exercise3

December 25, 2020

1 EXERCISE 3 - ML - Grundverfahren

1.1 1.) Constrained Optimization (6 Points)

You are given the following Optimization problem:

$$\min_{x} x^T M x + x^T h$$
$$s.t. x^T b \geq c,$$

where M is a positive definit, symmetric Matrix. Note that vectors and matrices are boldsymbol, where Matrices have capital letters. Derive the optimal solution for x independant of the Lagrangian multiplier(s) (i.e. you have to solve for the dual). Make sure that you mark vectors and matrices as a boldsymbol and small letters and capital letters respectively. Symbols which are not marked as boldsymbols will count as scalar. Take care of vector/matrix multiplication and derivatives. And make use of the properties of M. Don't forget to look up matrix-vector calculus in the matrix cookbook, if you don't remember the rules.

Lagrange dual problem:

$$\min_{x} L(x, \lambda) = x^T M x + x^T h - \lambda * (x^T b - c), s.t. \lambda >= 0$$

calculate the derivation of the lost funtion and set to 0:

$$\frac{\partial L(x, \lambda)}{\partial x} = 2Mx + h - \lambda * b \overset{!}{=} 0, \quad \lambda >= 0$$

$$x^* = \frac{1}{2} M^{-1}(\lambda b - h)$$

set the x into the lagrange dual problem $L(x, \lambda)$, we have

$$\min_{x} L(x, \lambda) = \frac{1}{4}(\lambda b - h)^T M^{-1}(\lambda b - h) + \frac{1}{2}(\lambda b - h)^T M^{-1} h - \frac{1}{2}\lambda * (\lambda b - h)^T M^{-1} b - \lambda c$$

calculate the derivation of the lost funtion and set to 0:

$$\frac{\partial L(x, \lambda)}{\partial x} = \frac{1}{2} M^{-1}(\lambda b - h) + \frac{1}{2} b^T M^{-1} h + b^T M^{-1} b \lambda - c \overset{!}{=} 0$$

we get the value of λ and x:

$$\lambda = (\frac{1}{2}M^{-1}b + b^T M^{-1}b)^{-1}(c + \frac{1}{2}M^{-1}h - \frac{1}{2}b^T M^{-1}h)$$

$$x = \frac{1}{2}M^{-1}[(\frac{1}{2}M^{-1}b + b^T M^{-1}b)^{-1}(c + \frac{1}{2}M^{-1}h - \frac{1}{2}b^T M^{-1}h) * b - h]$$

1.2 2.) k-Means (7 Points)

Here we will implement one of the most basic appraoches to clustering - the k-Means algorithm.
Let us start with some basic imports and implementing functionallity to visualize our results.

```
[1]: %matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from typing import Tuple, Optional

def visualize_2d_clustering(data_points: np.ndarray, assignments_one_hot: np.
 ↪ndarray, centers: np.ndarray, k: int,
                            centers_history: Optional[np.ndarray] = None, title:␣
 ↪Optional[str] = None):
    """Visualizes clusters, centers and path of centers"""
    plt.figure(figsize=(6, 6), dpi=100)
    assignments = np.argmax(assignments_one_hot, axis=1)

    for i in range(k):
        # get next color
        c = next(plt.gca()._get_lines.prop_cycler)['color']
        # get cluster
        cur_assignments = assignments == i
        # plot clusters
        plt.scatter(data_points[cur_assignments, 0],␣
 ↪data_points[cur_assignments, 1], c=c,
                    label="Cluster {:02d}".format(i))

        #plot history of centers if it is given
        if centers_history is not None:
            plt.scatter(centers_history[:, i, 0], centers_history[:, i, 1],␣
 ↪marker="x", c=c)
            plt.plot(centers_history[:, i, 0], centers_history[:, i, 1], c=c)

    plt.scatter(centers[:, 0], centers[:, 1], label="Centers", color="black",␣
 ↪marker="X")

    if title is not None:
```

2

```
        plt.title(title)

    plt.legend()
```

Next we going to implement the actual algorithm. As a quick reminder, K-Means works by iterating the following steps:

Start with k randomly picked centers

- 1.) Assign each point to the closest center
- 2.) Addjust centers by taking the average over all points assigned to it

Implementing them will be your task for this exericse

```
[2]: def assignment_step(data_points: np.ndarray, centers: np.ndarray) -> np.ndarray:
         """
         Assignment Step: Computes assignments to nearest cluster
         :param data_points: Data points to cluster  (shape: [N x data_dim])
         :param centers: current cluster centers (shape: [k, data_dim])
         :return Assignments (as one hot) (shape: [N, k])
         """
         #########################################################
         # TODO Implement the assignment step of the k-Means algorithm
         N, k = data_points.shape[0], centers.shape[0]
         one_hot_output = np.zeros((N, k))
         nor_inner = lambda x, y: np.inner(x, y) / (np.linalg.norm(x) * np.linalg.
      →norm(y))
         for n, data in enumerate(data_points):
             dist_list = [nor_inner(data, cens) for cens in centers]
             one_hot_output[n] = [i== max(dist_list) for i in dist_list]
         return one_hot_output
         #########################################################


     def adjustment_step(data_points: np.ndarray, assignments_one_hot: np.ndarray) ->␣
      →np.ndarray:
         """
         Adjustment Step: Adjust centers given assignment
         :param data_points: Data points to cluster  (shape: [N x data_dim])
         :param assignments_one_hot: assignment to adjust to (one-hot representation)␣
      →(shape: [N, k])
         :return Adjusted Centers (shape: [k, data_dim])
         """

         #########################################################
         # TODO Implement the adjustment step of the k-Means algorithm
         new_centers = np.zeros((assignments_one_hot.shape[1], data_points.shape[1]))
         for col in range(assignments_one_hot.shape[1]):
```

```
        # new_centers[col, :] = 0
        one_hot = np.tile(assignments_one_hot[:, col].T, (2, 1)).T
        new_centers[col, :] = np.sum(data_points * one_hot, axis=0) / np.
 ↪sum(one_hot, axis=0)
    return new_centers
    ##########################################################
```

Now to the final algorithm, as said we initialize the centers with random data points and iterate
the assignmenent and adjustment step

```
[3]: def k_means(data_points: np.ndarray, k: int, max_iter: int = 100, vis_interval:␣
     ↪int = 3) -> \
         Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
     """
     Simple K Means Implementation
     :param data_points: Data points to cluster  (shape: [N x data_dim])
     :param k: number of clusters
     :param max_iter: Maximum number of iterations to run if convergence is not␣
     ↪reached
     :param vis_interval: After how many iterations to generate the next plot
     :return: - cluster labels (shape: [N])
              - means of clusters (shape: [k, data_dim])
              - SSD over time (shape: [2 * num_iters])
              - History of means over iterations (shape: [num_iters, k, data_dim])
     """
     # Bookkeeping
     i = 0
     means_history = []
     ssd_history = []
     assignments_one_hot = np.zeros(shape=[data_points.shape[0], k])
     old_assignments = np.ones(shape=[data_points.shape[0], k])

     # Initialize with k random data points
     initial_idx = np.random.choice(len(data_points), k, replace=False)
     centers = data_points[initial_idx]
     means_history.append(centers.copy())

     # Iterate while not converged and max number iterations not reached
     while np.any(old_assignments != assignments_one_hot) and i < max_iter:
         old_assignments = assignments_one_hot

         # assignment
         assignments_one_hot = assignment_step(data_points, centers)

         # compute SSD
         diffs = np.sum(np.square(data_points[:, None, :] - centers[None, :, :]),␣
     ↪axis=-1)
```

```
            ssd_history.append(np.sum(assignments_one_hot * diffs))

            # adjustment
            centers = adjustment_step(data_points, assignments_one_hot)

            # compute SSD
            diffs = np.sum(np.square(data_points[:, None, :] - centers[None, :, :]),
    ↪axis=-1)
            ssd_history.append(np.sum(assignments_one_hot * diffs))


            # Plotting
            if i % vis_interval == 0:
                visualize_2d_clustering(data_points, assignments_one_hot, centers,
    ↪k, title="Iteration {:02d}".format(i))

            # Bookkeeping
            means_history.append(centers.copy())
            i += 1

        print("Took", i, "iterations to converge")
        return assignments_one_hot, centers, np.array(ssd_history), np.
    ↪stack(means_history, 0)
```

Finally we run the dataset and visualize the results. Here we provide 4 random datasets, each containing 500 2 samples and you can play around with the number of clustes, k, as well as the seed of the random number generator. Based on this seed the initial centers, and thus the final outcome, will vary.

```
[4]: np.random.seed(42)

data = np.load("samples_3.npy")
k = 8

cluster_labels, centers, ssd_history, centers_history = k_means(data, k)

# plot final clustering with history of centers over iterations
visualize_2d_clustering(data, cluster_labels, centers, k=k,
 ↪centers_history=centers_history, title="Final Clustering")

# plot SSD
plt.figure("SSD")
plt.semilogy(np.arange(start=0, stop=len(ssd_history) / 2, step=0.5),
 ↪ssd_history)
plt.xlabel("Iteration")
plt.ylabel("SSD")
plt.show()
```
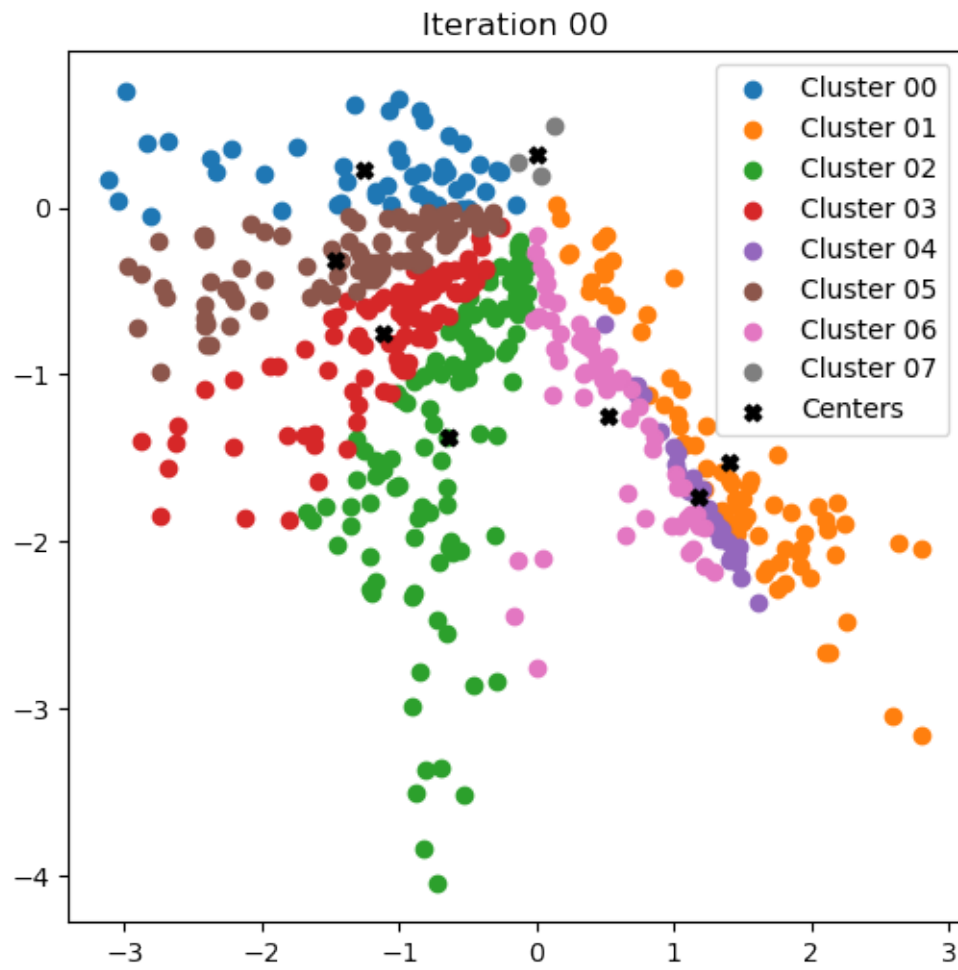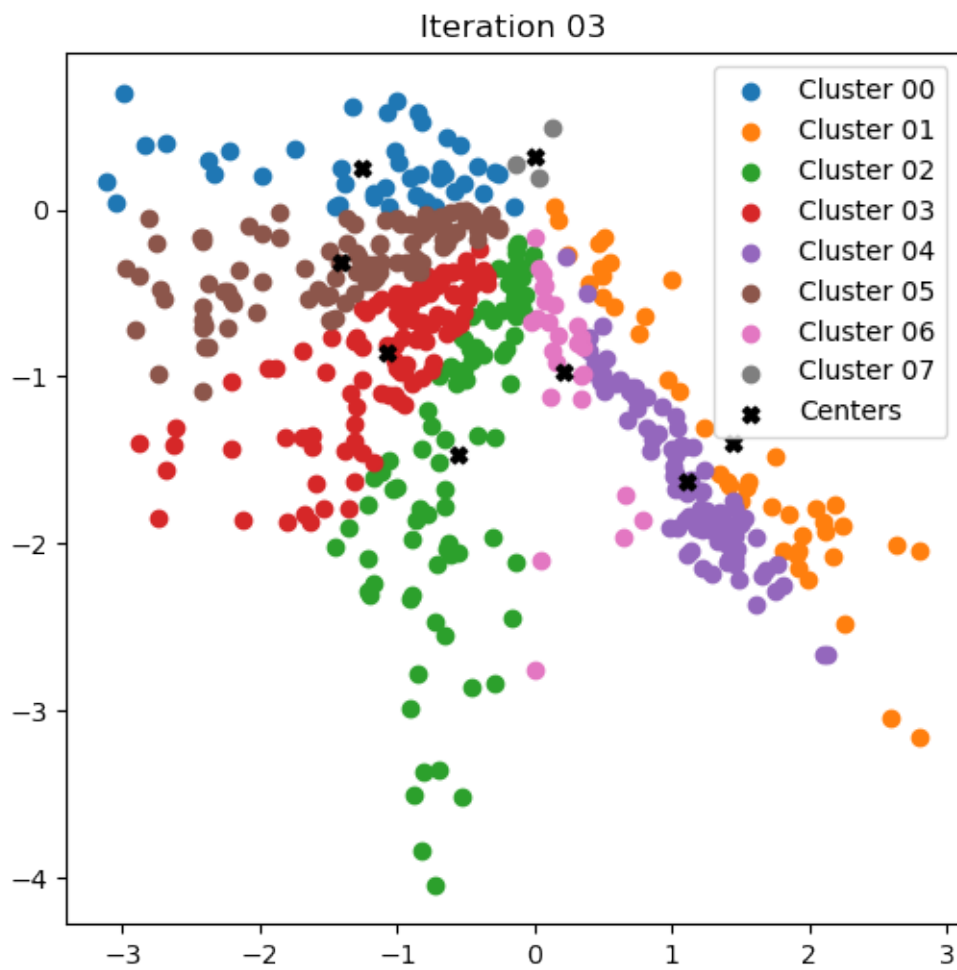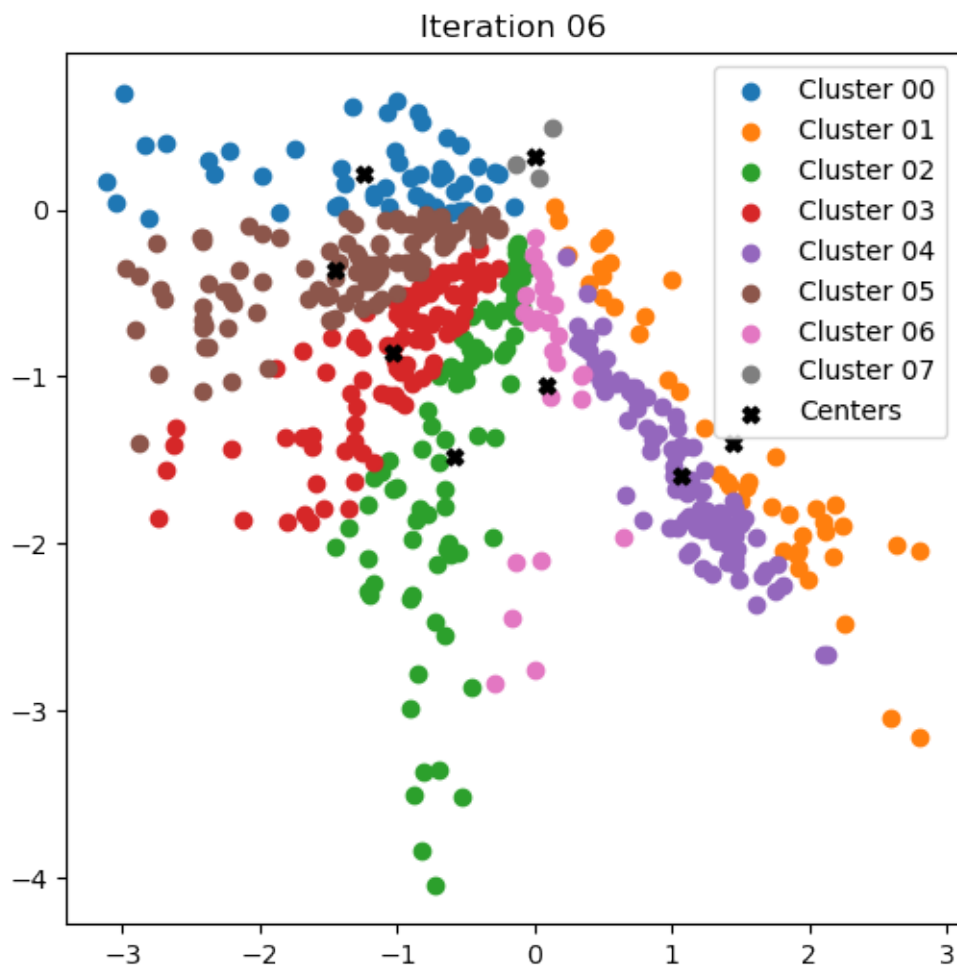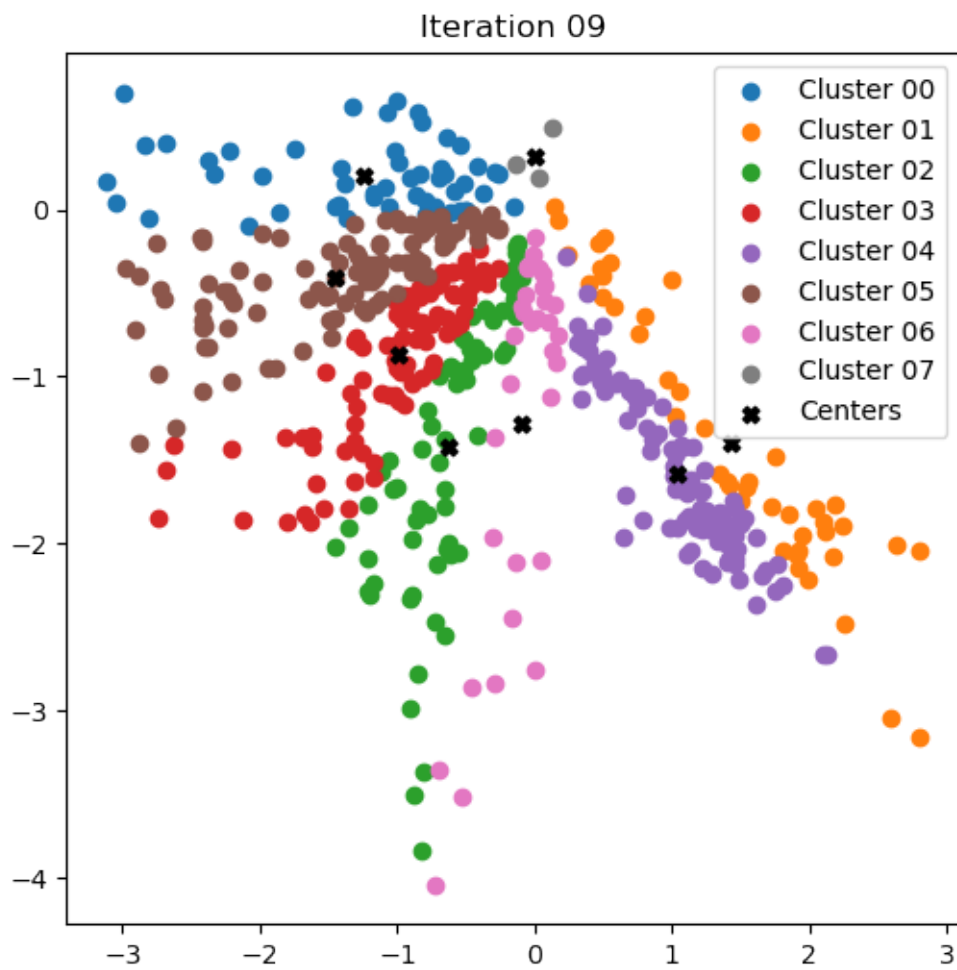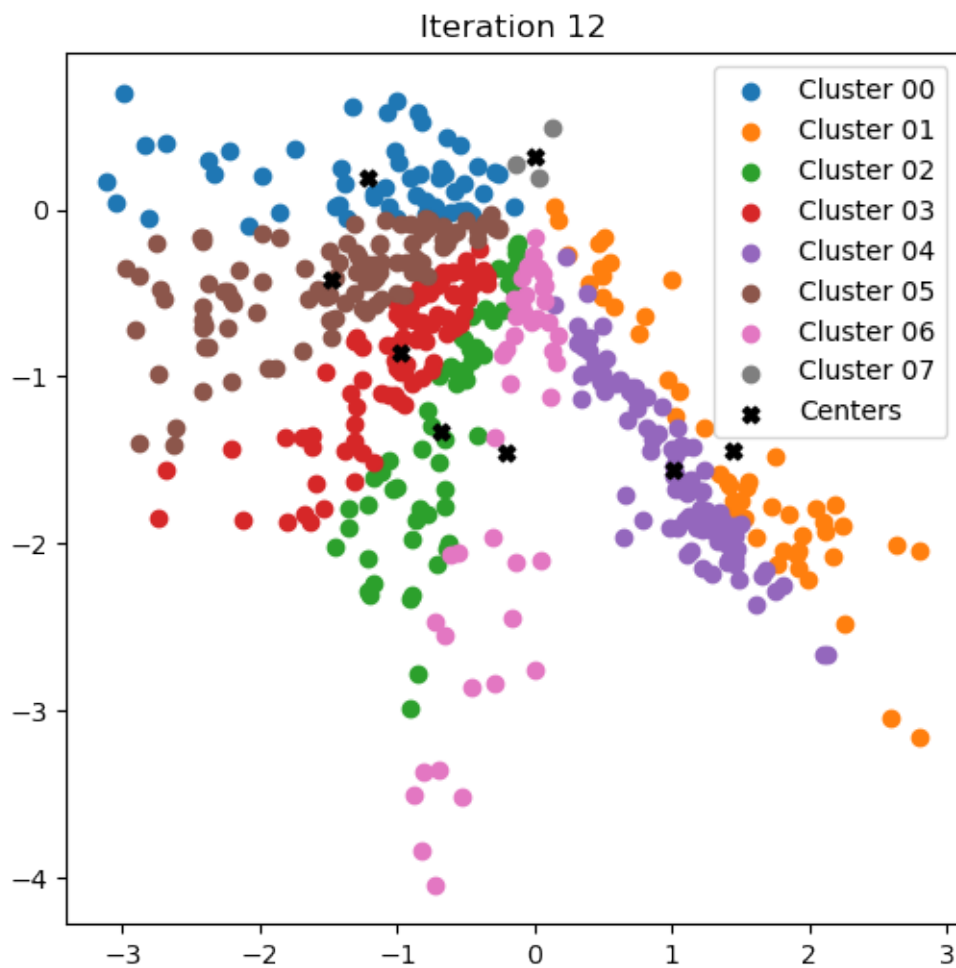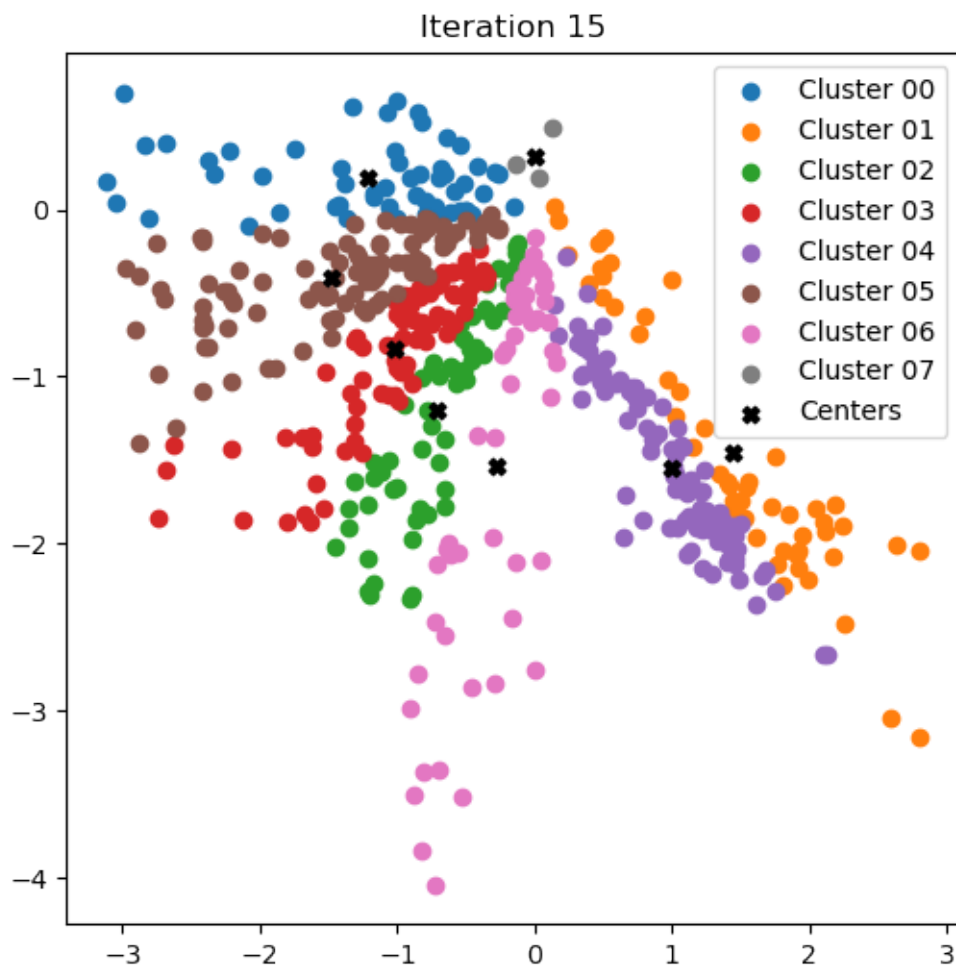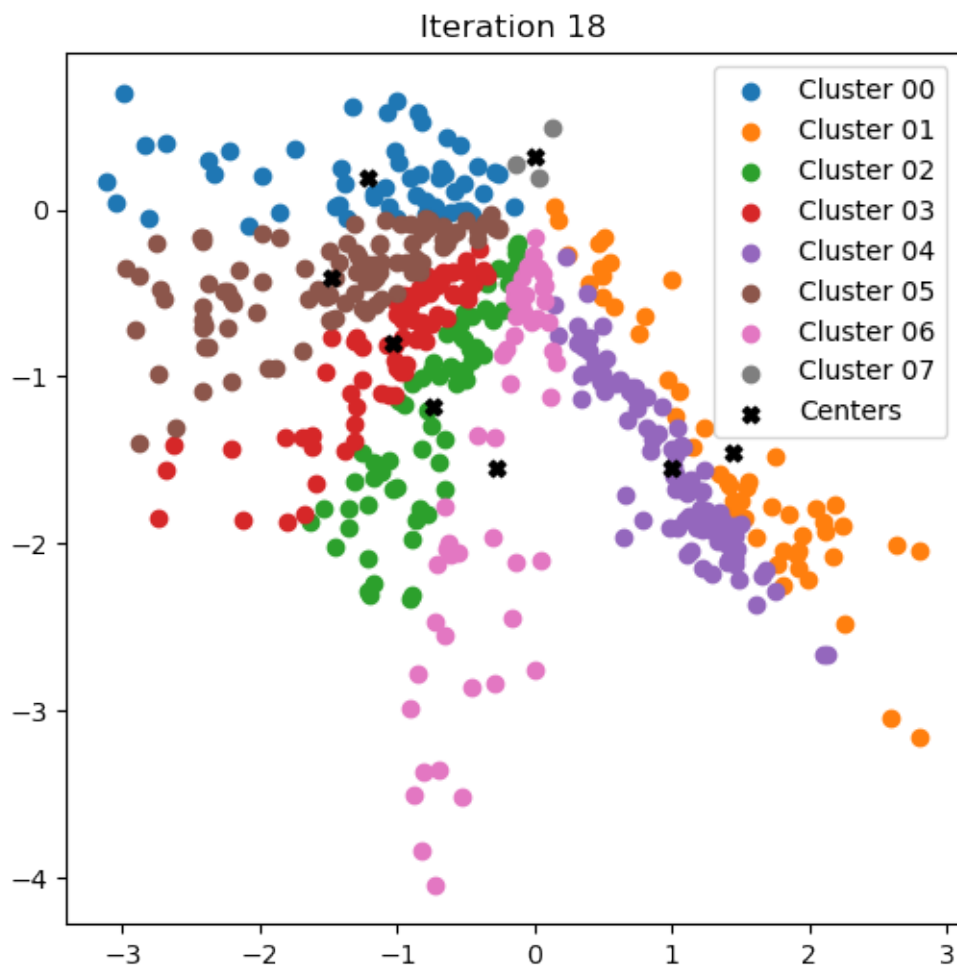
5

Took 31 iterations to converge

Iteration 00

Iteration 06

Iteration 09

Iteration 12

Iteration 15

Iteration 18

Iteration 21

Iteration 24

Iteration 27

Iteration 30
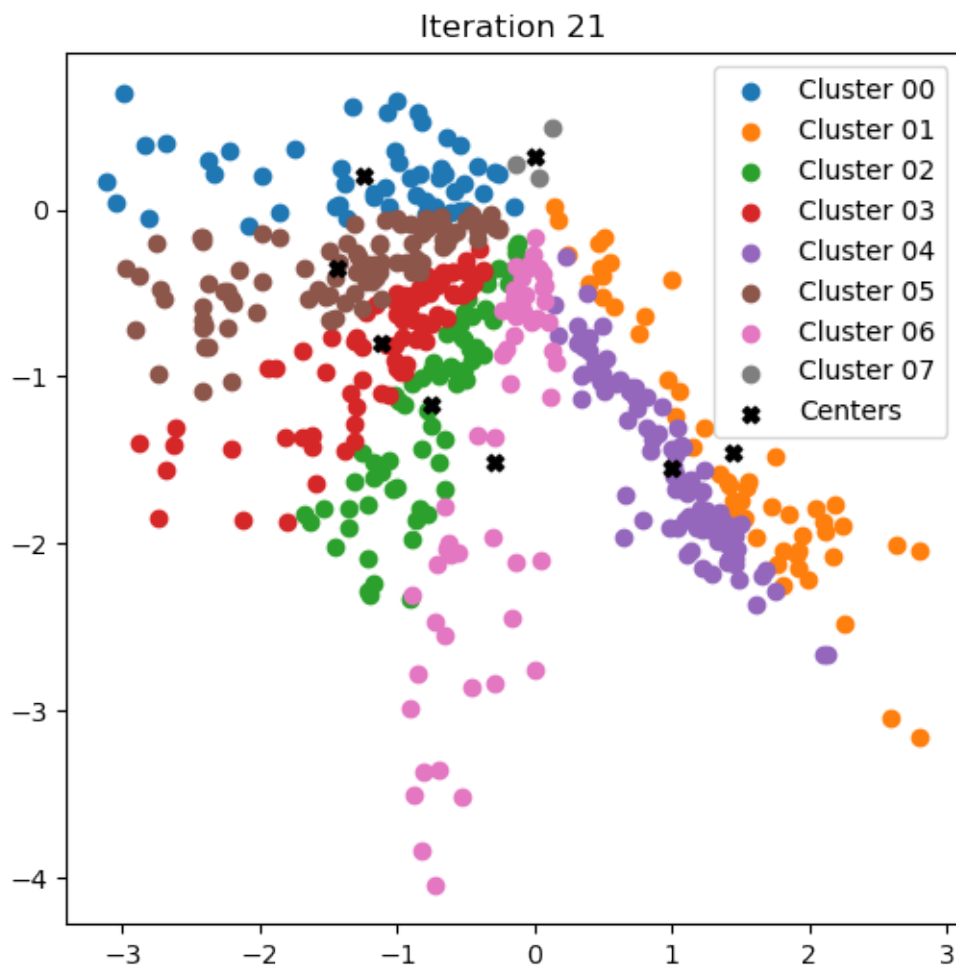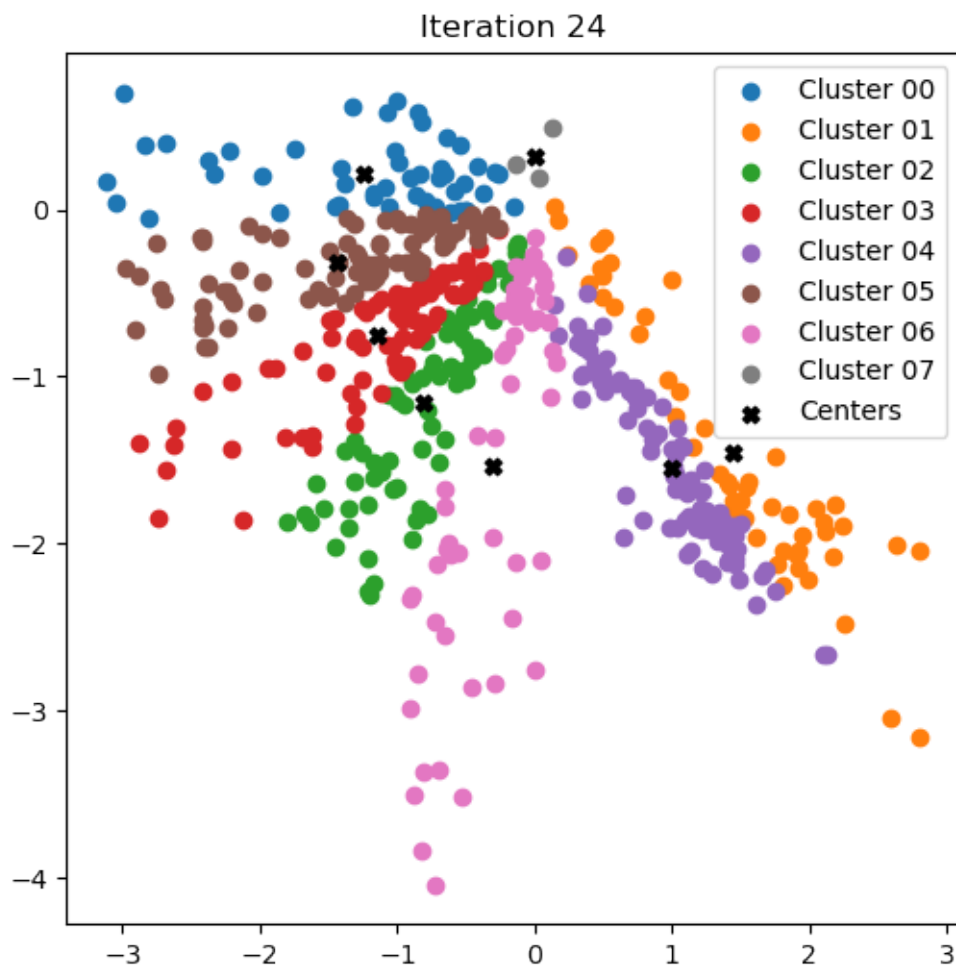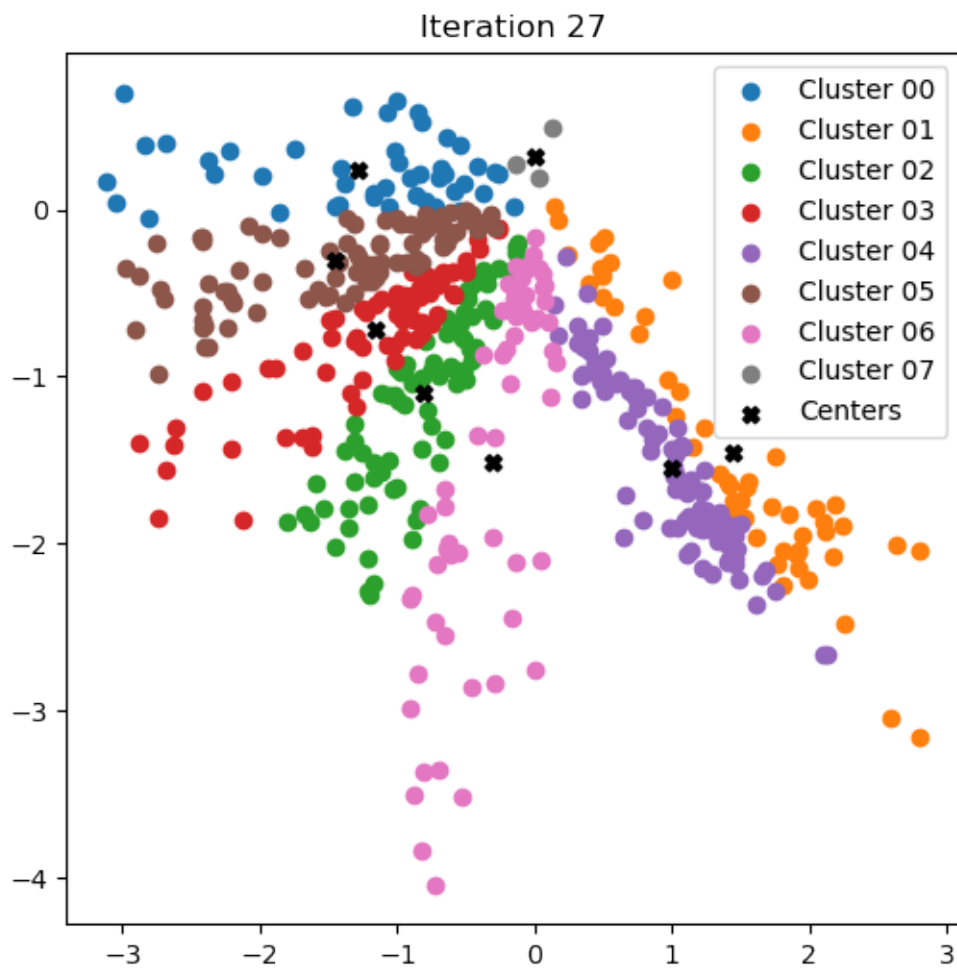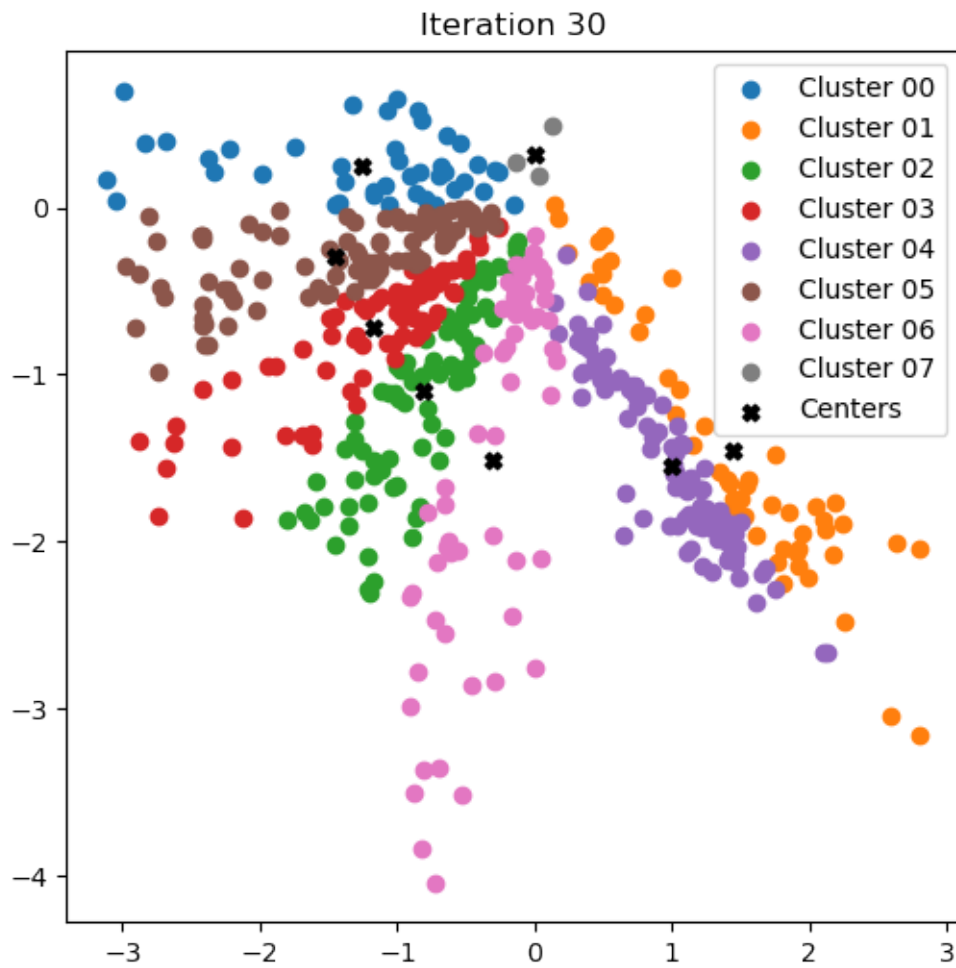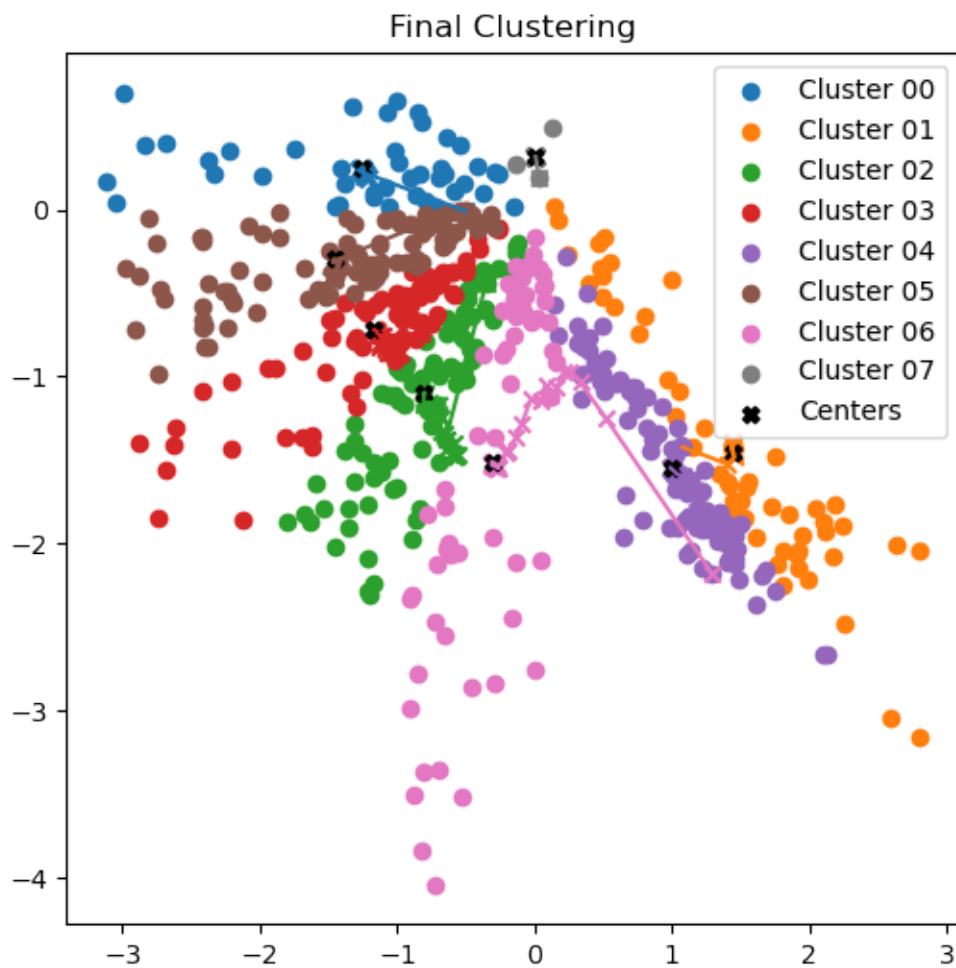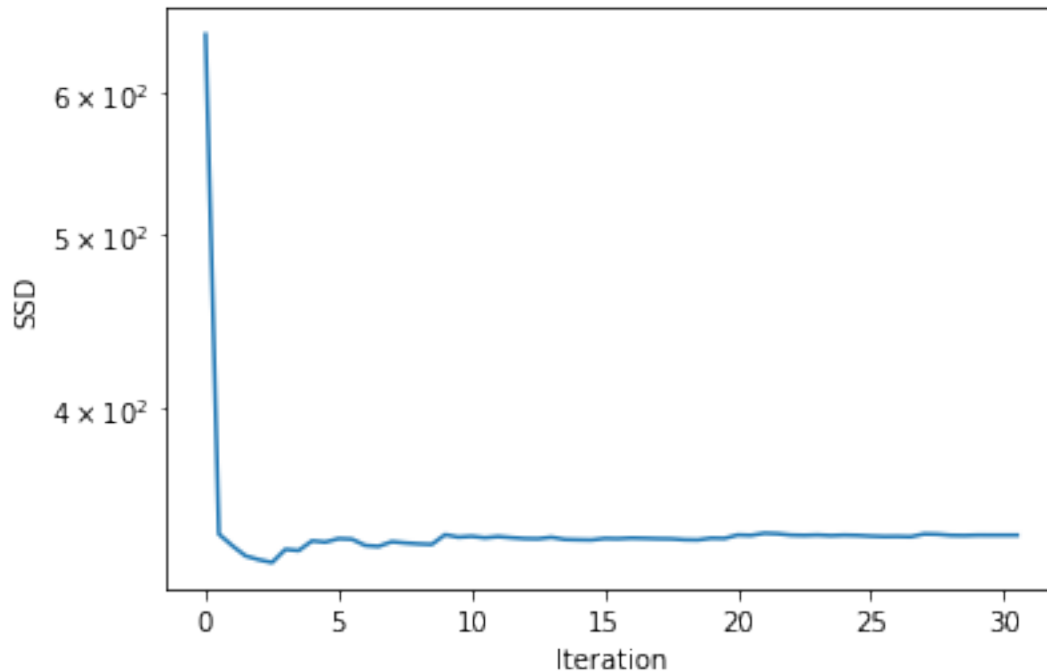
1.3 3.) Expectation Maximization for Gaussian Mixture Models (7 Points)

In the following we implement the Expectation Maximization (EM) Algorithm to fit a Gaussian Mixture Model (GMM) to data. We start with an implemenation for the log density of a single Gaussian (take some time to compare this implementation with the one used in the first exercies)...

```
[14]: %matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from typing import Tuple

def gaussian_log_density(samples: np.ndarray, mean: np.ndarray, covariance: np.
 ↪ndarray) -> np.ndarray:
    """
    Computes Log Density of samples under a Gaussian Distribution.
    We already saw an implementation of this in the first exercise and noted␣
 ↪there that this was not the "proper"
    way of doing it. Compare the two implementations.
    :param samples: samples to evaluate (shape: [N x dim)
    :param mean: Mean of the distribution (shape: [dim])
    :param covariance: Covariance of the distribution (shape: [dim x dim])
    :return: log N(x|mean, covariance) (shape: [N])
```

```python
    """
    dim = mean.shape[0]
    chol_covariance = np.linalg.cholesky(covariance)
    # Efficient and stable way to compute the log determinant and squared term␣
↪efficiently using the cholesky
    logdet = 2 * np.sum(np.log(np.diagonal(chol_covariance) + 1e-25))
    # (Actually, you would use scipy.linalg.solve_triangular but I wanted to␣
↪spare you the hustle of setting
    #  up scipy)
    chol_inv = np.linalg.inv(chol_covariance)
    exp_term = np.sum(np.square((samples - mean) @ chol_inv.T), axis=-1)
    return -0.5 * (dim * np.log(2 * np.pi) + logdet + exp_term)
```

… and some plotting functionaliy for 2D GMMs:

```python
[15]: def visualize_2d_gmm(samples, weights, means, covs, title):
          """Visualizes the model and the samples"""
          plt.figure(figsize=[7,7])
          plt.title(title)
          plt.scatter(samples[:, 0], samples[:, 1], label="Samples", c=next(plt.gca().
      ↪_get_lines.prop_cycler)['color'])

          for i in range(means.shape[0]):
              c = next(plt.gca()._get_lines.prop_cycler)['color']

              (largest_eigval, smallest_eigval), eigvec = np.linalg.eig(covs[i])
              phi = -np.arctan2(eigvec[0, 1], eigvec[0, 0])

              plt.scatter(means[i, 0:1], means[i, 1:2], marker="x", c=c)

              a = 2.0 * np.sqrt(largest_eigval)
              b = 2.0 * np.sqrt(smallest_eigval)

              ellipse_x_r = a * np.cos(np.linspace(0, 2 * np.pi, num=200))
              ellipse_y_r = b * np.sin(np.linspace(0, 2 * np.pi, num=200))

              R = np.array([[np.cos(phi), np.sin(phi)], [-np.sin(phi), np.cos(phi)]])
              r_ellipse = np.array([ellipse_x_r, ellipse_y_r]).T @ R
              plt.plot(means[i, 0] + r_ellipse[:, 0], means[i, 1] + r_ellipse[:, 1],␣
      ↪c=c,
                      label="Component {:02d}, Weight: {:0.4f}".format(i, weights[i]))
          plt.legend()
```

Now to the actual task: You need to implement 3 functions: - the log likelihhod of a GMM for evaluation - the E-Step of the EM algorithm for GMMs - the M-Step of the EM algorithm for GMMs (for this one now for loops are allowed. Using them here will lead to point deduction)

All needed equations are in the slides

The EM algorithm can be summarized as **### The Expectation-Maximization Algorithm 1. Initialize $\mu_\ell, \Sigma_\ell, \pi_\ell$ and compute initial log-likelihood as $\mathcal{L}(0)$. Set iteration counter $I = 1$ 1. Expectation step: Evaluate, using the current μ_ℓ, Σ_ℓ and π_ℓ**

$$\gamma(x_i^{[\textit{train}]}, y_{i,\ell}) = \frac{\pi_\ell \mathcal{N}(x_i^{[\textit{train}]}; \mu_\ell, \Sigma_\ell)}{\sum_{k=1}^m \pi_k \mathcal{N}(x_i^{[\textit{train}]}; \mu_k, \Sigma_k)}$$

3. Maximization step: Re-estimate the parameters as

$$\mu_\ell^{\textit{new}} = \frac{1}{N_\ell} \sum_{i=1}^N \gamma(x_i^{[\textit{train}]}, y_{i,\ell}) x_i^{[\textit{train}]}$$

$$\Sigma_\ell^{\textit{new}} = \frac{1}{N_\ell} \sum_{i=1}^N \gamma(x_i^{[\textit{train}]}, y_{i,\ell})(x_i^{[\textit{train}]} - \mu_\ell^{\textit{new}})(x_i^{[\textit{train}]} - \mu_\ell^{\textit{new}})^{\mathrm{T}}$$

$$\pi_\ell^{\textit{new}} = \frac{N_\ell}{N}$$

where

$$N_\ell = \sum_{i=1}^N \gamma(x_i^{[\textit{train}]}, y_{i,\ell})$$

4. Evaluate the log-likelihood

$$\mathcal{L}^{(I)} = \sum_{i=1}^N \log \left(\sum_{\ell=1}^m \pi_\ell \mathcal{N}(x_i^{[\textit{train}]}; \mu_\ell, \Sigma_\ell) \right)$$

If $|\mathcal{L}^{(I)} - \mathcal{L}^{(I-1)}| < \epsilon$ abort, otherwise go to step 2. (ϵ: small constant)

```python
[16]: def gmm_log_likelihood(samples: np.ndarray, weights: np.ndarray, means: np.
      ↪ndarray, covariances: np.ndarray) -> float:
          """ Computes the Log Likelihood of samples given parameters of a GMM.
          :param samples: samples "x" to compute ess for     (shape: [N, dim])
          :param weights: weights (i.e., p(z) ) of old model (shape: [num_components])
          :param means: means of old components p(x|z) (shape: [num_components, dim])
          :param covariances: covariances of old components p(x|z) (shape:␣
      ↪[num_components, dim, dim]
          :return: log likelihood
          """
          ###########################################################
          # TODO Implement the log-likelihood for Gaussian Mixtures
          k, N = weights.shape[0], samples.shape[0]
          L_o = 0
          L = np.zeros((N, k))
          for i in range(k):
              L[:, i] = np.log(weights[i] * np.exp(gaussian_log_density(samples,␣
      ↪means[i], covariances[i])))
          return np.sum(L)
          ###########################################################
```

```python
def e_step(samples: np.ndarray, weights: np.ndarray, means: np.ndarray,
→covariances: np.ndarray) -> np.ndarray:
    """ E-Step of EM for fitting GMMs. Computes estimated sufficient statistics
→(ess), p(z|x), using the old model from
    the previous iteration. In the GMM case they are often referred to as
→"responsibilities".
    :param samples: samples "x" to compute ess for    (shape: [N, dim])
    :param weights: weights (i.e., p(z) ) of old model (shape: [num_components])
    :param means: means of old components p(x|z) (shape: [num_components, dim])
    :param covariances: covariances of old components p(x|z) (shape:
→[num_components, dim, dim]
    :return: Responsibilities p(z|x) (Shape: [N x num_components])
    """
    ##############################################################
    # TODO Implement the E-Step for EM for Gaussian Mixtrue Models.
    norm_term = 0
    gema = np.zeros((samples.shape[0], weights.shape[0]))
    for i in range(weights.shape[0]):
        norm_term += weights[i] * gaussian_log_density(samples, means[i],
→covariances[i])
        gema[:, i] = weights[i] * gaussian_log_density(samples, means[i],
→covariances[i])
    for i in range(weights.shape[0]):
        gema[:, i]/=norm_term
    return gema
    ##############################################################


def m_step(samples: np.ndarray, responsibilities: np.ndarray) -> Tuple[np.
→ndarray, np.ndarray, np.ndarray]:
    """ M-Step of EM for fitting GMMs. Computes new parameters given samples and
→responsibilities p(z|x)
    :param samples: samples "x" to fit model to (shape: [N, dim])
    :param responsibilities: p(z|x) (Shape: [N x num_components]), as computed
→by E-step
    :return: - new weights p(z) (shape [num_components])
            - new means of components p(x|z) (shape: [num_components, dim])
            - new covariances of components p(x|z) (shape: [num_components,
→dim, dim]
    """
    ##############################################################
    # TODO: Implement the M-Step for EM for Gaussian Mixture models. You are not
→allowed to use any for loops!
    # Hint: Writing it directly without for loops is hard, especially if you are
→not experienced with broadcasting.
```

```
    # It's maybe easier to first implement it using for loops and then try␣
→getting rid of them, one after another.
        #######################################################
    # TODO: Implement the M-Step for EM for Gaussian Mixture models. You are not␣
→allowed to use any for loops!
    # Hint: Writing it directly without for loops is hard, especially if you are␣
→not experienced with broadcasting.
    # It's maybe easier to first implement it using for loops and then try␣
→getting rid of them, one after another.
    res = responsibilities
    N, k, dim = samples.shape[0], res.shape[1], samples.shape[1]
    means = np.zeros((k, dim))
    weights = np.zeros((k))
    covs = np.tile(np.eye(dim)[None, ...], [k, 1, 1])
    norm = np.sum(res, axis=0)
    mu = np.zeros((N, dim))
    for i in range(res.shape[1]):
        means[i, :] = res[:, i].T @ samples / norm[i]
        for j in range(dim):
            mu[:, j] = (res[:, i] * (samples - means[i, :])[:, j])
        covs[i, ...] = mu.T @ (samples-means[i, :]) / norm[i]
        weights[i] = norm[i] / N
    return weights, means, covs
        #######################################################
```

We wrap out functions with the actual algorithm, iterating E and M step

```python
[17]: def fit_gaussian_mixture(samples: np.ndarray, num_components: int, num_iters:␣
      →int = 30, vis_interval: int = 5):
          """Fits a Gaussian Mixture Model using the Expectation Maximization Algorithm
          :param samples: Samples to fit the model to (shape: [N, dim]
          :param num_components: number of components of the GMM
          :param num_iters: number of iterations
          :param vis_interval: After how many iterations to generate the next plot
          :return: - final weights p(z) (shape [num_components])
                   - final means of components p(x|z) (shape: [num_components, dim])
                   - final covariances of components p(x|z) (shape: [num_components,␣
      →dim, dim]
                   - log_likelihoods: log-likelihood of data under model after each␣
      →iteration (shape: [num_iters])
          """
          # Initialize Model: We initialize with means randomly picked from the data,␣
      →unit covariances and uniform
          # component weights. This works here but in general smarter initialization␣
      →techniques might be necessary, e.g.,
          # k-means
```

```python
        initial_idx = np.random.choice(len(samples), num_components, replace=False)
        means = samples[initial_idx]
        covs = np.tile(np.eye(data.shape[-1])[None, ...], [num_components, 1, 1])
        weights = np.ones(num_components) / num_components


        # bookkeeping:
        log_likelihoods = np.zeros(num_iters)

        # iterate E and M Steps
        for i in range(num_iters):
            responsibilities = e_step(samples, weights, means, covs)
            weights, means, covs = m_step(samples, responsibilities)

            # Plotting
            if i % vis_interval == 0:
                visualize_2d_gmm(data, weights, means, covs, title="After Iteration␣
 ↪{:02d}".format(i))

            log_likelihoods[i] = gmm_log_likelihood(samples, weights, means, covs)
        return weights, means, covs, log_likelihoods
```

Finally we load some data and run the algorithm. Feel free to play around with the parameters a bit.

```python
[18]: ## ADAPTABLE PARAMETERS:

np.random.seed(0)
num_components = 5
num_iters = 30
vis_interval = 5

# CHOOSE A DATASET
#data = np.load("samples_1.npy")
data = np.load("samples_2.npy")
#data = np.load("samples_3.npy")
#data = np.load("samples_u.npy")

# running and ploting
final_weights, final_means, final_covariances, log_likeihoods = \
    fit_gaussian_mixture(data, num_components, num_iters, vis_interval)
visualize_2d_gmm(data, final_weights, final_means, final_covariances,␣
 ↪title="Final Model")

plt.figure()
plt.title("Log-Likelihoods over time")
plt.plot(log_likeihoods)
plt.xlabel("iteration")
```

```
plt.ylabel("log-likelihood")
plt.show()
```


After Iteration 00

Component 00, Weight: 0.3385
Component 01, Weight: 0.1805
Component 02, Weight: 0.1645
Component 03, Weight: 0.1798
Component 04, Weight: 0.1367
Samples

After Iteration 05

- Component 00, Weight: 0.3388
- Component 01, Weight: 0.1812
- Component 02, Weight: 0.1590
- Component 03, Weight: 0.1846
- Component 04, Weight: 0.1363
- Samples

After Iteration 10

Component 00, Weight: 0.3389
Component 01, Weight: 0.1812
Component 02, Weight: 0.1590
Component 03, Weight: 0.1846
Component 04, Weight: 0.1363
Samples

After Iteration 15

Component 00, Weight: 0.3389
Component 01, Weight: 0.1812
Component 02, Weight: 0.1590
Component 03, Weight: 0.1846
Component 04, Weight: 0.1363
Samples

After Iteration 20

Component 00, Weight: 0.3389
Component 01, Weight: 0.1812
Component 02, Weight: 0.1590
Component 03, Weight: 0.1846
Component 04, Weight: 0.1363
● Samples

After Iteration 25

Component 00, Weight: 0.3389
Component 01, Weight: 0.1812
Component 02, Weight: 0.1590
Component 03, Weight: 0.1846
Component 04, Weight: 0.1363
Samples

Final Model

Component 00, Weight: 0.3389
Component 01, Weight: 0.1812
Component 02, Weight: 0.1590
Component 03, Weight: 0.1846
Component 04, Weight: 0.1363
Samples

Log-Likelihoods over time