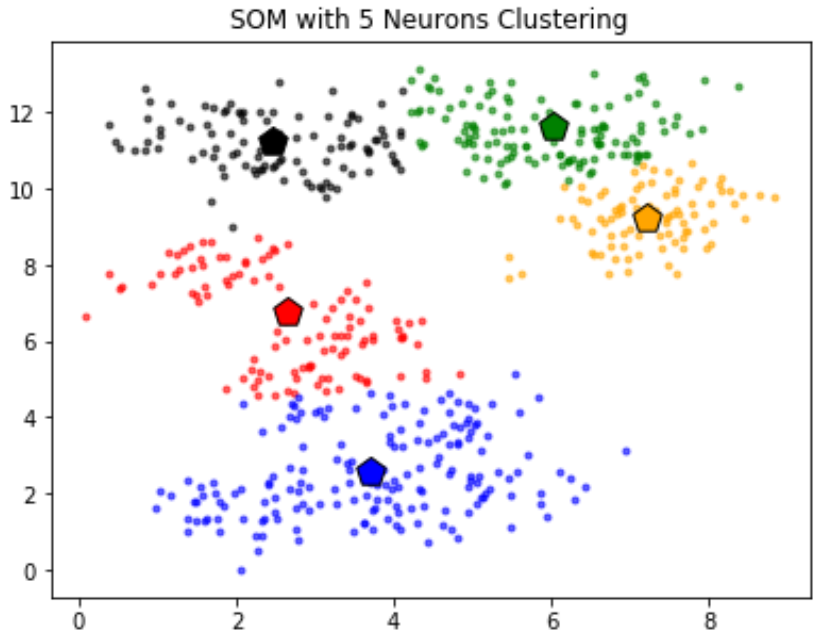

Figure 7: Scatter plot of K=5 SOM algorithm

K=10 Experiment

Source code:

```
x, w = SOM(patterns, 10, seed=440)
b = findClusterClasses(x, w)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink"]
plotClusters(clusters, centers, colors, "SOM with 10 Neurons Clustering", name="
    som_cluster_10")
z = calculateError(b, x, w)
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 502.2479051856347

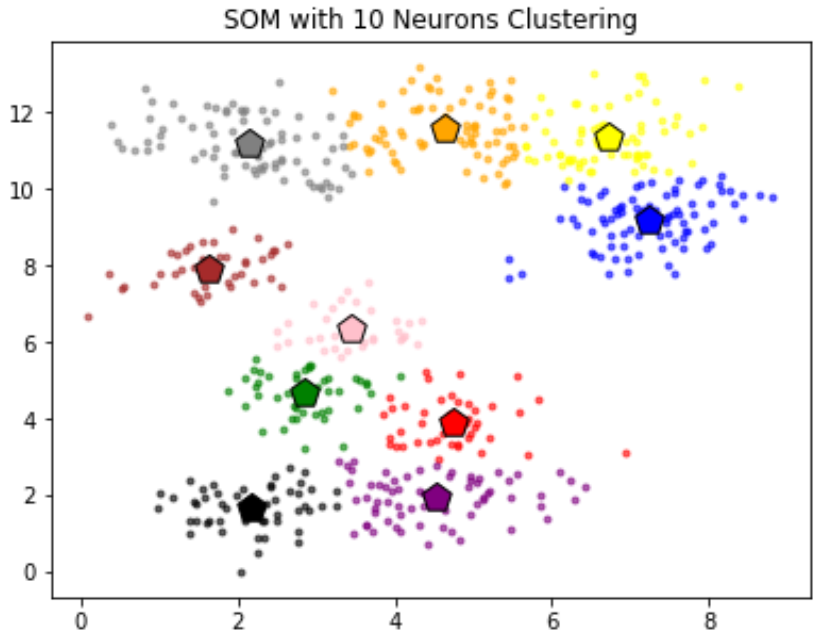

Figure 8: Scatter plot of K=10 SOM algorithm

K=15 Experiment

Source code:

```
x, w = SOM(patterns, 15, seed=50)
b = findClusterClasses(x, w)
clusters, centers = splitClusters(x, w, b)
colors = ["black", "red", "orange", "green", "blue", "yellow", "gray", "purple", "brown
    ", "pink", "cyan", "magenta", "lightgreen", "navy","crimson"]
plotClusters(clusters, centers, colors, "SOM with 15 Neurons Clustering", name="
    som_cluster_15")
z = calculateError(b, x, w)
print("Error value is: {0}".format(z))
```

Resulting output:

Error value is: 315.41906937658575

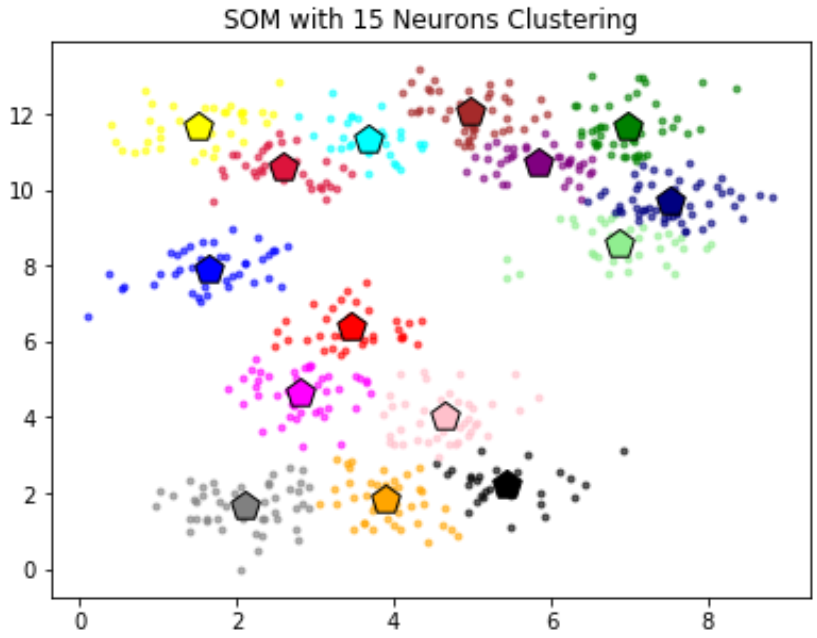

Figure 9: Scatter plot of K=15 SOM algorithm

Conclusion

The results of all three algorithms are shown in below table:

K	K-Means batch	K-Means online	SOM
5	1204.57	1214.52	1216.90
10	502.21	511.66	502.25
15	315.46	328.04	315.42

For all of the algorithms, the error value is reduces as K increases since we don't provide a cost for number of clusters. All three algorithms are converged to close results. Since we run K-means algorithms for 1000 times, the effect of randomness decreases for these two algorithms but SOM algorithm strongly depends on the initial locations.

As a result, all three algorithms found almost the same results and the same clusters.

3 SOM for the Euclidean Travelling Salesman Problem

In this section, the travelling salesman problem (TSP) is solved using a solver and SOM algorithms. First, the exact optimal solution is found with Gurobi solver. Then the SOM algorithm is used with

Gaussian kernel neighborhood calculation. Finally the SOM algorithm is used with elastic band neighborhood to find a good solution.

The function used to plot the TSP map with a given route:

```
def plotTSPMap(cities_data, route, title, figsize=[6.4, 4.8], name="graph"):
    plt.figure(figsize=figsize)
    plt.scatter(cities_data["x"], cities_data["y"], color="green", marker="o", label="
        Cities")
    if len(route) > 0:
        plt.plot(route[:,0], route[:,1], 'b-', label="Route")
        plt.plot(route[[-1,0],0], route[[-1,0],1], 'b-')
    plt.title(title)
    plt.legend()
    plt.savefig("{0}.png".format(name))
```

The source code used to read the data and plot the map:

```
cities_data = pd.read_csv('IE440Final19ETSPData.txt', sep=',', header=0);

tsp_patterns = np.array(cities_data[["x","y"]])

plotTSPMap(cities_data, [], "TSP Map", [15, 5])
```

The map of the cities with given coordinates:

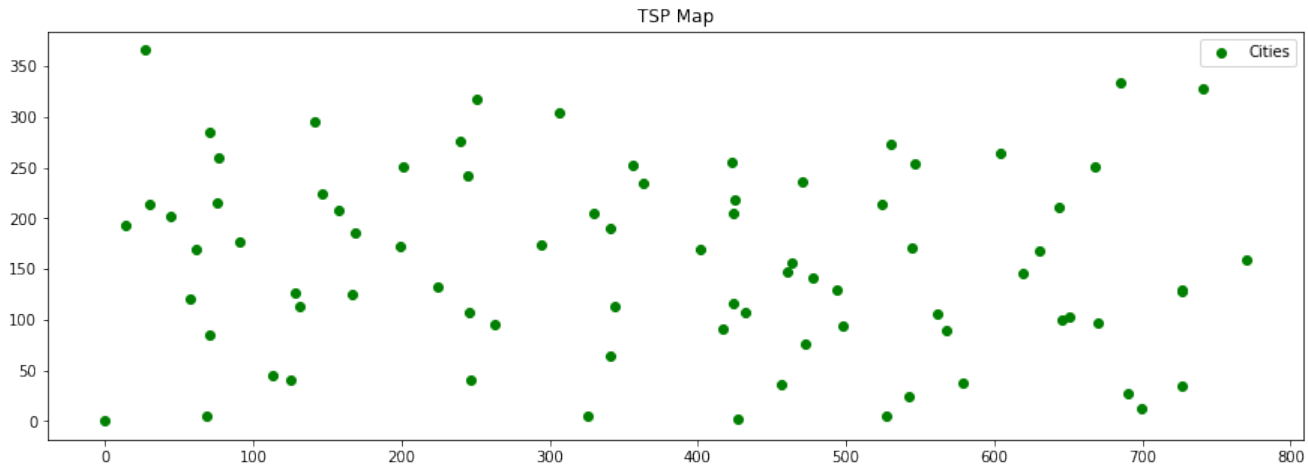

Figure 10: Scatter plot map of the cities

Some useful functions used to find the route from resulting codebooks and calculate the length of tour:

```
def totalLengthOfTour(route):
    total = 0
    shifted_route = np.insert(route[1:], 80, route[0], axis=0)
    distances = np.linalg.norm(route[:,:-1]-shifted_route[:,:-1], axis=1)
    return np.sum(distances)

def findRoute(x, w):
```

```
    I = np.size(w, 0)
    P = np.size(x, 0)
    route = np.insert(x, 2, 0, axis=1)
    for p in range(P):
        ip = np.linalg.norm(x[p] - w, axis=1).argmin()
        route[p,2] = ip
    return route[route[:,2].argsort()]
```

Exact Solution

An optimization model is constructed with given data and it's solved using Gurobi solver. Since Gurobi 8.0.1 is used as solver version, the source code is written in Python 2.7 and the results are saved to a file. The complete source code of the exact solution can be found in Appendix.

Since subtour elimination is an important problem in model construction and number of all permutations of subtours is very very large, a more efficient approach is used. Initially no subtour elimination constraint is added to model and the model is optimized. After that, the subtours formed in the solution are added to model as a constraint and the model is optimized again. After each solution, model is updated to eliminate the formed subtours by adding new constraints until no subtour remains. When the model is solved using that approach, the following results are obtained:

```
Coefficient statistics:
  Matrix range     [1e+00, 1e+00]
  Objective range  [1e+00, 8e+02]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 7e+01]
Presolved: 450 rows, 6480 columns, 19116 nonzeros

Continuing optimization...


Cutting planes:
  Gomory: 57
  Cover: 1
  MIR: 3
  Flow cover: 26
  Inf proof: 12
  Zero half: 91
  Mod-K: 1

Explored 243048 nodes (2350113 simplex iterations) in 0.02 seconds
Thread count was 4 (of 4 available processors)

Solution count 10: 3904.21 3904.21 3904.21 ... 3904.93

Optimal solution found (tolerance 1.00e-04)
Best objective 3.904212452201e+03, best bound 3.903849433126e+03, gap 0.0093%
gurobi>
```

Best objective : 3904.2124522

The source code used to read the solution sequence and plot the route:

```
def plotTSPExactSolution(cities_data, pairs, title, figsize=[6.4, 4.8], name="graph"):
    plt.figure(figsize=figsize)
    plt.scatter(cities_data["x"], cities_data["y"], color="green", marker="o", label="
        Cities")
    for p in range(len(pairs)):
        pair_coordinates = cities_data.loc[[pairs.loc[p]["i"],pairs.loc[p]["j"]]]
        plt.plot(pair_coordinates["x"], pair_coordinates["y"], "b-")
    plt.title(title)
    plt.savefig("{0}.png".format(name))

solution = pd.read_csv("solver_solution.csv", header=None, names = ["i", "j"])
plotTSPExactSolution(cities_data, solution, "TSP Exact Solution", [15,5], name="
    tsp_exact")
```

The optimal route map:

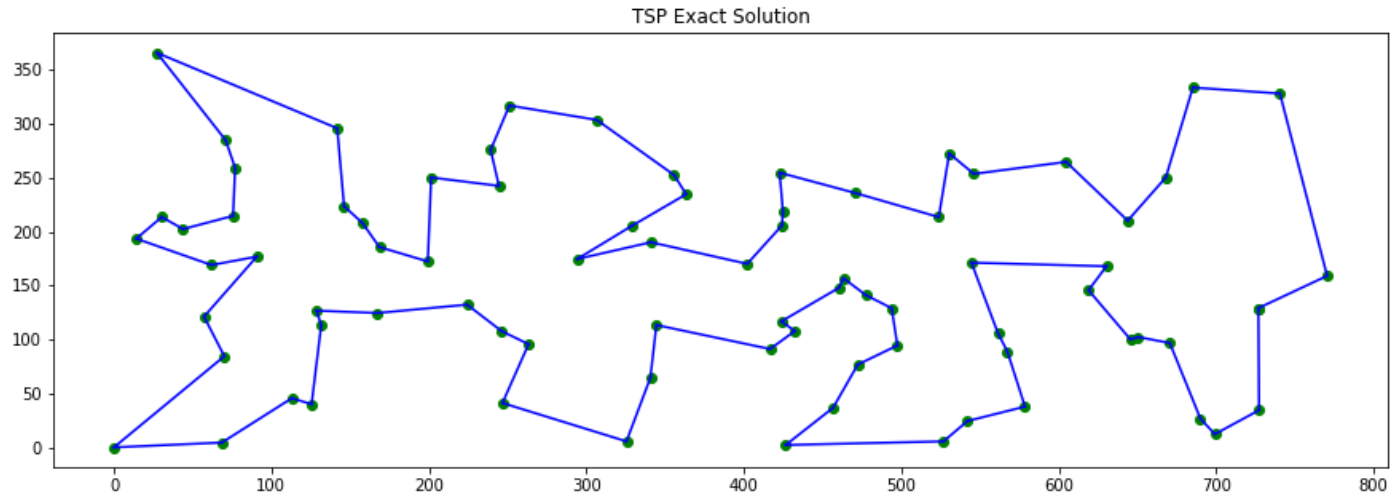

Figure 11: TSP exact solution route

In the following sections, problem is solved using SOM-TSP algorithm. The same algorithm function is used for both of them, but the neighborhood function is given as a parameter to the function so the results are different.

SOM-TSP Algorithm Implementation

Since the only difference is neighborhood calculation for the following parts, the same SOM-TSP function is used for the following parts and neighborhood function is given as a parameter.

The source code used to initialize codebooks inside the map randomly:

```
def initializeWeights(patterns, I):
    x_min = patterns[:,0].min()
    x_max = patterns[:,0].max()
```

```
    y_min = patterns[:,0].min()
    y_max = patterns[:,0].max()
    w = np.random.rand(I, 2)
    w[:,0] = x_min + w[:,0] * (x_max - x_min)
    w[:,1] = y_min + w[:,1] * (y_max - y_min)
    return w
```

The source code for SOM-TSP algorithm:

```
def SOM_TSP(patterns, neurons_number, neighborhood_func, iteration_number, alpha_init
    =0.9, beta=0.9, seed=440):
    np.random.seed(seed)
    I = neurons_number
    P = np.size(patterns,0)
    t = 0
    w = initializeWeights(patterns, I)
    x = np.copy(patterns)
    alpha = alpha_init
    sigma = np.sqrt( (x[:,0].max() - x[:,0].min())**2 + (x[:,1].max() - x[:,1].min())**2
        )
    while(t < iteration_number):
        np.random.shuffle(x)
        for p in range(P):
            ip = np.linalg.norm(x[p] - w, axis=1).argmin()
            for i in range(I):
                neighborhood = neighborhood_func(w, i, ip, I, sigma)
                delta_w_i = alpha * neighborhood * (x[p] - w[i])
                w[i] += delta_w_i
        sigma = max(1e-8, beta * sigma) # To avoid divide by zero error
        if t < 1000:
            alpha = alpha_init * (1 - t / 2000)
        else:
            alpha = 0.999**t
        t += 1
    return x, w
```

SOM-TSP with Gaussian Kernel

In this section, TSP is solved using SOM algorithm with Gaussian Kernel neighborhood calculation. Like the clustering SOM algorithm, learning rate is decreased slowly for the first 1000 iterations and σ is decreased very slowly using a β value close to 1 like 0.99. Also σ is initialized large enough to cover all codebooks.

The source code for Gaussian Kernel neighborhood calculation:

```
def gaussian_kernel(w, i, ip, I, sigma):
    gaussian = np.exp(-np.linalg.norm(w[i] - w[ip])**2 / sigma**2)
    return gaussian
```

The algorithm is run for M=81, 81*2 and 81*3 codebooks with 2000 iterations.

M=81 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81
x, w = SOM_TSP(tsp_patterns, M, gaussian_kernel, 2000, beta=0.99, seed=123)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))

plotTSPMap(cities_data, route, "SOM-TSP Gaussian with 81 Neurons", [15, 5], name="
    tsp_gauss_1")
```

Resulting output:

Total length of tour: 25215.55510993335

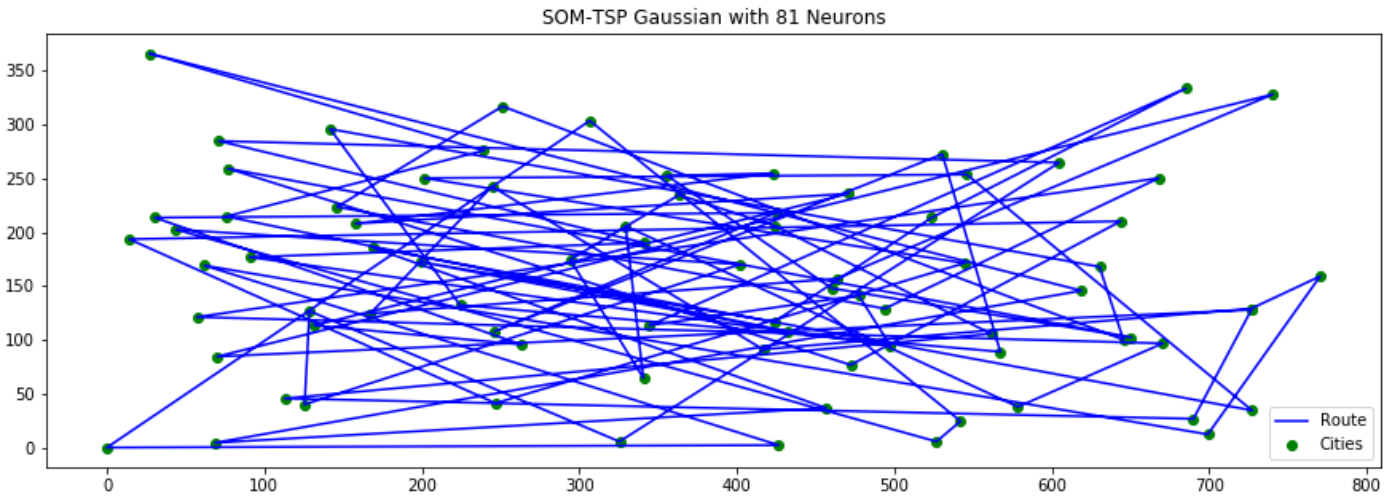

Figure 12: TSP Gaussian Kernel result with M=81

M=81*2 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81*2
x, w = SOM_TSP(tsp_patterns, M, gaussian_kernel, 2000, beta=0.99, seed=462)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))

plotTSPMap(cities_data, route, "SOM-TSP Gaussian with 81*2 Neurons", [15, 5], name="
    tsp_gauss_2")
```

Resulting output:

Total length of tour: 23201.053222172

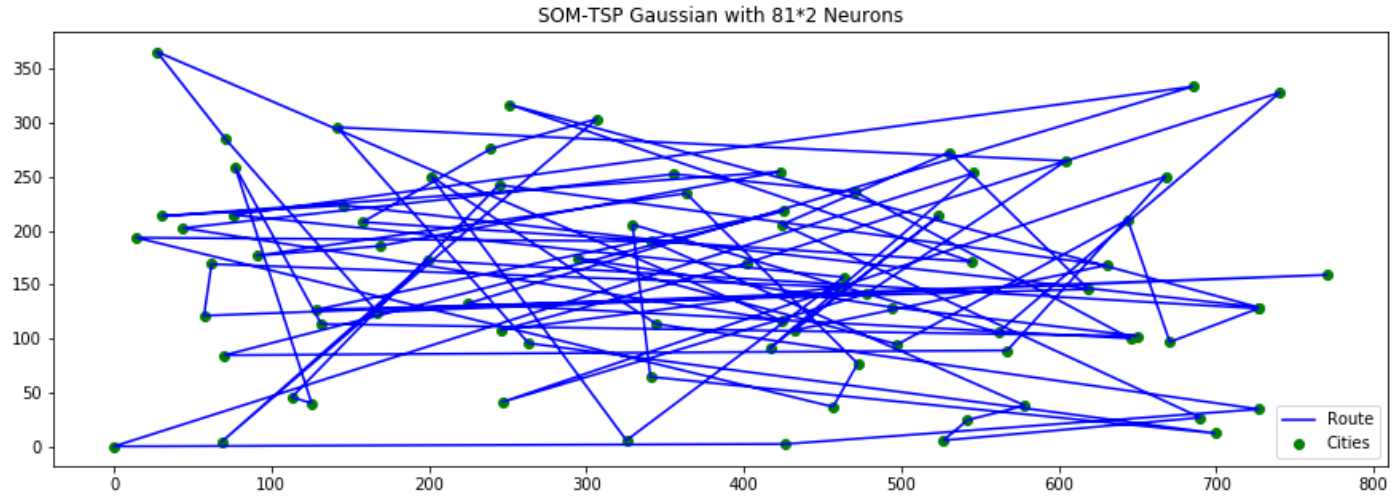

Figure 13: TSP Gaussian Kernel result with M=81*2

M=81*3 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81*3
x, w = SOM_TSP(tsp_patterns, M, gaussian_kernel, 2000, beta=0.99, seed=375)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))

plotTSPMap(cities_data, route, "SOM-TSP Gaussian with 81*3 Neurons", [15, 5], name="
    tsp_gauss_3")
```

Resulting output:

Total length of tour: 22794.005170326225

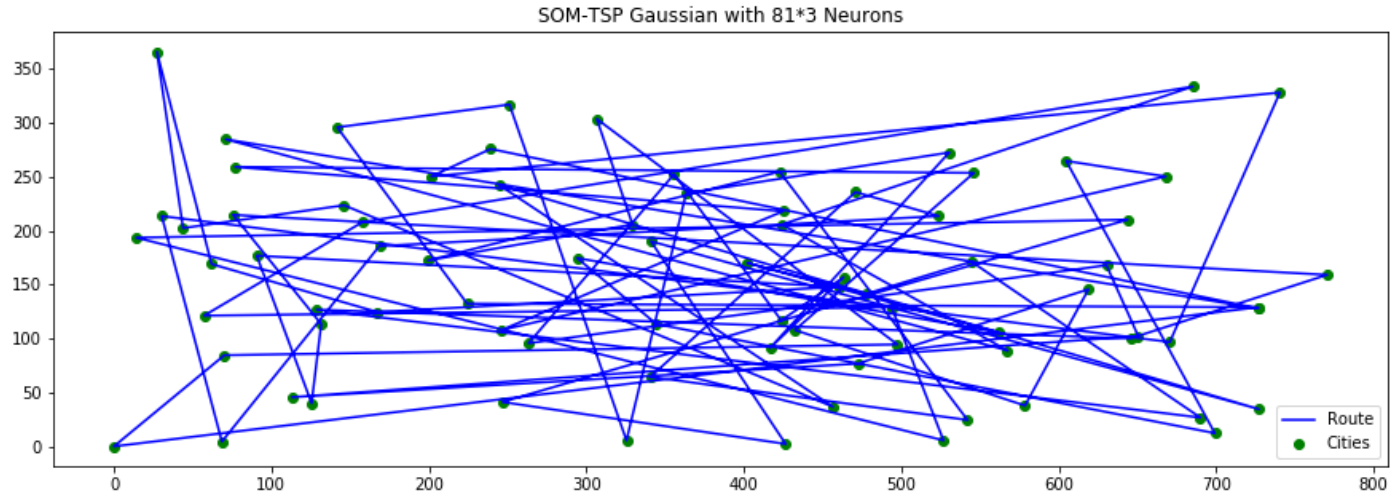

Figure 14: TSP Gaussian Kernel result with M=81*3

SOM-TSP with Elastic Band

In this section, TSP is solved using SOM algorithm with elastic band neighborhood calculation. Like the implementation in previous part, learning rate is decreased slowly for the first 1000 iterations and σ is decreased very slowly using a β value close to 1 like 0.99. Also σ is initialized large enough to cover all codebooks.

The source code for elastic band neighborhood calculation:

```
def elastic_band(w, i, ip, I, sigma):
    d = min(abs(i-ip), I - abs(i-ip))
    elastic_neigh = np.exp(-d**2 / sigma**2)
    return elastic_neigh
```

The algorithm is run for M=81, 81*2, 81*3, 81*6 and 81*10 codebooks with 2000 iterations.

M=81 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81
x, w = SOM_TSP(tsp_patterns, M, elastic_band, 2000, beta=0.99, seed=50)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81 Neurons", [15, 5], name="tsp_som_1")
```

Resulting output:

Total length of tour: 4004.128125547439

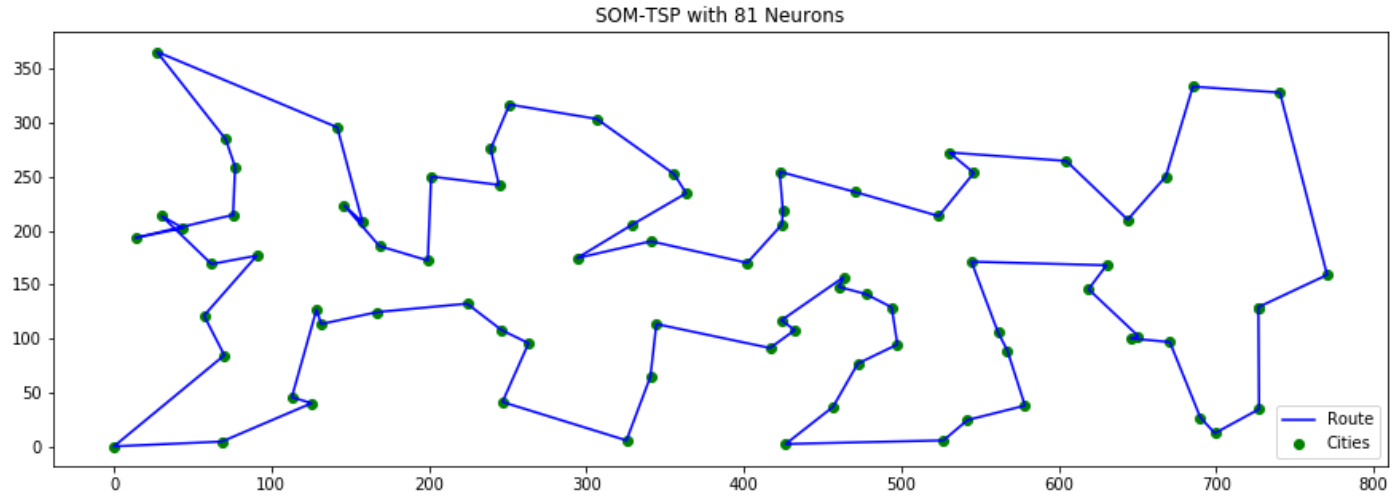

Figure 15: TSP elastic band result with M=81

M=81*2 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81*2
x, w = SOM_TSP(tsp_patterns, M, elastic_band, 2000, beta=0.99, seed=21)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81*2 Neurons", [15, 5], name="tsp_som_2")
```

Resulting output:

Total length of tour: 3904.2124522005647

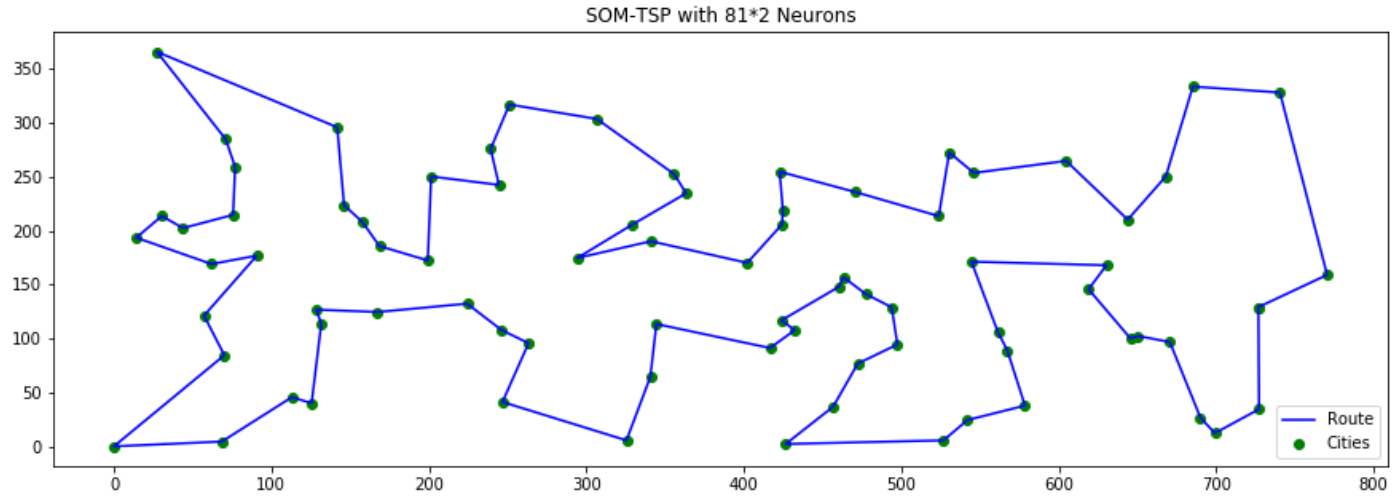

Figure 16: TSP elastic band result with M=81*2

M=81*3 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81*3
x, w = SOM_TSP(tsp_patterns, M, elastic_band, 2000, beta=0.99, seed=90)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81*3 Neurons", [15, 5], name="tsp_som_3")
```

Resulting output:

Total length of tour: 3925.028039886966

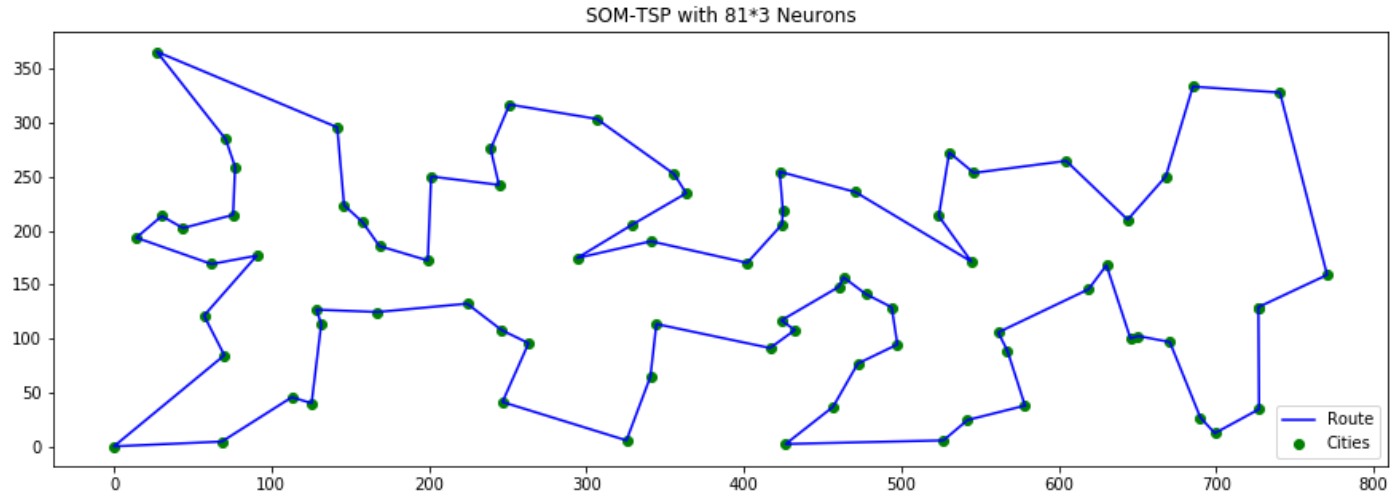

Figure 17: TSP elastic band result with M=81*3

M=81*6 Neurons Experiment

Source code to find the solution and plot the route:

```
M = 81*6
x, w = SOM_TSP(tsp_patterns, M, elastic_band, 2000, beta=0.99, seed=33)
route = findRoute(x, w)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81*6 Neurons", [15, 5], name="tsp_som_4")
```

Resulting output:

Total length of tour: 3904.2124522005643

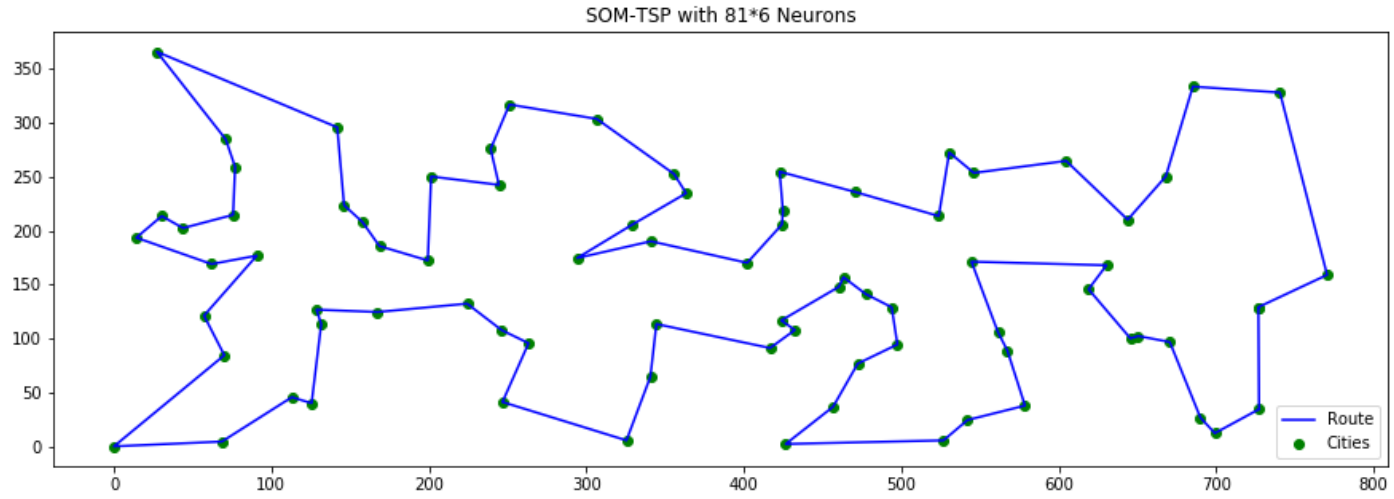

Figure 18: TSP elastic band result with M=81*4

M=81*10 Neurons Experiment

2 experiments with different β values are performed for M=81*10 neurons to show the effect of reducing rate of σ.

Experiment with $\beta = 0.99$

```
x1, w1 = SOM_TSP(tsp_patterns, 81*10, elastic_band, 2000, beta=0.99, seed=400)
route = findRoute(x1, w1)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81*10 Neurons", [15, 5], name="tsp_som_5")
```

Resulting output:

Total length of tour: 3947.5347539390987

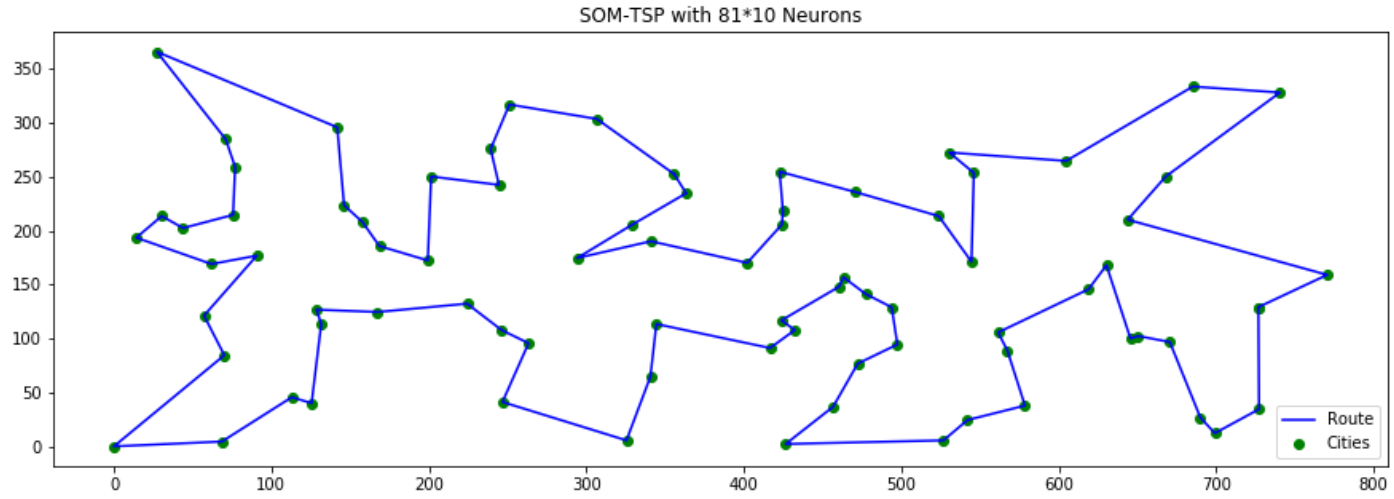

Figure 19: TSP elastic band result with M=81*10 and $\beta = 0.99$

Experiment with $\beta = 0.995$

```
x2, w2 = SOM_TSP(tsp_patterns, 81*10, elastic_band, 2000, beta=0.995, seed=400)
route = findRoute(x2, w2)
distance = totalLengthOfTour(route)
print("Total length of tour: {0}".format(distance))
plotTSPMap(cities_data, route, "SOM-TSP with 81*10 Neurons", [15, 5], name="tsp_som_6")
```

Resulting output:

Total length of tour: 3904.2124522005647

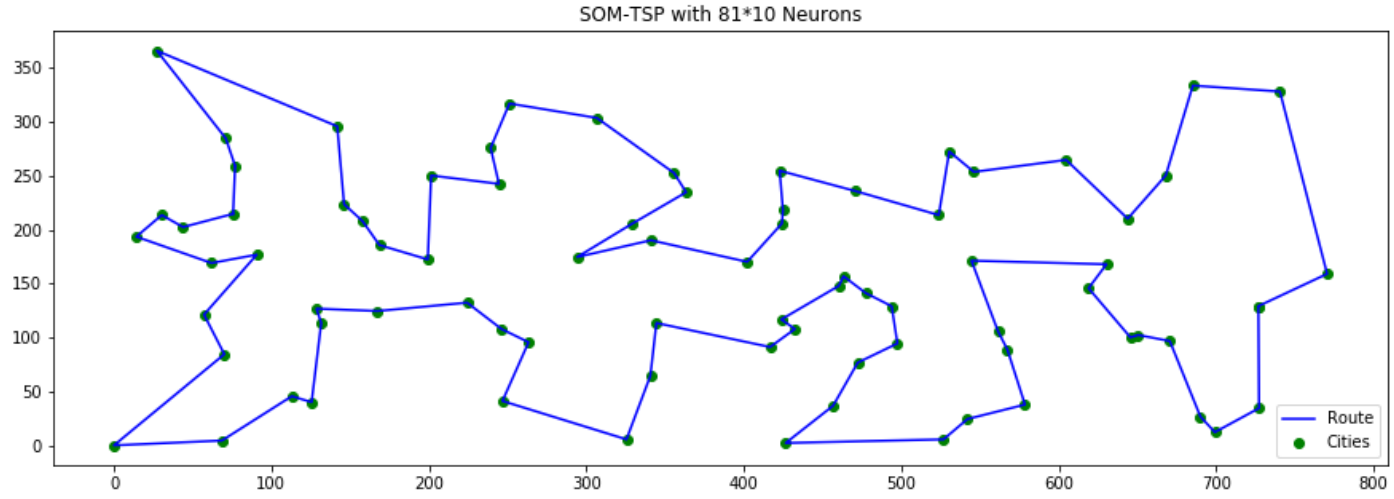

Figure 20: TSP elastic band result with M=81*10 and $\beta = 0.995$

Conclusion

SOM-TSP algorithm converged with elastic band neighborhood but it doesn't converged with Gaussian kernel neighborhood. Gaussian kernel neighborhood is not an appropriate neighborhood function for TSP. The reason is that, Gaussian kernel calculation doesn't consider the sequence of neurons, it only considers the locations of neurons. Therefore, two nonconsecutive neurons' neighborhood value might be very bigger than consecutive ones'. And when these neurons are returned as output, their order is completely independent from their neighborhood and we cannot find a proper route with their sequence.

The SOM algorithm with elastic band neighborhood returns very good results. Most of them are close to exact solution result and some of them are actually the same with exact solution. The worst result is obtained with M=81 neurons. The reason might be that, since 81 cities exists, some neurons might get closer to the same city so the number of neurons would be insufficient. For M=81*2 and M=81*6 with $\beta = 0.99$ the results are the same with exact solution (global optimum) while the result of M=81*3 is not same with it. The reason is the randomness of the initial locations because different seed values are used for experiments. For M=81*10, the algorithm gets closer to global optimum with $\beta = 0.99$ but it's not equal to global optimum. However when β is increased to 0.995 the result becomes equal to global optimum. It shows the effect of σ value in neighborhood calculation since it reduces slower when β is increased.

4 Appendix

Source code of the TSP Exact Solution with Gurobi solver:

```
# Solved with Gurobi 8.0.1
# Requires Python 2.7

import math
import csv
from itertools import combinations, permutations
from gurobipy import *


def euclideanDistance(point1, point2):
    return math.sqrt( (point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

def constructModel():
    m = Model()
    # Create binary variables
    vars = {}
    for i in range(81):
        for j in range(81):
            vars[i,j] = m.addVar(obj=c[i,j], vtype=GRB.BINARY,
```

```
                            name="x[{0}][{1}]".format(i,j))
        m.update()
    # Incoming and outgoing constraints
    for i in range(81):
        m.addConstr(quicksum(vars[i,j] for j in range(81)) == 1)
        vars[i,i].ub = 0
    for j in range(81):
        m.addConstr(quicksum(vars[i,j] for i in range(81)) == 1)
    m.update()
    return m, vars

def addSubtourConstraints(model, xs, subtours):
    for tour in subtours:
        s = len(tour.keys()) - 1
        model.addConstr(quicksum(xs[ind] for ind in list(tour.keys())) <= s)
    model.update()

def solveModel(model, xs):
    model.optimize()
    solution = model.getAttr('x', xs)
    # Get visited pairs
    pairs = []
    for i in range(81):
        for j in range(81):
            if solution[i,j] > 0.5:
                pairs.append([i,j])
    return m.objVal, pairs

def writePairsSolution(pairs, filename = 'solver_solution.csv'):
    with open(filename, 'w') as csv_file:
        writer = csv.writer(csv_file)
        for p in pairs:
            writer.writerow(p)


# Data read operation
tsp_patterns = []
with open('IE440Final19ETSPData.txt', mode='r') as csv_file:
    csv_reader = csv.DictReader(csv_file)
    for row in csv_reader:
        tsp_patterns.append([float(row["x"]),float(row["y"])])

# Cost calculation
c = {}

for i in range(81):
    for j in range(81):
        c[i,j] = euclideanDistance(tsp_patterns[i],tsp_patterns[j])

# Constructing the model and solving it
m, xs = constructModel()
obj, pairs = solveModel(m, xs)
pairs.sort()
# Find subroutes after each solution and
```

```
# add them to constraints until no subroute remains
while True:
    indices = {}
    for i in range(len(pairs)):
        indices[i] = 1
    subtours = []
    for k in range(len(indices)):
        if indices[k] == 0:
            continue
        tour = {}
        t = 0
        i = pairs[k][0]
        while t == len(tour.keys()):
            j = pairs[i][1]
            tour[i,j] = 1
            t+=1
            indices[i] = 0
            i = j
        if len(tour.keys()) < 81:
            subtours.append(tour)
    addSubtourConstraints(m, xs, subtours)
    obj, pairs = solveModel(m, xs)
    pairs.sort()
    if len(subtours) == 0:
        break

# Extract results as a csv file
writePairsSolution(pairs)
```