EXERCISE 3 - ML - Grundverfahren

1.) Constrained Optimization (6 Points)

You are given the following Optimization problem:

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

where $\boldsymbol{M}$ is a positive definit, symmetric Matrix. Note that vectors and matrices are boldsymbol, where Matrices have capital letters.
Derive the optimal solution for $\boldsymbol{x}$ 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 $\boldsymbol{M}$.
Don't forget to look up matrix-vector calculus in the matrix cookbook, if you don't remember the rules.

The Lagrangian is

$$L(x, \lambda) = \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{h} - \lambda(\boldsymbol{x}^T \boldsymbol{b} - c)$$

Thus, the dual optimization problem is

$$\max_{\lambda} \ \min_{\boldsymbol{x}} L(\boldsymbol{x}, \lambda)$$

We thus find $\boldsymbol{x}^*$

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

Observe that, since $\boldsymbol{M}$ is symmetric

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

And

$$\begin{aligned}\boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{h} - \lambda(\boldsymbol{x}^T \boldsymbol{b} - c) &= \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} + \boldsymbol{x}^T(\boldsymbol{h} - \lambda \boldsymbol{b}) - \lambda c \\ &= \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} + \boldsymbol{x}^T \boldsymbol{h} - \lambda(\boldsymbol{x}^T \boldsymbol{b} - c) \\ &= \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} + \boldsymbol{x}^T(-1)(\lambda \boldsymbol{b} - \boldsymbol{h}) - \lambda c\end{aligned}$$

Thus,

$$
\begin{aligned}
L(\boldsymbol{x}^*, \lambda) &= \frac{1}{2}(\lambda \boldsymbol{b} - \boldsymbol{h})^T \boldsymbol{M}^{-1} \boldsymbol{M} \frac{1}{2} M^{-1}(\lambda \boldsymbol{b} - \boldsymbol{h}) - \frac{1}{2}(\lambda \boldsymbol{b} - \boldsymbol{h})^T \boldsymbol{M}^{-1}(\lambda \boldsymbol{b} - \boldsymbol{h}) - \lambda c \\
&= \frac{1}{4}(\lambda \boldsymbol{b} - \boldsymbol{h})^T M^{-1}(\lambda \boldsymbol{b} - \boldsymbol{h}) - \frac{1}{2}(\lambda \boldsymbol{b} - \boldsymbol{h})^T \boldsymbol{M}^{-1}(\lambda \boldsymbol{b} - \boldsymbol{h}) - \lambda c \\
&= -\frac{1}{4}(\lambda \boldsymbol{b} - \boldsymbol{h})^T M^{-1}(\lambda \boldsymbol{b} - \boldsymbol{h}) - \lambda c
\end{aligned}
$$

We know find λ^* using the chain rule

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

It follows that

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

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.

In [98]:

```python
%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_history: Optional[np.ndarray] = None, title: Opt
    """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]
                    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="
            plt.plot(centers_history[:, i, 0], centers_history[:, i, 1], c=c)

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

    if title is not None:
        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

In [99]:

```
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
    ############################################################
    deltas = [p - centers for p in data_points]
    dist = lambda vs : [np.sqrt(np.sum(v ** 2)) for v in vs]
    distances = [dist(vs) for vs in deltas]
    onehot = lambda v : [1 if n == np.amin(v) else 0 for n in v]
    onehots = np.array([onehot(v) for v in distances])
    return onehots

def adjustment_step(data_points: np.ndarray, assignments_one_hot: np.ndarray) -> np.
    """
    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) (sh
    :return Adjusted Centers (shape: [k, data_dim])
    """

    ############################################################
    # TODO Implement the adjustment step of the k-Means algorithm
    ############################################################
    avgs = np.zeros((assignments_one_hot.shape[1], data_points.shape[1]))
    counts = np.sum(assignments_one_hot, axis=0)
    for idx in range(0, data_points.shape[0]):
        avgs[assignments_one_hot[idx].argmax()] += data_points[idx];
    avgs = avgs / counts[:,None]
    return avgs
```

In [87]:

```
data_points = np.array([[1,2],[3,4],[5,6],[7,8]])
centers = np.array([[1,2],[3,4],[5,6]])
print(assignment_step(data_points, centers))
```

```
[[1 0 0]
 [0 1 0]
 [0 0 1]
 [0 0 1]]
```

In [131…

```
data_points = np.array([[1,2],[3,4],[5,6],[7,8]])
centers = np.array([[1,2],[3,4],[5,6]])
```

```
onehots = assignment_step(data_points, centers)
print(adjustment_step(data_points, onehots))
```

```
[[1. 2.]
 [3. 4.]
 [6. 7.]]
```

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

In [100…

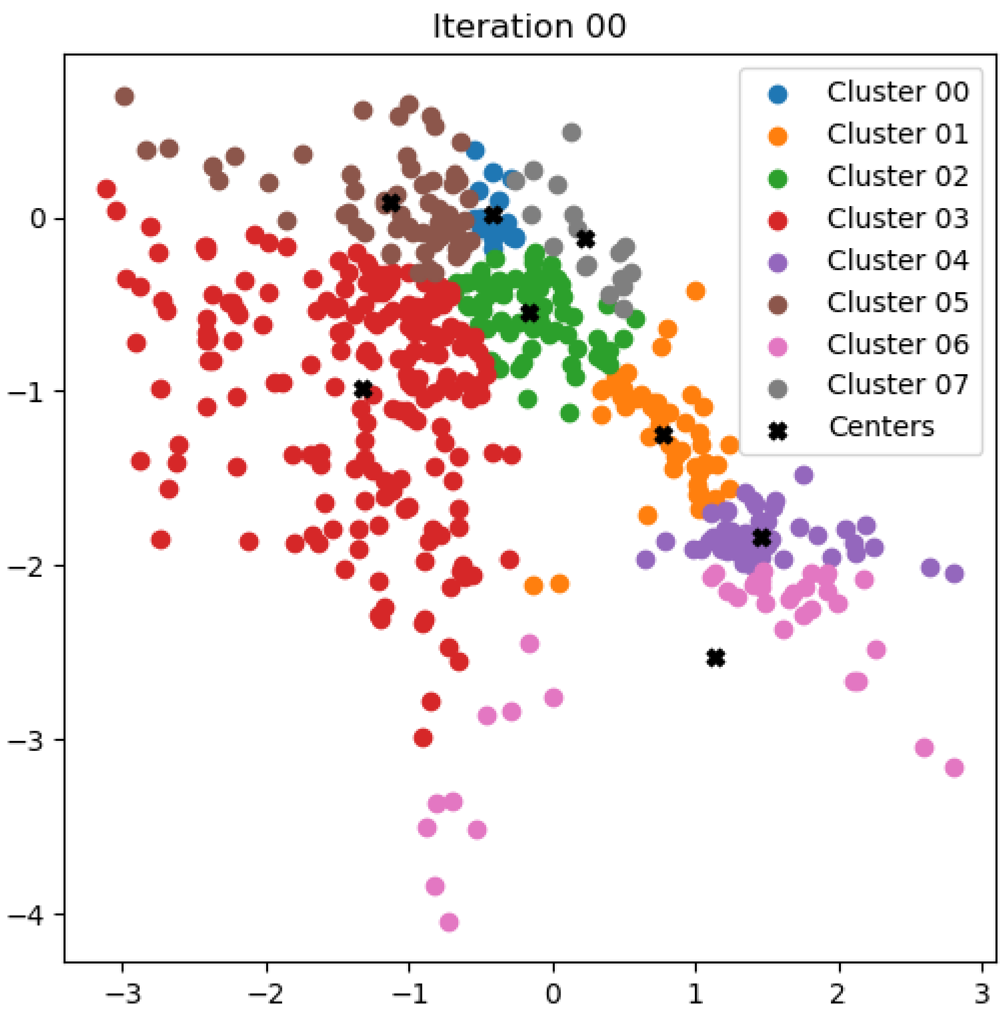
```
def k_means(data_points: np.ndarray, k: int, max_iter: int = 100, vis_interval: int
        Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """
    Simple K Means Implementation
    :param data_points: Data points to cluster  (shape: [N x data_dim])
    :param k: number of clusters
    :param max_iter: Maximum number of iterations to run if convergence is not reach
    :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, :, :]), axi
        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, :, :]), axi
        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, ti

        # 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_histo
```

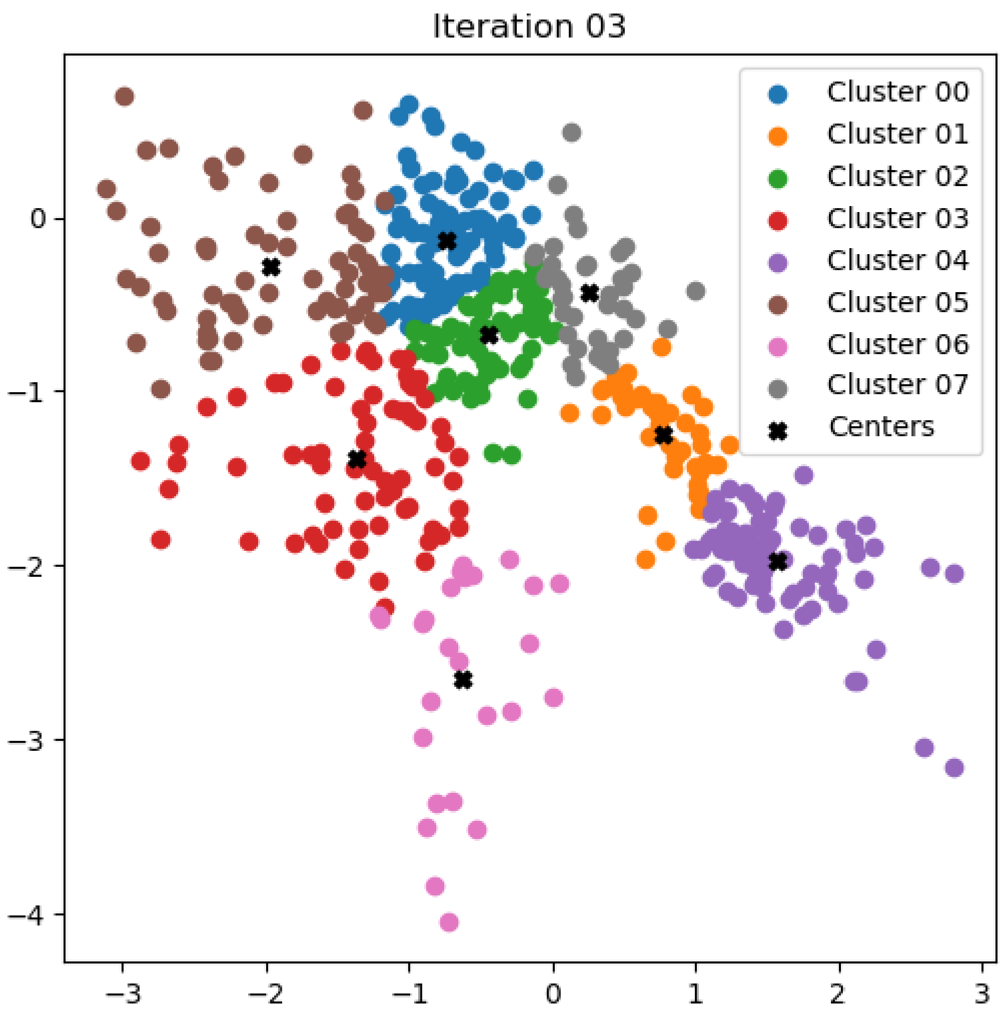
Finally we run the dataset and visualize the results. Here we provide 4 random datasets, each containing 500 2 samples and you can play around with the number of clustes, k, as well as the seed of the random number generator. Based on this seed the initial centers, and thus the final outcome, will vary.

In [101…

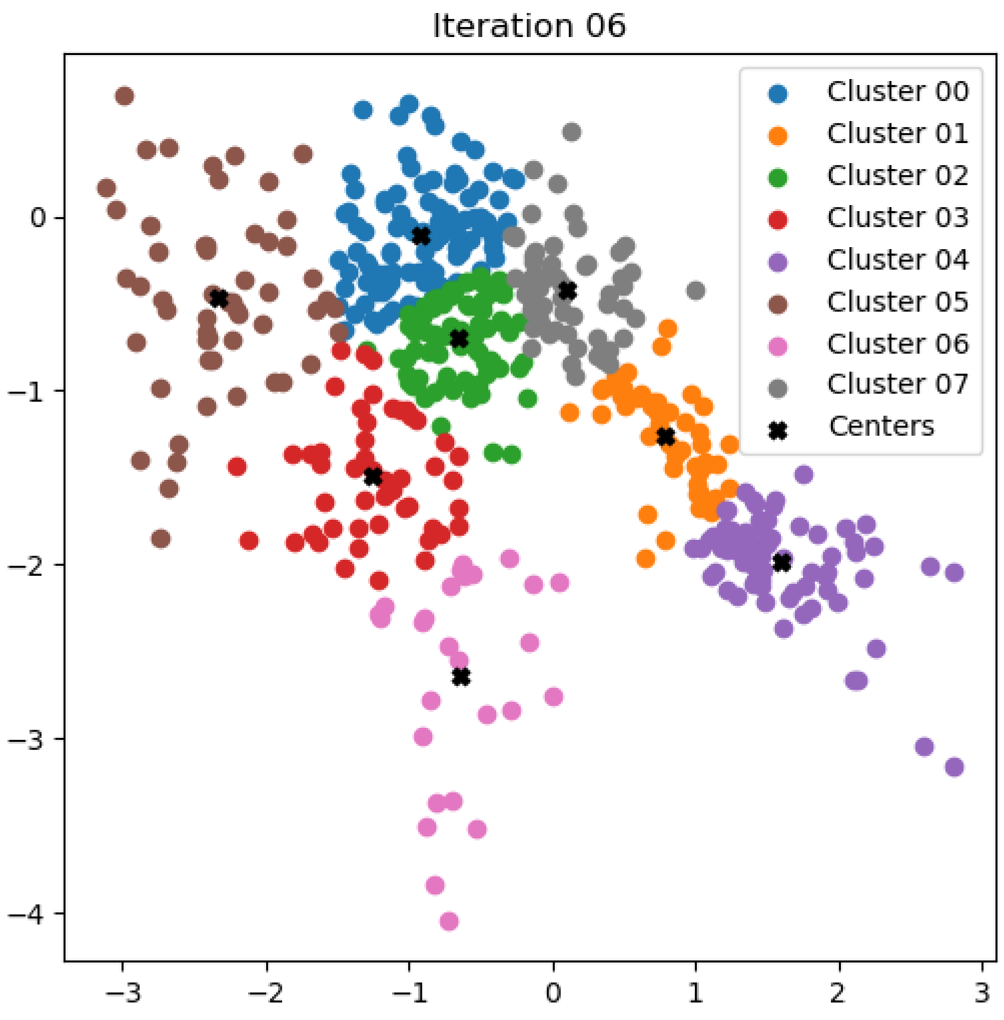
```
np.random.seed(42)

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_

# 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()
```

```
Took 17 iterations to converge
```

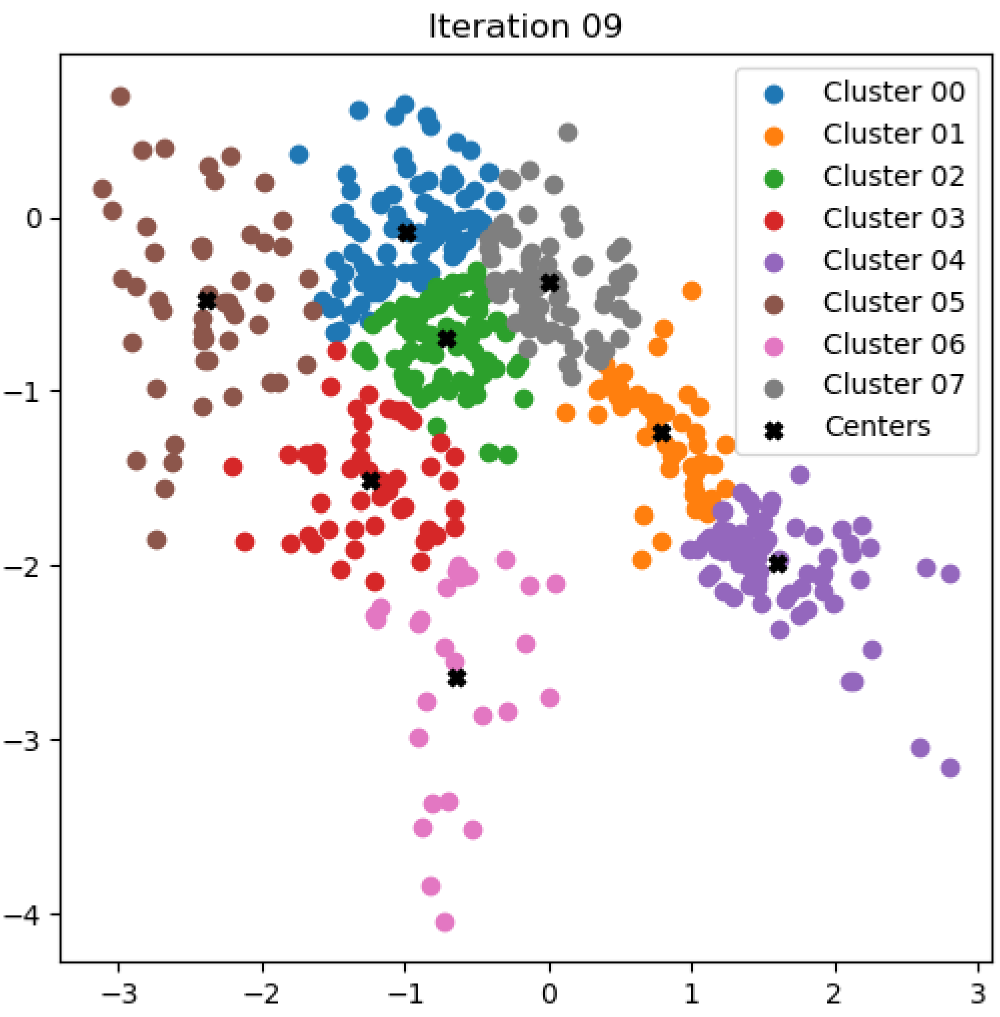


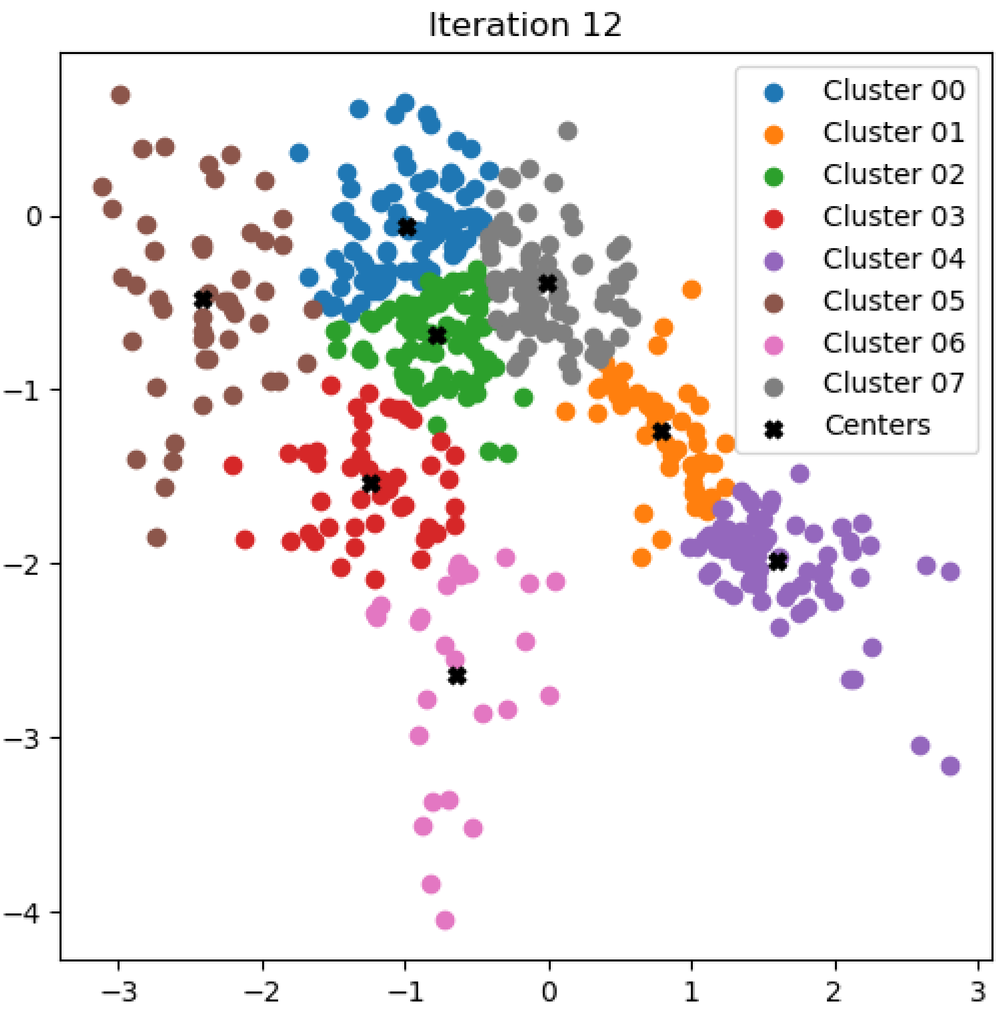


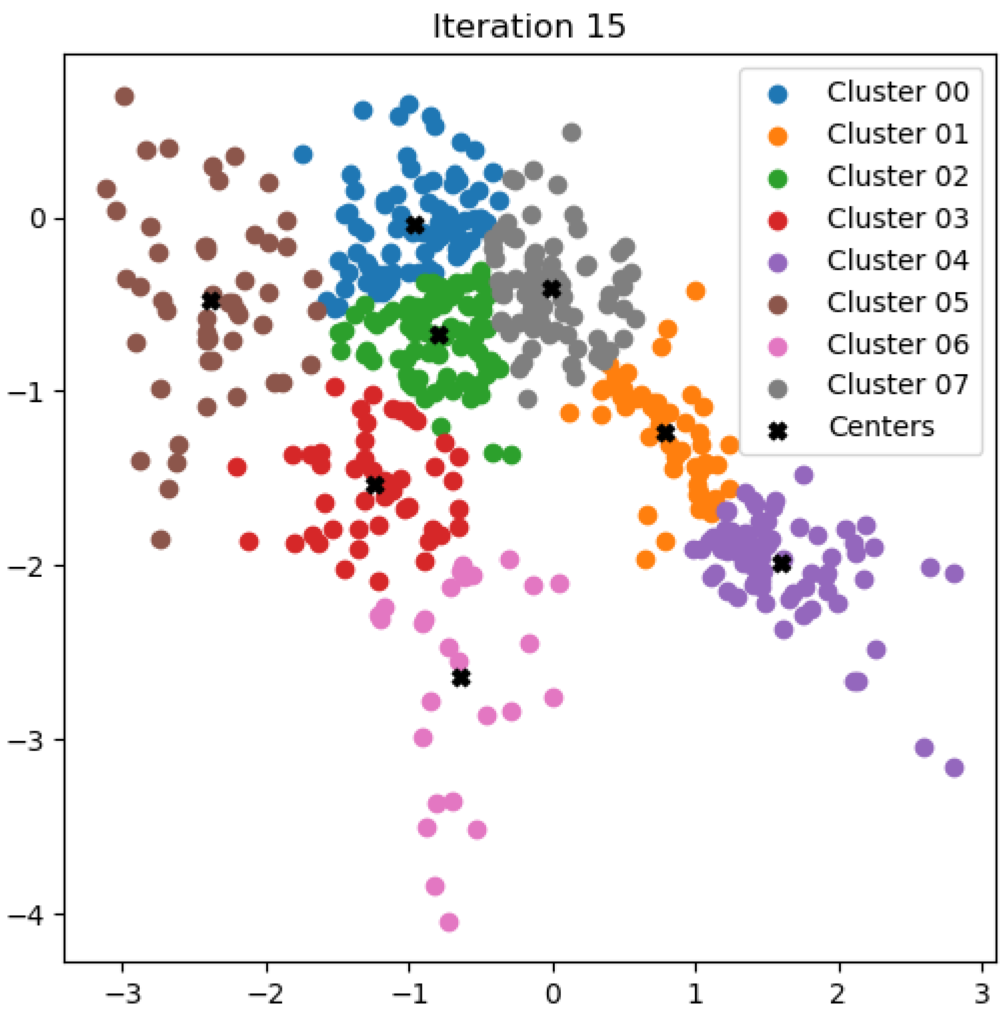
Iteration 03
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3

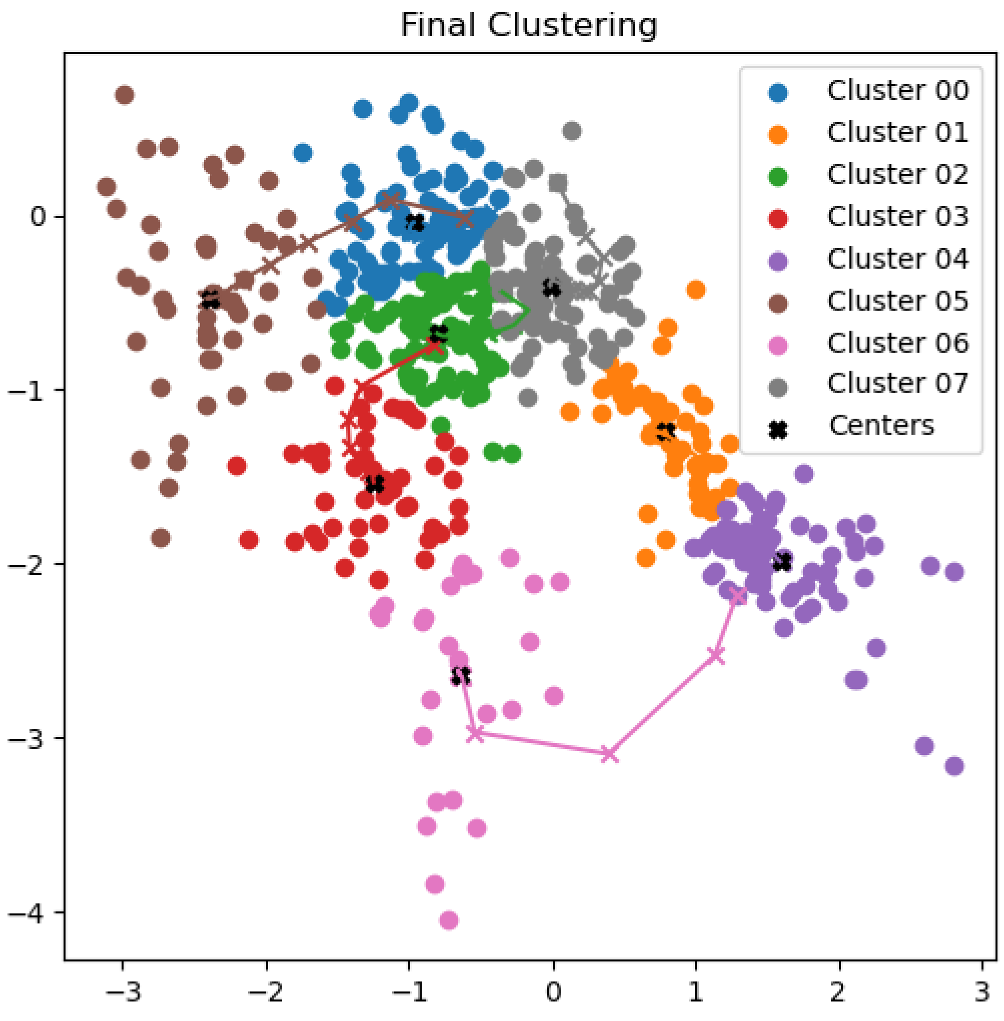


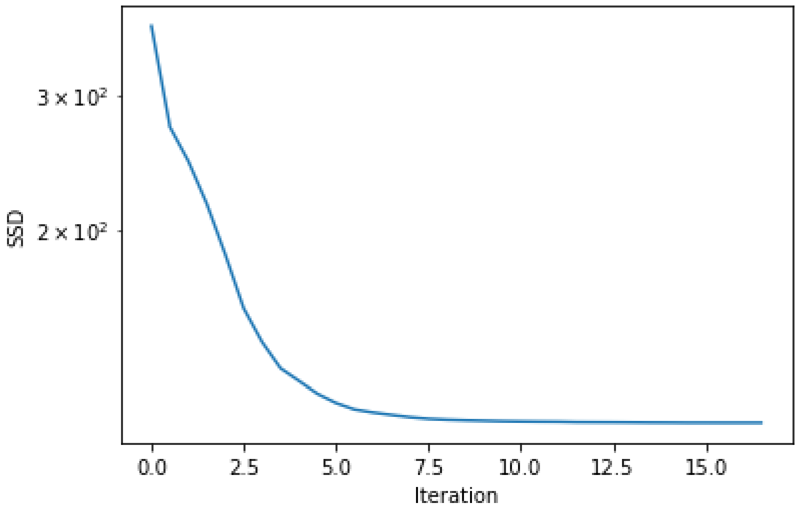
Iteration 06
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3


Iteration 09
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3


Iteration 12
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3


Iteration 15
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3


Final Clustering
Cluster 00
Cluster 01
Cluster 02
Cluster 03
Cluster 04
Cluster 05
Cluster 06
Cluster 07
Centers
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
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)...

In [32]:

```
%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.ndarr
    """
    Computes Log Density of samples under a Gaussian Distribution.
    We already saw an implementation of this in the first exercise and noted there t
    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])
    """
    dim = mean.shape[0]
    chol_covariance = np.linalg.cholesky(covariance)
    # Efficient and stable way to compute the log determinant and squared term effic
    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 y
    #  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:

In [33]:

```
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

    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 likelihod 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

In [97]:

```
def gmm_log_likelihood(samples: np.ndarray, weights: np.ndarray, means: np.ndarray,
    """ 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
    :return: log likelihood
    """
    ############################################################
    # TODO Implement the log-likelihood for Gaussian Mixtures
    ############################################################
    N = samples.shape[0]
    num_components = weights.shape[0]
    probs = np.zeros((N,num_components))
    for idx in range(0, num_components):
        probs[:,idx] = weights[idx] * np.exp(gaussian_log_density(samples, means[idx

    return np.log(np.sum(probs))

def e_step(samples: np.ndarray, weights: np.ndarray, means: np.ndarray, covariances:
    """ E-Step of EM for fitting GMMs. Computes estimated sufficient statistics (ess
    the previous iteration. In the GMM case they are often referred to as "responsib
    :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
    :return: Responsibilities p(z|x) (Shape: [N x num_components])
    """
    ############################################################
    # TODO Implement the E-Step for EM for Gaussian Mixtrue Models.
    ############################################################
    # for each sample, how much does each component contribute

    num_components = weights.shape[0]
    R = np.array([np.exp(gaussian_log_density(samples, means[idx,:], covariances[idx
    R = np.multiply(R, weights[:,None]).T
    divisor = np.sum(R,axis=1)

    return np.divide(R,divisor[:,None])




def m_step(samples: np.ndarray, responsibilities: np.ndarray) -> Tuple[np.ndarray, n
    """ M-Step of EM for fitting GMMs. Computes new parameters given samples and res
    :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-
    :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, di
    """
    ##########################################################
    # TODO: Implement the M-Step for EM for Gaussian Mixture models. You are not all
    # Hint: Writing it directly without for loops is hard, especially if you are not
    # It's maybe easier to first implement it using for loops and then try getting r
    ##########################################################

    N = samples.shape[0]
    norm = N
    num_components = responsibilities.shape[1]
    dim = samples.shape[1]

    sum_qi= np.sum(responsibilities,axis=0)
    weights = sum_qi/norm
    means = np.divide((responsibilities.T@samples), sum_qi[:,None])

    vecs = samples[:,None,:] - means

    covs = np.einsum('...j,...k->...jk',vecs,vecs)
    covs = (covs.reshape(N,num_components,dim ** 2) * responsibilities[:,:,None]).re
    cov_sums = np.sum(covs, axis=0)
    cov_sums = cov_sums.reshape(num_components, dim ** 2) / sum_qi[:,None]
    cov_sums = cov_sums.reshape(num_components, dim, dim)
    return weights,means,cov_sums
```

In [93]:

```
samples = np.array([[0,0],[1,1],[2,2],[3,3],[4,4], [5,5]])
weights = np.array([.5,.5,.5])
means = np.array([[0,0],[3,3],[4,4]])
covs = np.array([np.cov(samples.T), np.cov(samples.T), np.cov(samples.T)])
rs = e_step(samples, weights, means, covs)


m_step(samples,rs)
```

Out[93]:

```
(array([0.28598666, 0.37654129, 0.33747204]),
 array([[1.02754762, 1.02754762],
        [2.78613413, 2.78613413],
        [3.42855225, 3.42855225]]),
 array([[[1.33820725, 1.33820725],
         [1.33820725, 1.33820725]],

        [[2.41080066, 2.41080066],
         [2.41080066, 2.41080066]],

        [[2.02783904, 2.02783904],
         [2.02783904, 2.02783904]]]))
```

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

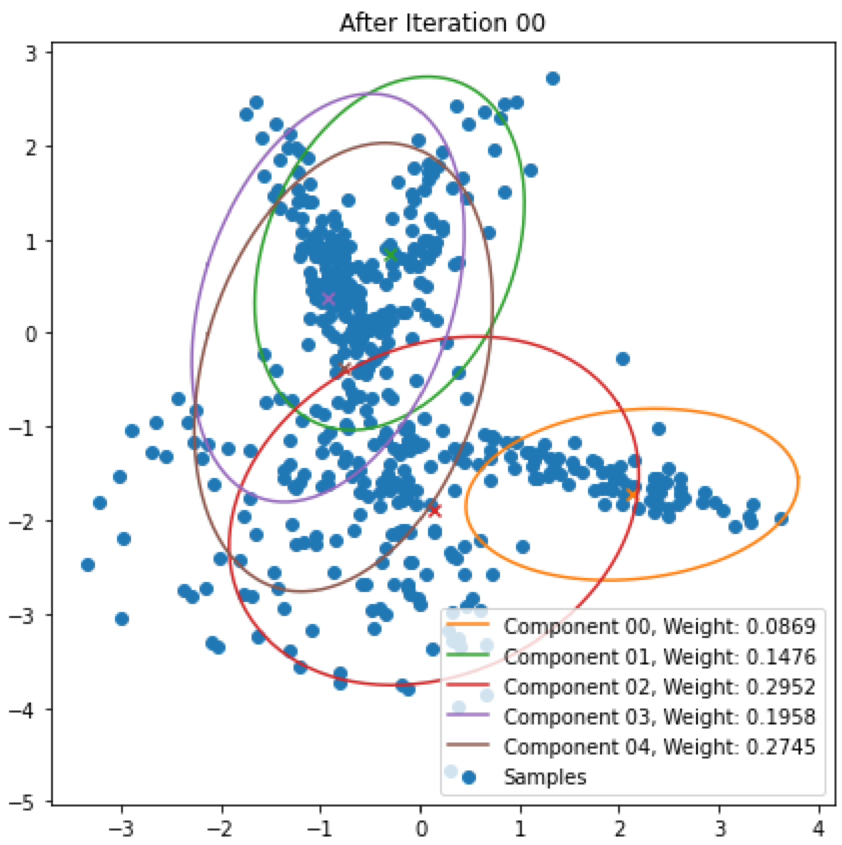
In [95]:

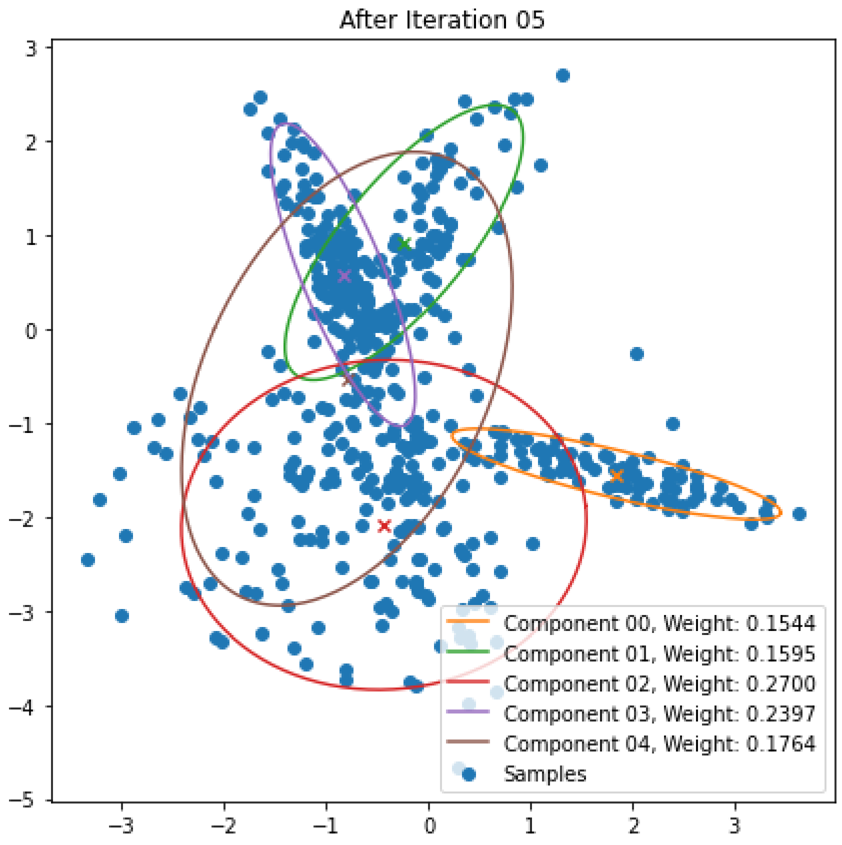
```
def fit_gaussian_mixture(samples: np.ndarray, num_components: int, num_iters: int =
    """Fits a Gaussian Mixture Model using the Expectation Maximization Algorithm
    :param samples: Samples to fit the model to (shape: [N, dim]
    :param num_components: number of components of the GMM
    :param 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,
             - log_likelihoods: log-likelihood of data under model after each iterat
    """
    # Initialize Model: We initialize with means randomly picked from the data, unit
    # component weights. This works here but in general smarter initialization techn
    # k-means
    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 {:02

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

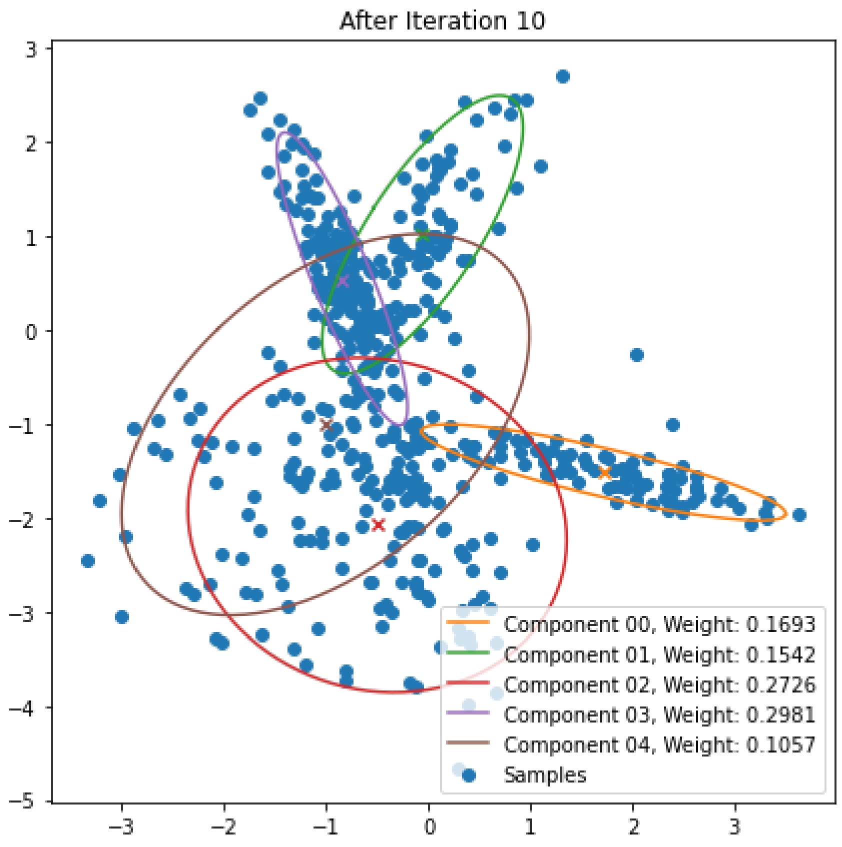
Finally we load some data and run the algorithm. Feel free to play around with the parameters a bit.

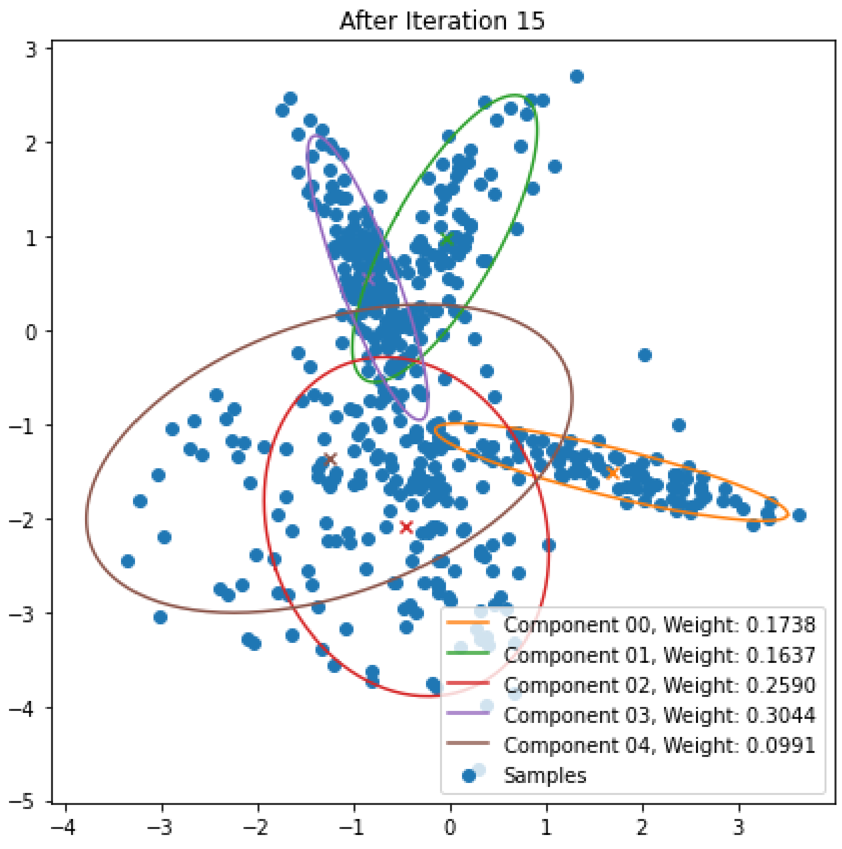
In [96]:

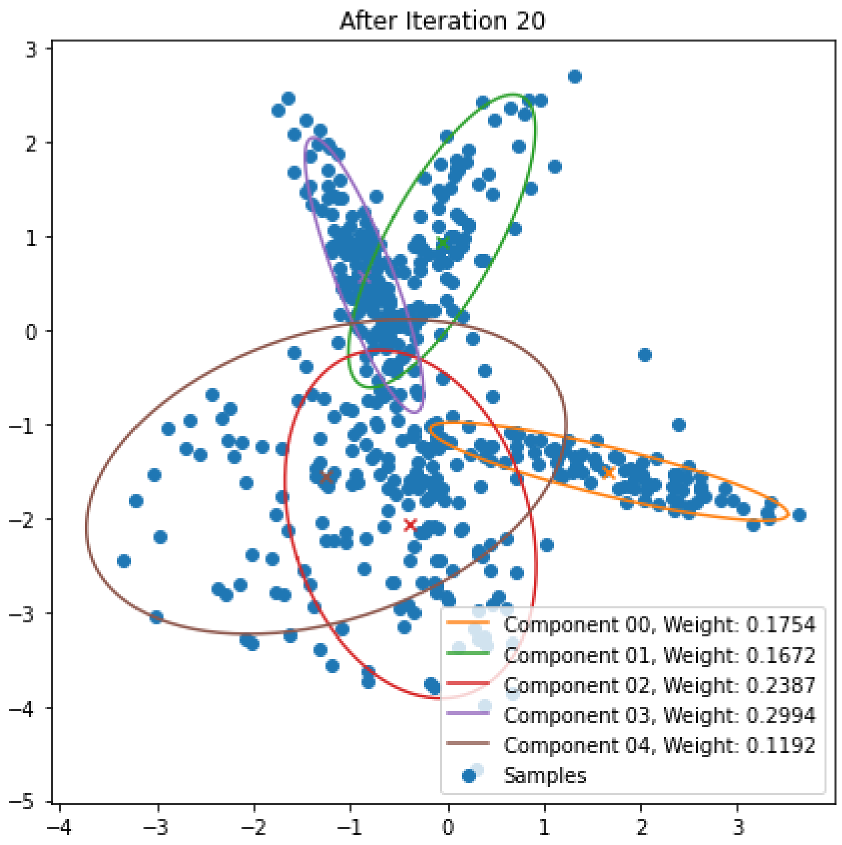
```
## ADAPTABLE PARAMETERS:

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 M

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

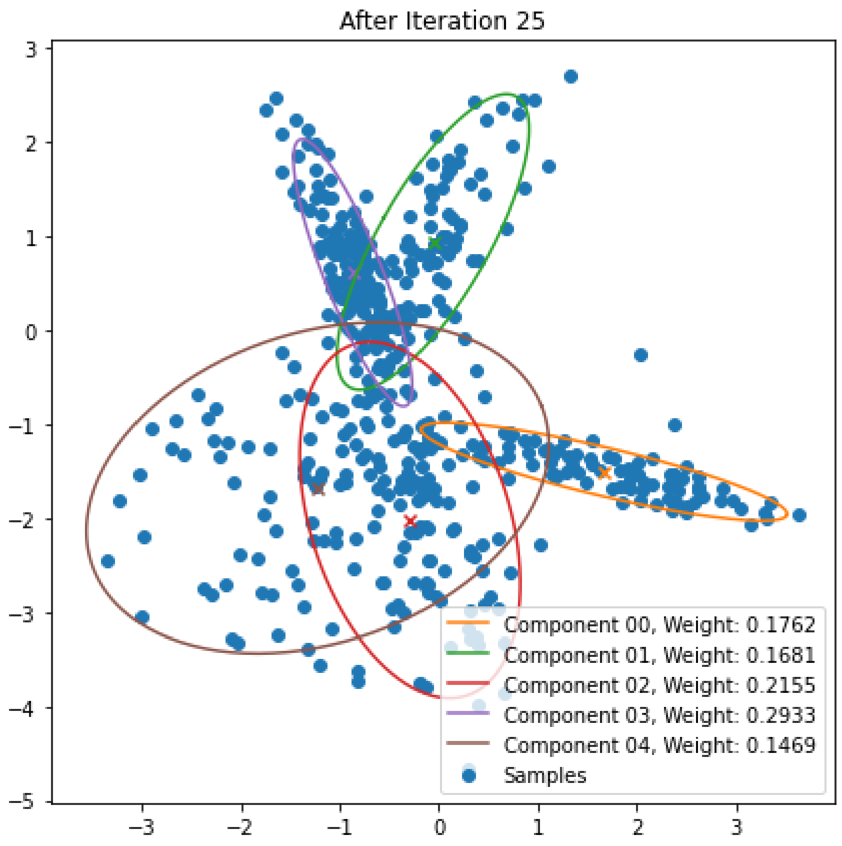


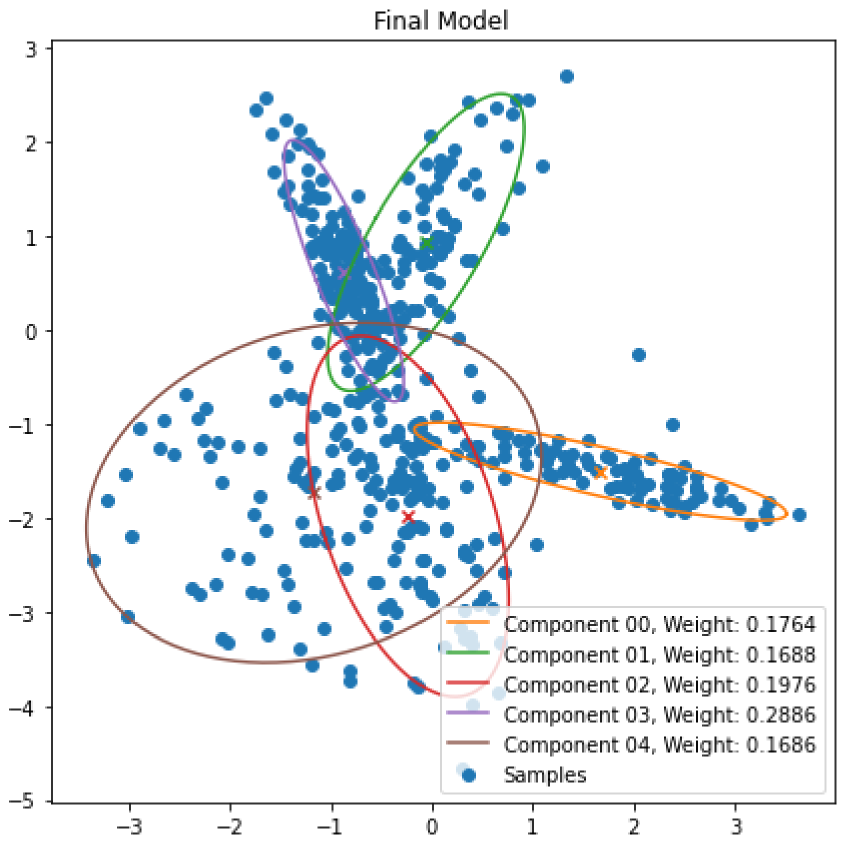


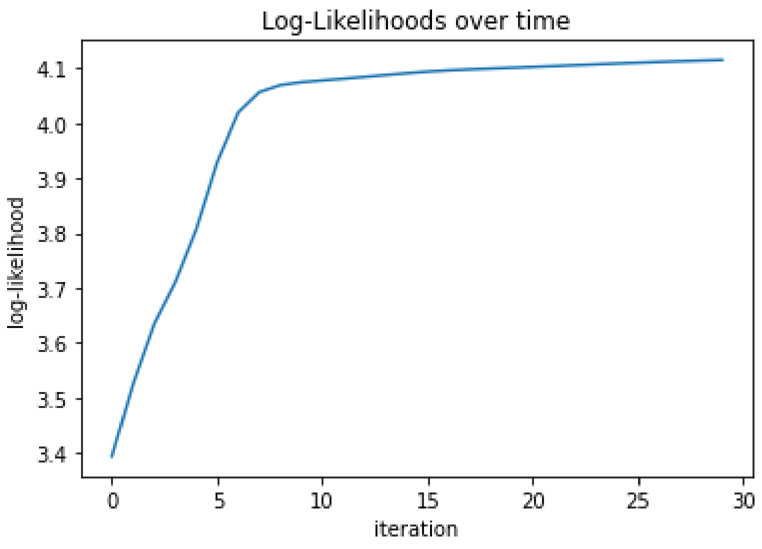

After Iteration 10
Component 00, Weight: 0.1693
Component 01, Weight: 0.1542
Component 02, Weight: 0.2726
Component 03, Weight: 0.2981
Component 04, Weight: 0.1057
Samples


After Iteration 15
Component 00, Weight: 0.1738
Component 01, Weight: 0.1637
Component 02, Weight: 0.2590
Component 03, Weight: 0.3044
Component 04, Weight: 0.0991
Samples


After Iteration 20
Component 00, Weight: 0.1754
Component 01, Weight: 0.1672
Component 02, Weight: 0.2387
Component 03, Weight: 0.2994
Component 04, Weight: 0.1192
Samples


After Iteration 25
Component 00, Weight: 0.1762
Component 01, Weight: 0.1681
Component 02, Weight: 0.2155
Component 03, Weight: 0.2933
Component 04, Weight: 0.1469
Samples


Final Model
Component 00, Weight: 0.1764
Component 01, Weight: 0.1688
Component 02, Weight: 0.1976
Component 03, Weight: 0.2886
Component 04, Weight: 0.1686
Samples
3
2
1
0
−1
−2
−3
−4
−5
−3
−2
−1
0
1
2
3


Log-Likelihoods over time
log-likelihood
iteration
4.1
4.0
3.9
3.8
3.7
3.6
3.5
3.4
0
5
10
15
20
25
30