Introduction to Intelligent Systems
Lab Session 4

Group 17
Tobias Pucher (s5751659) & Thanakit Chaichanawong (s5752094)

October 11, 2023

Assignment 3

1. Introduction

The vector quantization is one of the popular way for reducing data. It is similar to clustering in someway. For simple explanation, the center point of the vector diving the section is called prototype vector which will be place randomly initially, then the learning based on distance will moving this point around until saturation. The saturation is when the vectors don't move much as it approach optimal point. The speed of movement is depending on learning rate, while learning time is depending on epochs. Epochs is basically number of time the algorithm will go through the data set. At the end of learning, we can obtain learning curve to see how effective is the algorithm. The data will be gathered in each epochs called "quantization error."

2. Methods

2.1. Initialization

```
def place_vector():
    indices = np.random.choice(len(data_array),prototype_no)
    prototype_vectors = data_array[indices]
    return prototype_vectors
```

This function allow the program to randomly select the points with function random.choice from numpy by returning indexes. The random.choice also select multiple point at once when given the index. Then the the indexes will be mapped to data and return from function.

2.2. In each epochs: Moving vectors

```
def move_vector(permuted_data,prototype_vectors):
    new_prototypes = np.copy(prototype_vectors)
    for i in range(len(permuted_data)):
      example_point = permuted_data[i]
      distances = []
      for proto in prototype_vectors:
        dist = np.linalg.norm(example_point - proto)
        distances.append(dist)
```

The move_vector function will responsible for moving the prototype vectors in one epoch. Beginning with copying previous prototype vectors to new array. In each epoch, the program will run through every single data points once in for-loop. The data involved are permuted outside this function. And in side the outer loop, inner loop will responsible to hold the distance to each prototype.

```
      winner_index = np.argmin(distances)
      winner = prototype_vectors[winner_index]

      step = learning_rate * (example_point - winner)
      new_prototypes[winner_index] = winner + step

    return new_prototypes
```

When distance is obtained, winner prototype will be calculated with np.argmin() to get the index of the data. Then the prototype will move toward the winner with the factor of learning_rate.

2.3. In each epochs: Quantization error calculation

```
def quantization_error(prototype_vectors):
    total_distances = 0.0
    for i in range(len(data_array)):
      dist = np.linalg.norm(data_array[i] - prototype_vectors)
      total_distances += pow(np.min(dist),2)
    return total_distances / len(data_array)
```

The next of this program is calculating the quantization error. Implementing np.linalg.norm to obtain the euclidean distance from each point to each vector in for-loop. In each iteration, the distance will be added to the total_distance by power of 2. Finally returning the total distance divided by the number of data.

2.4. Plotting

```
def plotting():
  fig = plt.figure(figsize=(10, 8))
  plot = fig.add_subplot()
  plot.scatter(data_array[:,0],data_array[:,1], c="blue", marker='o',
   label='Data Points')
  plot.scatter(prototype_vectors[:,0],prototype_vectors[:,1], c="red",
    marker='o', label='Data Points')
  plot.set_xlabel('X values')
  plot.set_ylabel('Y values')
  plot.set_title('Iteration: '+str(i))
  plt.show()
```

This function is dedicated to the plotting data points and prototype vectors with blue for data point and red for vectors.

2.5. main

```
  prototype_vectors = place_vector()
  trajectories = []
  Hvq = []
  for prot in prototype_vectors:
    traj = [[prot[0]], [prot[1]]]
    trajectories.append(traj)
```

When running the program, trajectories and quantization arrays will keep record of corresponding value for later used. The for-loop prot aimed to created path of trajectories depending on number of prototype vector.

```
  for epoch_idx in range(max_epochs):
    permuted_data = np.random.permutation(data_array)
    new_prototypes = move_vector(permuted_data,prototype_vectors)
    prototype_vectors=new_prototypes

    for i, prot in enumerate(prototype_vectors):
      trajectories[i][0].append(prot[0])
      trajectories[i][1].append(prot[1])

    Hvq.append(quantization_error(prototype_vectors)
```

In this part of main, the for-loop representing number of epochs. In each epochs, data will be randomized with function np.random.permutation before calling move_vector. The return value of move_vector are the new prototype vector, which will be used in next epochs. And finally, changes of prototype vectors and quantization error of each epochs will be record to the respective arrays.

```
  plt.figure(figsize=(10, 6))
  plt.plot(range(1, max_epochs + 1), Hvq, label='Vector Quantization
   Error', color='green')
  plt.xlabel('Iterations')
  plt.ylabel('Vector Quantization Error')
  plt.title('Vector Quantization Error over Iterations')
  plt.legend()
  plt.grid(True)
  plt.show()
```

After every epochs, the program will plotting the learning curve with data from the array of Hvq.

```
def plot_trajectories(data_array, trajectories, max_epochs):
  fig = plt.figure(figsize=(10, 8))
  plot = fig.add_subplot()
  plot.scatter(data_array[:,0],data_array[:,1], c="blue", marker='o',
   label='Data Points')
  plot.set_xlabel('X values')
  plot.set_ylabel('Y values')
  plot.set_title('Trajectories for Epochs: '+str(max_epochs))

  for i,t in enumerate(trajectories):
    x=t[0]
    y=t[1]
    plot.plot(x, y, marker='.', label=f'Prototype Trajectory of Prot {
   i}')

  fig.legend()
  fig.show()
plot_trajectories(data_array, trajs, max_epochs)3
```

Finally, the trajectories method will constructe the trajectories of each prototype vectors throughout each epochs from trajectories array. For-loop iterate number of time corresponding to the number of prototype vector.

3. Result

3.1. Learning curves experiment 1

Figure 1: *Experiment K=2 Learning rate=0.1 over 20 iterations*

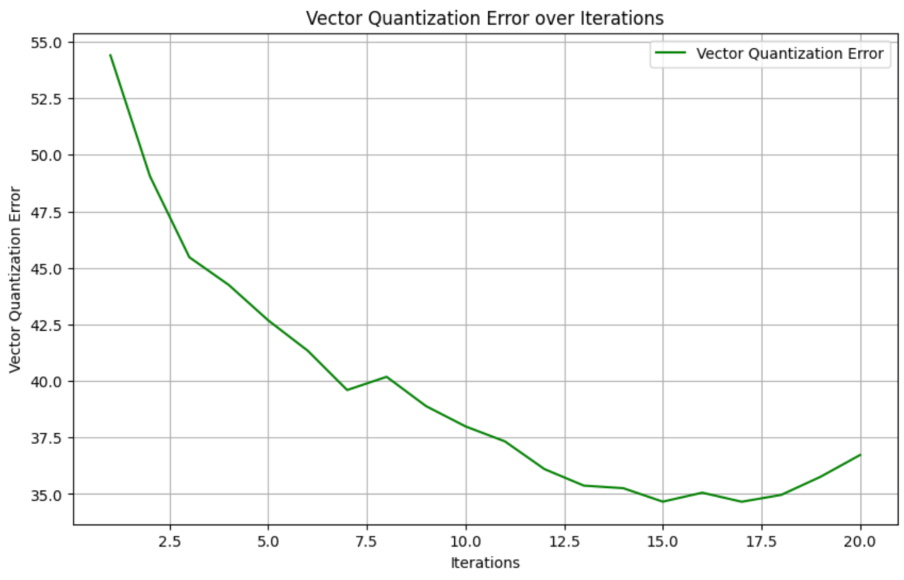


Figure 2: *Experiment K=2 Learning rate=0.03 over 100 iterations*

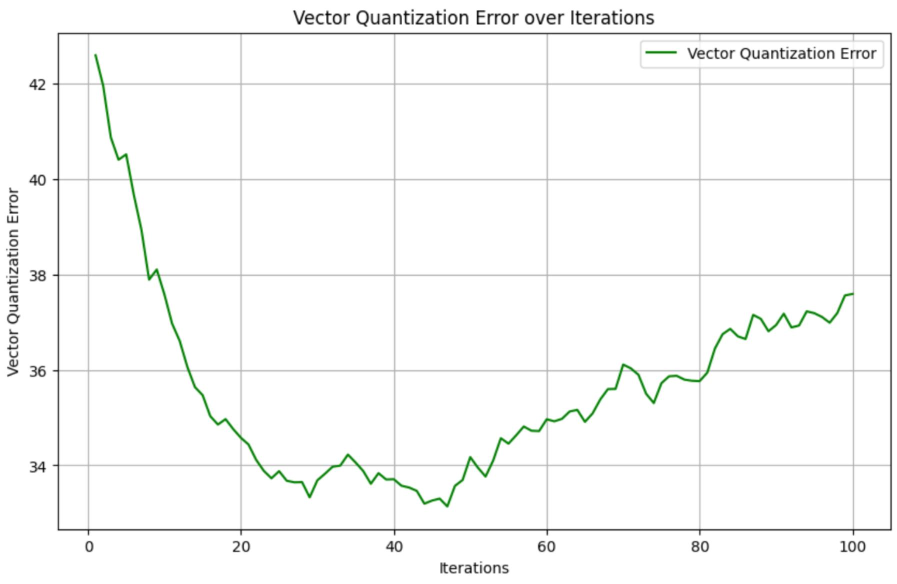


Figure 3: *Experiment K=2 Learning rate=0.01 over 150 iterations*

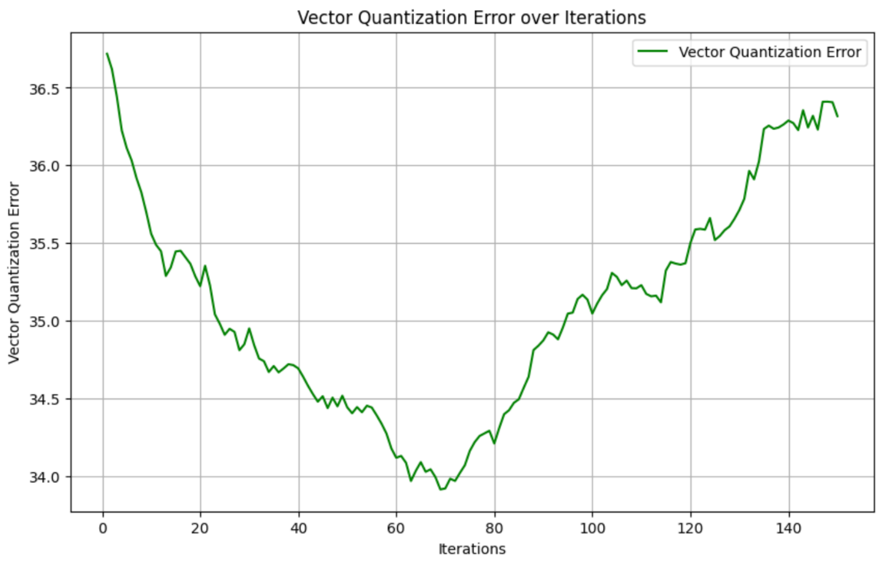


Figure 4: *Experiment K=4 Learning rate=0.1 over 100 iterations*

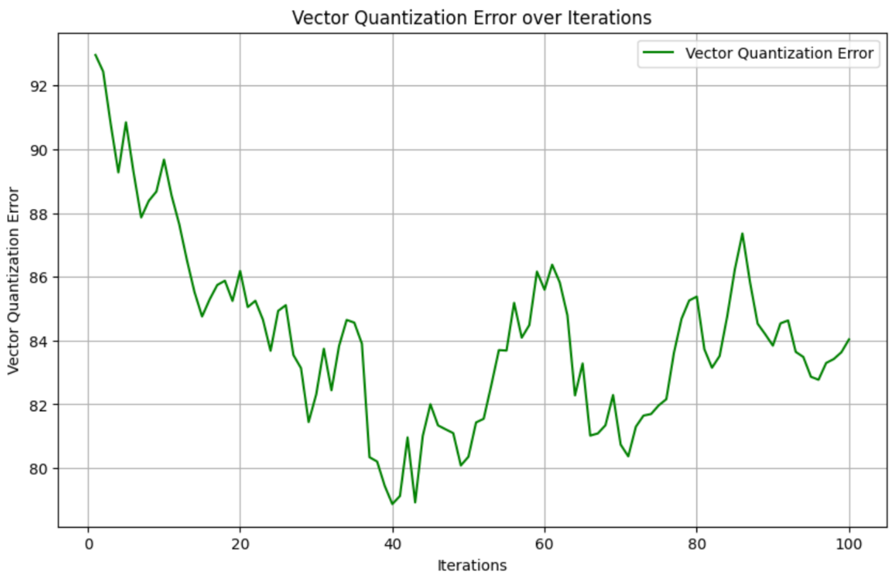


Figure 5: *Experiment K=4 Learning rate=0.03 over 100 iterations*

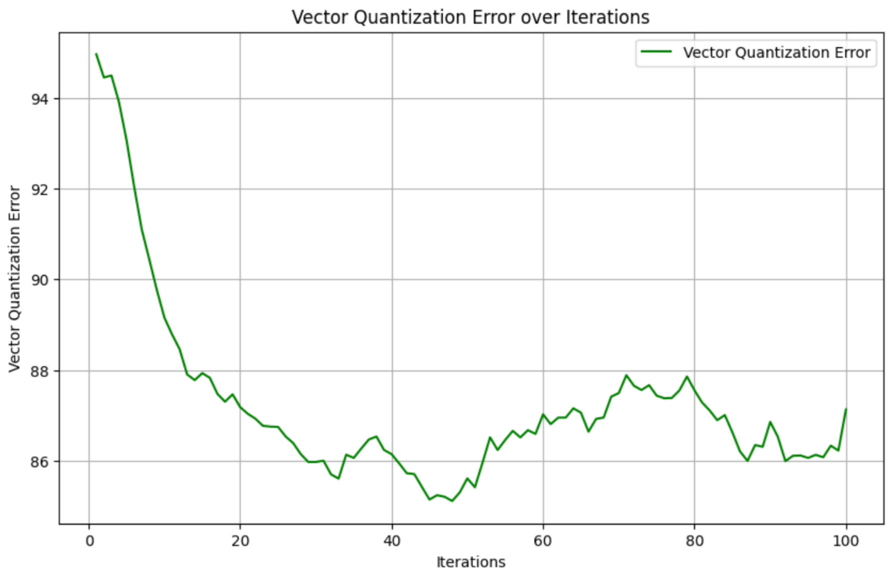
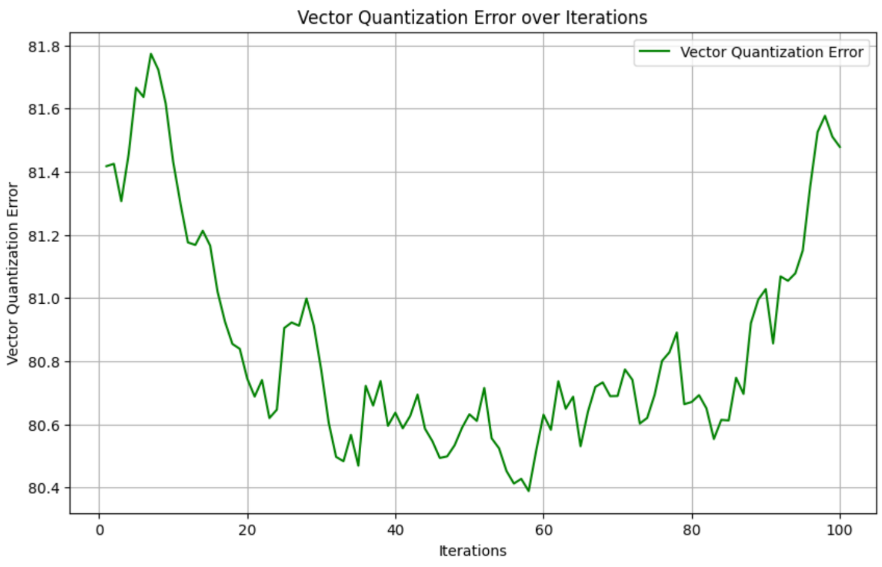
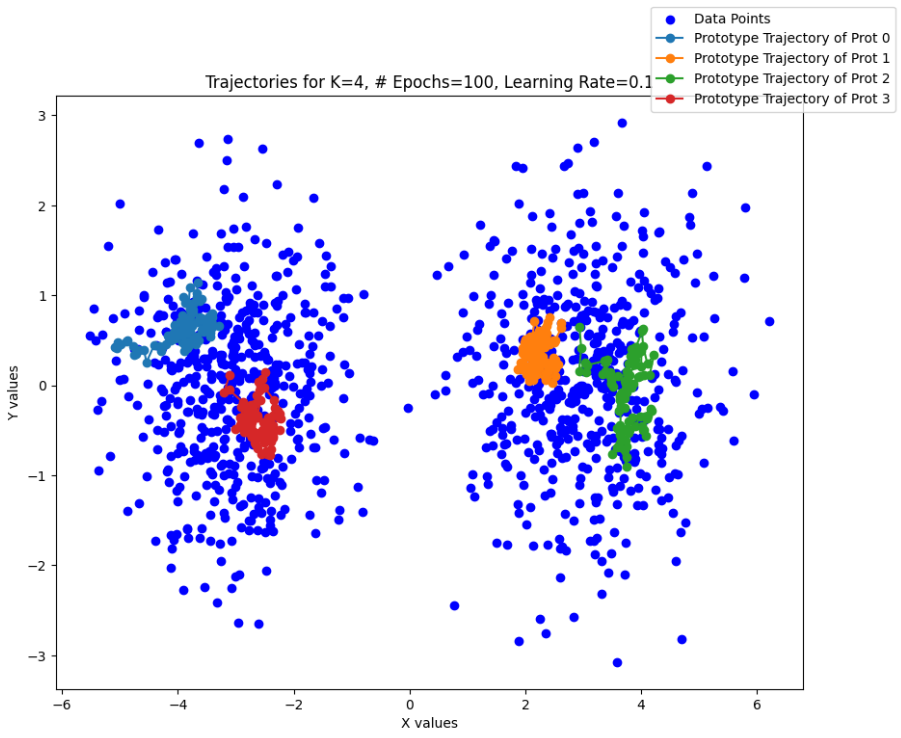
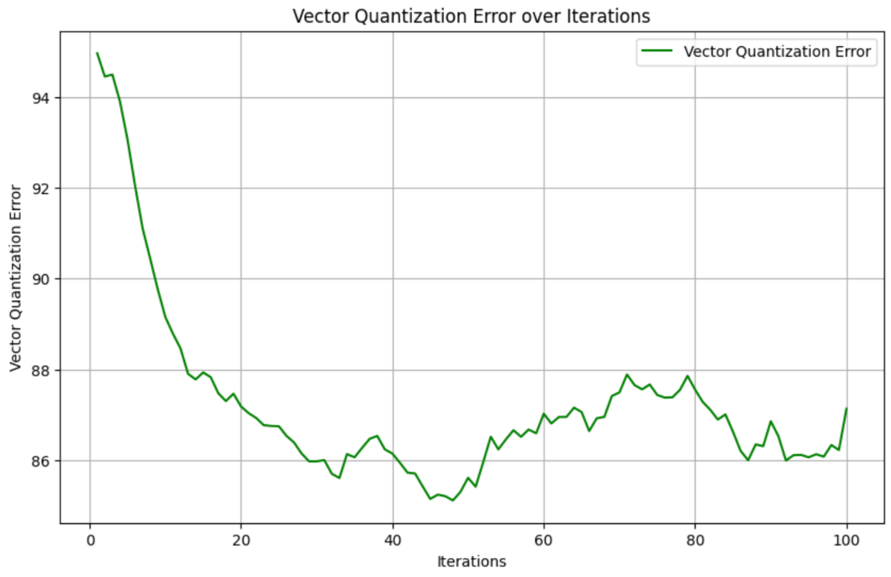


Figure 6: *Experiment K=4 Learning rate=0.01 over 100 iterations*

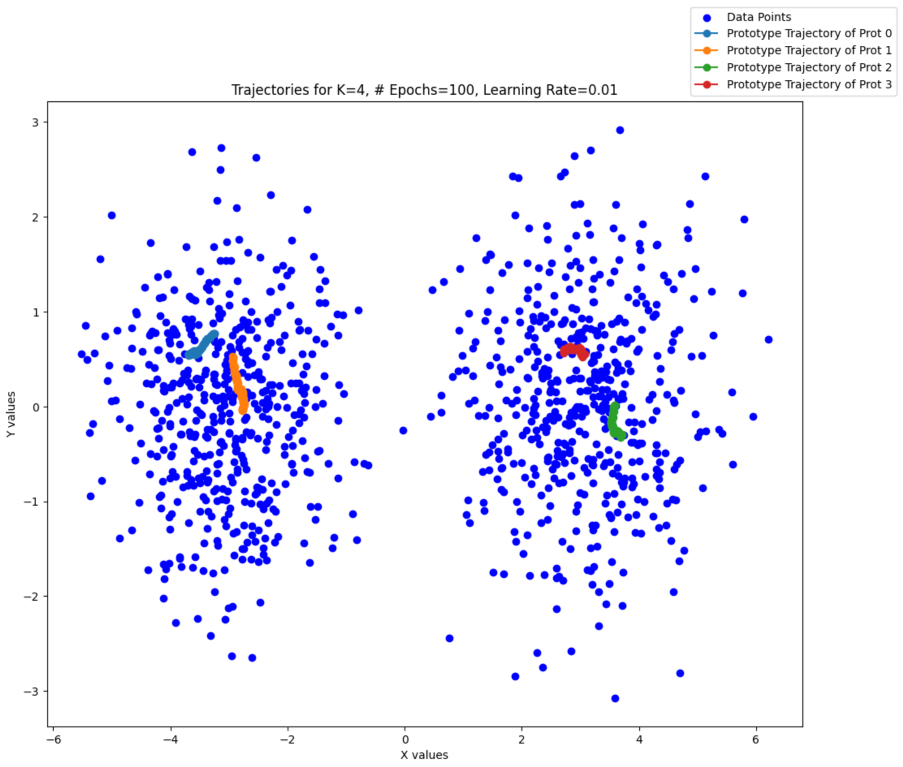


3.2. Trajectories prototypes experiment 2

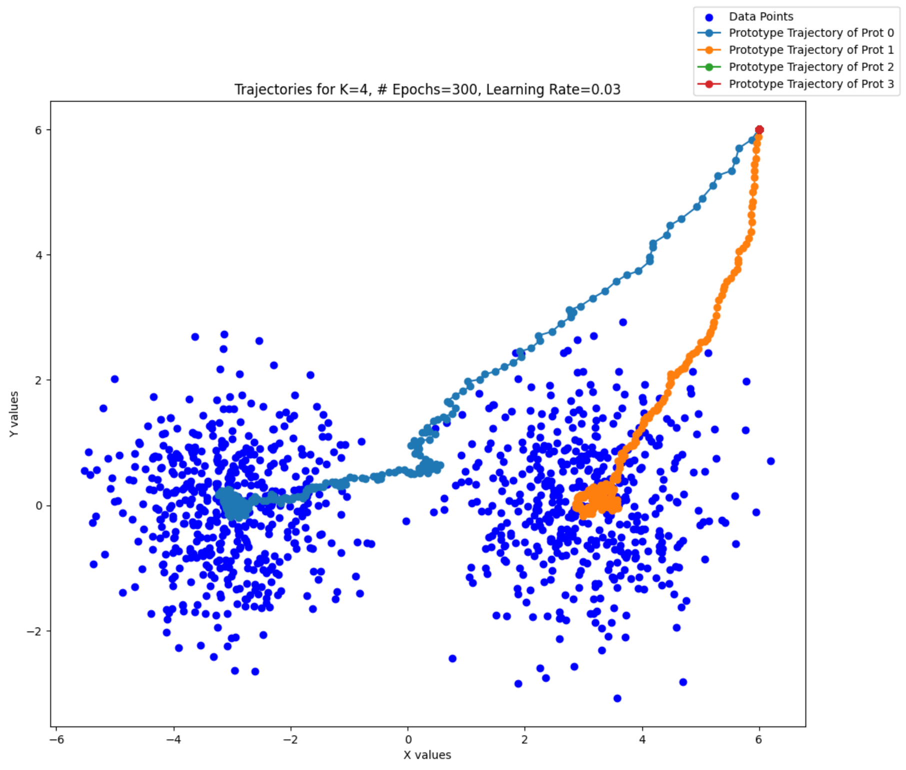




Vector Quantization Error over Iterations
Vector Quantization Error
Vector Quantization Error
Iterations
94
92
90
88
86
0
20
40
60
80
100



Data Points
Prototype Trajectory of Prot 0
Prototype Trajectory of Prot 1
Prototype Trajectory of Prot 2
Prototype Trajectory of Prot 3
Trajectories for K=4, # Epochs=100, Learning Rate=0.01
Y values
X values
3
2
1
0
−1
−2
−3
−6
−4
−2
0
2
4
6



4. Discussion

The number of epochs strongly depends on the learning factor. During our experiments we noticed that it is often its very hard to predict a correct number of epochs without any reference. We noticed that when the points get moved the error is oftentimes at its minimum between the two clusters, which would be the optimal minimum. But because the algorithm continues the point moves further into the cluster of points past the local minimum of error and then travels around in the cluster without much change. For a high learning rate of 0.1 with a relatively low number of epochs (e.g 20) a very good result can be achieved, but also potential local minima get skipped. This can be seen in Figure 1. A trade-off would be a lower learning factor for example 0.01 in Figure 3 which finds other possible local error minima when the prototypes get moved when already inside the cluster or between the clusters. But also as already mentioned the local minima that exist between two clusters which are from an error perspective very important. In the case of the two clusters in the data the global minima for K=2 actually lies between the two clusters but the algorithm approaches a local minima that exists when the two points are each in one of the clusters. So all in all if the points start inside the clusters then the minimum that potentially lies when the prototypes are between the clusters are never reached, but when both start near each other then a better local minima is explored/passed during the learning phase.

For K=4 different effects can be noticed. The plot Figure **??** shows the update steps effect on the prototypes in a meaningful way. Because all prototypes started in the same cluster (by random) the closest (always the same) prototype gets moved by all of the example points from the opposite cluster, resulting in a nice learning path / trajectory of this particular prototype. During this learning period the other prototypes stay relatively static inside of the cluster and approach other more insignificant local error minima.

For a K=4 when multiple of the prototypes are near one of the cluster then some might get drawn towards the other cluster, therefore making the learning rate and epoch number even more important. In this case with random choose of prototype initial position often times 3 of the 4 points initially get placed new the same cluster. This placement will never reach (but pass) the global minimum but rather approach some local minimum of error which is fine because its not intended by the algorithm as its implemented now.

For K=4 we also tried a less smart initial prototype placement as seen in Figure **??**. The configuration used was placing all 4 prototypes in the top right corner of the data set at position $[6.0, 6.0]$. The epoch was also tripled to 300 epochs. Because of the inner workings of the algorithm in the beginning all examples attract the same prototype. The next prototype gets moved only when the first was moved far enough and so on. This is very inefficient and after 300 epochs the last two prototypes were not even moved once. This shows that a statistically better placement like random choice is far superior than to a fixed and equal placement. Of course also fixed values might make sense but with this would need other smart algorithms to choose such.

During our experiments we had thoughts about storing the overall local minima configuration of prototypes when a new local minima is encountered, which would be storage efficient and could be of interest when learning is done, to revisit such configurations. These configurations can be very different from the expected "clusters".

5. Contributions

We worked bot hon the code as well as the final report in a 50:50 split.