MITx - 686x_Project_4_CollaborativeFilteringviaGaussianMixtures

August 20, 2019

0.1 Project 4: Collaborative Filtering via Gaussian Mixtures

0.2 1. Introduction

Your task is to build a mixture model for collaborative filtering. You are given a data matrix containing movie ratings made by users where the matrix is extracted from a much larger Netflix database. Any particular user has rated only a small fraction of the movies so the data matrix is only partially filled. The goal is to predict all the remaining entries of the matrix.

You will use mixtures of Gaussians to solve this problem. The model assumes that each user's rating profile is a sample from a mixture model. In other words, we have K possible types of users and, in the context of each user, we must sample a user type and then the rating profile from the Gaussian distribution associated with the type. We will use the Expectation Maximization (EM) algorithm to estimate such a mixture from a partially observed rating matrix. The EM algorithm proceeds by iteratively assigning (softly) users to types (E-step) and subsequently re-estimating the Gaussians associated with each type (M-step). Once we have the mixture, we can use it to predict values for all the missing entries in the data matrix.

Setup: As with the last project, please use Python's NumPy numerical library for handling arrays and array operations; use matplotlib for producing figures and plots.

1. Note on software: For all the projects, we will use python 3.6 augmented with the NumPy numerical toolbox, the matplotlib plotting toolbox. In this project, we will also use the typing library, which is already included in the standard library (no need to install anything).
2. Download netflix.tar.gz and untar it in to a working directory. The archive contains the following python files:

- kmeans where we have implemented a baseline using the K-means algorithm
- naive_em.py where you will implement a first version of the EM algorithm (tabs 3-4)
- em.py where you will build a mixture model for collaborative filtering (tabs 7-8)
- common.py where you will implement the common functions for all models (tab 5)
- main.py where you will write code to answer the questions for this project
- test.py where you will write code to test your implementation of EM for a given test case

Additionnally, you are provided with the following data files: * toy_data.txt a 2D dataset that you will work with in tabs 2-5 * netflix_incomplete.txt the netflix dataset with missing entries to be completed * netflix_complete.txt the netflix dataset with missing entries completed * test_incomplete.txt a test dataset to test for you to test your code against our implementation * test_complete.txt a test dataset to test for you to test your code against our implementation * test_solutions.txt a test dataset to test for you to test your code against our implementation updated on August 8th

Tip: Throughout the whole online grading system, you can assume the NumPy python library is already imported as np. In some problems you will also have access to other functions you've already implemented. Look out for the "Available Functions" Tip before the codebox, as you did in the previous projects.

This project will unfold both on MITx and on your local machine. However, we encourage you to first implement the functions locally and run them to validate basic functionality. Think of the online graders as a submission box to submit your code when it is ready. You should not have to use the online graders to debug your code.

0.3 2. K-means

For this part of the project you will compare clustering obtained via K-means to the (soft) clustering induced by EM. In order to do so, our K-means algorithm will differ a bit from the one you learned. Here, the means are estimated exactly as before but the algorithm returns additional information. More specifically, we use the resulting clusters of points to estimate a Gaussian model for each cluster. Thus, our K-means algorithm actually returns a mixture model where the means of the component Gaussians are the K centroids computed by the K-means algorithm. This is to make it such that we can now directly plot and compare solutions returned by the two algorithms as if they were both estimating mixtures.

Read a 2D toy dataset using X = np.loadtxt('toy_data.txt'). Your task is to run the K-means algorithm on this data using the implementation we have provided in kmeans.py Initialize K-means using common.init(X, K, seed, where K is the number of clusters and seed is the random see used to randomly initialize the parameters.

Note that init(X,K) returns a K-component mixture model with means, variances and mixing proportions. The K-means algorithm will only care about the means, however, and returns a mixture that is retrofitted based on the K-means solution.

Try K=[1,2,3,4] on this data, plotting each solution using our common.plot function. Since the initialization is random, please use seeds 0,1,2,3,4 to and select the one that minimizes the total distortion cost. Save the associated plots (best solution for each K). The code for this task can be written in main.py.

common.py

[1]:
```
"""Mixture model for collaborative filtering"""
from typing import NamedTuple, Tuple
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.patches import Circle, Arc

class GaussianMixture(NamedTuple):
    """Tuple holding a gaussian mixture"""
    mu: np.ndarray  # (K, d) array - each row corresponds to a gaussian␣
 ↪component mean
    var: np.ndarray  # (K, ) array - each row corresponds to the variance of a␣
 ↪component
```

```
    p: np.ndarray  # (K, ) array = each row corresponds to the weight of a␣
↪component


def init(X: np.ndarray, K: int,
         seed: int = 0) -> Tuple[GaussianMixture, np.ndarray]:
    """Initializes the mixture model with random points as initial
    means and uniform assingments

    Args:
        X: (n, d) array holding the data
        K: number of components
        seed: random seed

    Returns:
        mixture: the initialized gaussian mixture
        post: (n, K) array holding the soft counts
            for all components for all examples

    """
    np.random.seed(seed)
    n, _ = X.shape
    p = np.ones(K) / K

    # select K random points as initial means
    mu = X[np.random.choice(n, K, replace=False)]
    var = np.zeros(K)
    # Compute variance
    for j in range(K):
        var[j] = ((X - mu[j])**2).mean()

    mixture = GaussianMixture(mu, var, p)
    post = np.ones((n, K)) / K

    return mixture, post


def plot(X: np.ndarray, mixture: GaussianMixture, post: np.ndarray,
         title: str):
    """Plots the mixture model for 2D data"""
    _, K = post.shape

    percent = post / post.sum(axis=1).reshape(-1, 1)
    fig, ax = plt.subplots()
    ax.title.set_text(title)
    ax.set_xlim((-20, 20))
    ax.set_ylim((-20, 20))
    r = 0.25
    color = ["r", "b", "k", "y", "m", "c"]
    for i, point in enumerate(X):
        theta = 0
        for j in range(K):
            offset = percent[i, j] * 360
            arc = Arc(point,
                      r,
                      r,
                      0,
                      theta,
                      theta + offset,
                      edgecolor=color[j])
            ax.add_patch(arc)
            theta += offset
    for j in range(K):
        mu = mixture.mu[j]
        sigma = np.sqrt(mixture.var[j])
        circle = Circle(mu, sigma, color=color[j], fill=False)
        ax.add_patch(circle)
        legend = "mu = ({:0.2f}, {:0.2f})\n stdv = {:0.2f}".format(
            mu[0], mu[1], sigma)
        ax.text(mu[0], mu[1], legend)
    plt.axis('equal')
    plt.show()


def rmse(X, Y):
    return np.sqrt(np.mean((X - Y)**2))
```

[2]:
```
def bic(X: np.ndarray, mixture: GaussianMixture,
        log_likelihood: float) -> float:
    """Computes the Bayesian Information Criterion for a
    mixture of gaussians

    Args:
        X: (n, d) array holding the data
        mixture: a mixture of spherical gaussian
        log_likelihood: the log-likelihood of the data

    Returns:
        float: the BIC for this mixture
    """
    raise NotImplementedError
```

kmeans.py

[3]:
```python
"""Mixture model based on kmeans"""
from typing import Tuple
import numpy as np
from common import GaussianMixture

def kestep(X: np.ndarray, mixture: GaussianMixture) -> np.ndarray:
    """E-step: Assigns each datapoint to the gaussian component with the
    closest mean

    Args:
        X: (n, d) array holding the data
        mixture: the current gaussian mixture

    Returns:
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples

        """
    n, _ = X.shape
    K, _ = mixture.mu.shape
    post = np.zeros((n, K))

    for i in range(n):
        tiled_vector = np.tile(X[i, :], (K, 1)) # cluster K
        sse = ((tiled_vector - mixture.mu)**2).sum(axis=1)  # distance to mu, K␣
↪cluster
        j = np.argmin(sse) # mim cluster
        post[i, j] = 1     # set flag 1 (assign cluster)

    return post
```

[4]:
```python
def kmstep(X: np.ndarray, post: np.ndarray) -> Tuple[GaussianMixture, float]:
    """M-step: Updates the gaussian mixture. Each cluster
    yields a component mean and variance.

    Args: X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
        float: the distortion cost for the current assignment
    """
    n, d = X.shape
    _, K = post.shape

    n_hat = post.sum(axis=0)
    p = n_hat / n

    cost = 0
    mu = np.zeros((K, d))
    var = np.zeros(K)

    for j in range(K):
        mu[j, :] = post[:, j] @ X / n_hat[j]
        sse = ((mu[j] - X)**2).sum(axis=1) @ post[:, j]
        cost += sse
        var[j] = sse / (d * n_hat[j])

    return GaussianMixture(mu, var, p), cost
```

[5]:
```
def krun(X: np.ndarray, mixture: GaussianMixture,
        post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray, float]:
    """Runs the mixture model

    Args:
        X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: distortion cost of the current assignment
    """

    prev_cost = None
    cost = None
    while (prev_cost is None or prev_cost - cost > 1e-4):
        prev_cost = cost
        post = kestep(X, mixture)
        mixture, cost = kmstep(X, post)

    return mixture, post, cost
```

run K Means

[6]:
```
import numpy as np
X = np.loadtxt("toy_data.txt")

# TODO: Your code here

def kmeans_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = init(X, k, s)
            _, _, c = krun(X, m, p)
            seed.setdefault(s, []).append(c)
        cost.setdefault(k, min(seed.values()))
    return cost

K = [1,2,3,4]
S = [0, 1, 2, 3, 4]
kmeans_cost(X, K, S)
```

```
[6]: {1: [5462.297452340001],
      2: [1684.9079502962372],
      3: [1329.5948671544297],
      4: [1035.4998265394659]}
```

0.4 3. Expectation–maximization algorithm

Recall the Gaussian mixture model presented in class:

$$P(x|) = \sum_{j=1}^{K} {}_j N(x; {}^{(j)}, {}_j^2 I)$$

where denotes all the parameters in the mixture (means ${}^{(j)}$, mixing proportions j, and variances ${}_j^2$). The goal of the EM algorithm is to estimate these unknown parameters by maximizing the log-likelihood of the observed data $x^{(1)}, \ldots, x^{(n)}$. Starting with some initial guess of the unknown parameters, the algorithm iterates between E- and M-steps. The E-Step softly assigns each data point $x^{(i)}$ to mixture components. The M-step takes these soft-assignments as given and finds a new setting of the parameters by maximizing the log-likelihood of the weighted dataset (expected complete log-likelihood).

Implement the EM algorithm for the Gaussian mixture model desribed above. To this end, complete the functions estep, mstep and run in naive_em.py. In our notation,

- X: an (n,d) Numpy array of n data points, each with d features
- K: number of mixture components
- mu: (K,d) Numpy array where the jth row is the mean vector ${}^{(j)}$
- p: (K,) Numpy array of mixing proportions j, j=1,…,K
- var: (K,) Numpy array of variances 2j, j=1,…,K

The convergence criteria that you should use is that the improvement in the log-likelihood is less than or equal to 10^6 multiplied by the absolute value of the new log-likelihood. In slightly more algebraic notation:

new log-likelihoodold log-likelihood 10^6 | new log-likelihood |

Your code will output updated versions of a GaussianMixture (with means mu, variances var and mixing proportions p) as well as an (n,K) Numpy array post, where post[i,j] is the posterior probability $p(j|x^{(i)})$, and LL which is the log-likelihood of the weighted dataset.

Here are a few points to check to make sure that your implementation is indeed correct:

1. Make sure that all your functions return objects with the right dimension.
2. EM should monotonically increase the log-likelihood of the data. Initialize and run the EM algorithm on the toy dataset as you did earlier with K-means. You should check that the LL values that the algorithm returns after each run are indeed always monotonically increasing (non-decreasing).
3. Using K=3 and a seed of 0 , on the toy dataset, you should get a log likelihood of -1388.0818.
4. As a runtime guideline, in your testing on the toy dataset, calls of run using the values of K that we are testing should run in on the order of seconds (i.e. if each call isn't fairly quick, that may be an indication that something is wrong).
5. Try plotting the solutions obtained with your EM implementation. Do they make sense?

0.4.1 Implementing E-step

Write a function estep that performs the E-step of the EM algorithm

Available Functions: You have access to the NumPy python library as np, to the GaussianMixture class and to typing annotation typing.Tuple as Tuple

[7]:
```
"""Mixture model for matrix completion"""
from typing import Tuple
import numpy as np
from scipy.special import logsumexp
from common import GaussianMixture
```

[8]:
```
def em_estep(X: np.ndarray, mixture: GaussianMixture) -> Tuple[np.ndarray,␣
 ↪float]:
    """E-step: Softly assigns each datapoint to a gaussian component

    Args:
        X: (n, d) array holding the data, with incomplete entries (set to 0)
        mixture: the current gaussian mixture

    Returns:
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the assignment

    """

    n, d = X.shape
    K, _ = mixture.mu.shape
    post = np.zeros((n, K))
    log_likelihood = np.zeros((n, 1))

    def normal_pdf(x: np.ndarray, mu: np.ndarray, var: float):
        x_mu = np.matrix(x-mu)
        norm_const = 1.0 / ( (2*np.pi*var)**(d/2) )
        norm_exp = np.exp( -1.0 / (2*var) * np.linalg.norm(x_mu)**2 )
        return norm_const * norm_exp
    # calculate post: (n, k) array holding the soft counts
    for i in range(n):
        tiled_vector = np.tile(X[i, :], (K, 1))
        for j in range(K):
            pij = mixture.p[j]*normal_pdf(tiled_vector[j,:], mixture.mu[j],␣
↪mixture.var[j])
            post[i, j] = pij
        log_likelihood[i] = np.log(sum(post[i,:]))
        post[i,:] = post[i,:]/sum(post[i,:])

    return post, sum(log_likelihood)
```

Implementing M-step Write a function mstep that performs the M-step of the EM algorithm

Available Functions: You have access to the NumPy python library as np, to the GaussianMixture class and to typing annotation typing.Tuple as Tuple

[9]:
```
def em_mstep(X: np.ndarray, post: np.ndarray, mixture: GaussianMixture,
          min_variance: float = .25) -> GaussianMixture:
    """M-step: Updates the gaussian mixture by maximizing the log-likelihood
    of the weighted dataset

    Args:
        X: (n, d) array holding the data, with incomplete entries (set to 0)
        post: (n, K) array holding the soft counts
            for all components for all examples
        mixture: the current gaussian mixture
        min_variance: the minimum variance for each gaussian

    Returns:
        GaussianMixture: the new gaussian mixture
    """
    n, d = X.shape
    _, K = post.shape

    n_hat = post.sum(axis=0)
    p = n_hat / n

    mu = np.zeros((K, d))
    var = np.zeros(K)

    for j in range(K):
        mu[j, :] = post[:, j] @ X / n_hat[j]
        sse = ((mu[j] - X)**2).sum(axis=1) @ post[:, j]
        var[j] = sse / (d * n_hat[j])

    return GaussianMixture(mu, var, p)
```

Implementing run Write a function run that runs the EM algorithm. The convergence criterion you should use is described above.

Available Functions: You have access to the NumPy python library as np, to the GaussianMixture class and to typing annotation typing.Tuple as Tuple. You also have access to the estep and mstep functions you have just implemented

[10]:
```python
def em_run(X: np.ndarray, mixture: GaussianMixture,
        post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray, float]:
    """Runs the mixture model

    Args:
        X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the current assignment
    """

    n, d = X.shape
    _, K = post.shape

    post, prev_cost = em_estep(X, mixture)
    mixture = em_mstep(X, post, mixture)
    post, cost = em_estep(X, mixture)

    while (cost - prev_cost >= 1e-6 * abs(cost)):
        prev_cost = cost
        mixture = em_mstep(X, post, mixture)
        post, cost = em_estep(X, mixture)

    return mixture, post, cost
```

test EM

[11]:
```python
X = np.loadtxt("toy_data.txt")

def em_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = init(X, k, s)
            _, _, c = em_run(X, m, p)
            seed.setdefault(s, []).append(c)
        cost.setdefault(k, max(seed.values()))
    return cost


K = 3
n, d = X.shape
seed = 0

em_cost(X, [K], [seed])
```

[11]:
```
{3: [array([-1138.89089969])]}
```

0.5 4. Comparing K-means and EM

Generate analogous plots to K-means using your EM implementation. Note that the EM algorithm can also get stuck in a locally optimal solution. For each value of K, please run the EM algorithm with seeds 0,1,2,3,4 and select the solution that achieves the highest log-likelihood. Compare the K-means and mixture solutions for K=[1,2,3,4] . Ask yourself when, how, and why they differ.

0.5.1 Reporting log likelihood values

Report the maximum likelihood for each K using seeds 0,1,2,3,4:
Log-likelihood | K=1 = -1307.22343176
Log-likelihood | K=2 = -1175.71462937
Log-likelihood | K=3 = -1138.89089969
Log-likelihood | K=4 = -1138.6011757

[12]:
```
X = np.loadtxt("toy_data.txt")

def em_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = init(X, k, s)
            _, _, c = em_run(X, m, p)
            seed.setdefault(s, []).append(c)
        cost.setdefault(k, max(seed.values()))
    return cost

K = [1,2,3,4]
S = [0, 1, 2, 3, 4]

for k in K:
    print(em_cost(X, [k], S))
```

```
{1: [array([-1307.22343176])]}
{2: [array([-1175.71462937])]}
```

```
{3: [array([-1138.89089969])]}
{4: [array([-1138.6011757])]}
```

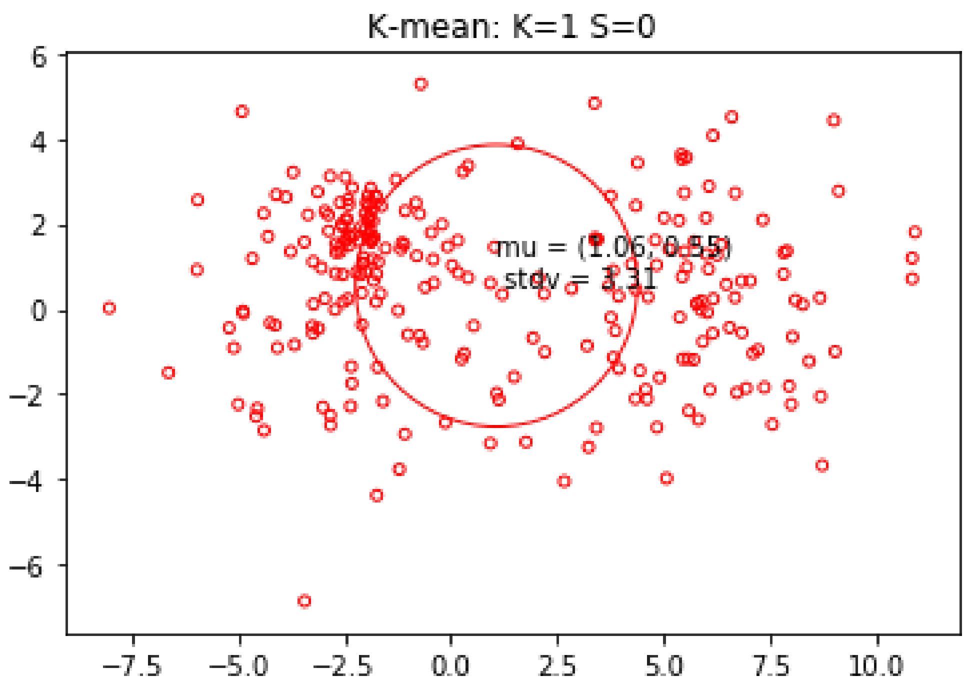
0.5.2 Analysing plots

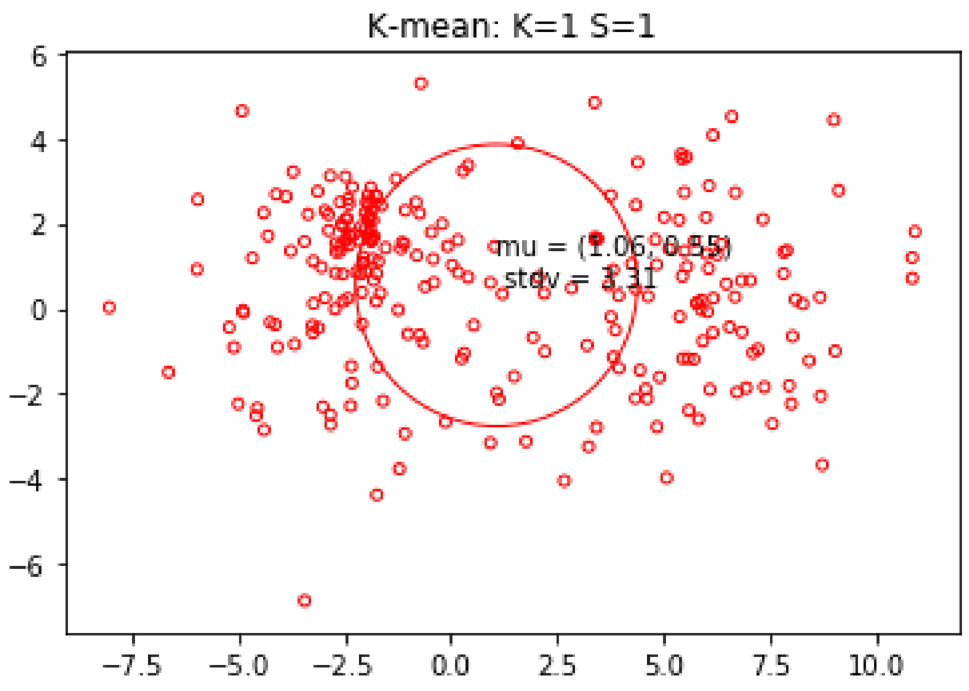
Which of the following sentences are true? (Check all that apply)

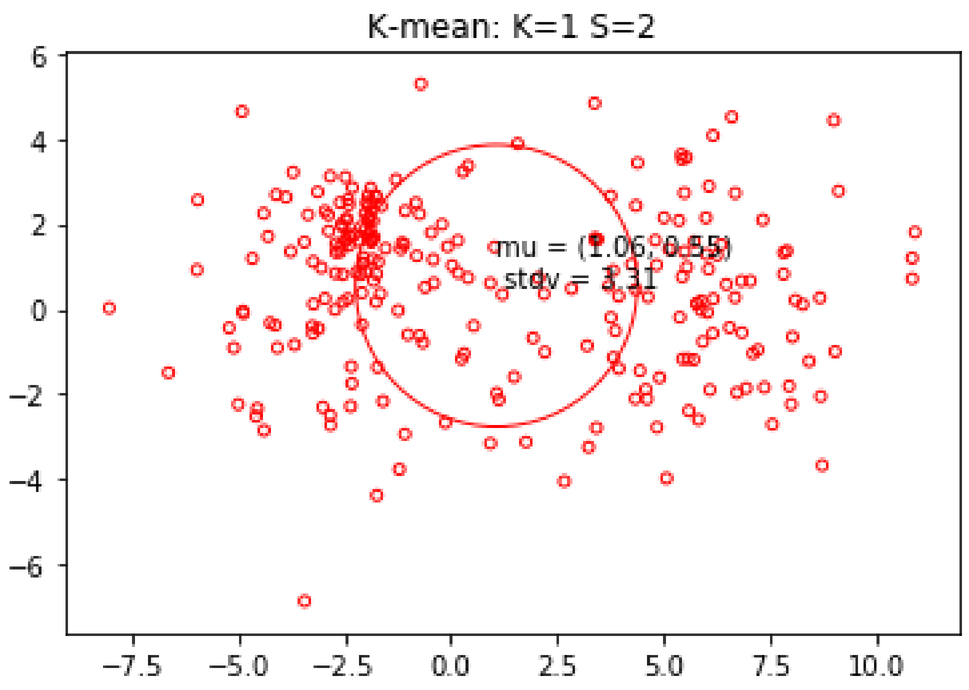
- In the case K=1, the mixture parameters and point assignments are the same for both methods
- In the case K=2, both methods have simlilar parameters and point assignments
- In the case K=3, the k-means solution accounts for point density better than EM
- In the case K=4, the k-means solution equally spaces the clusters to minimize distortion cost

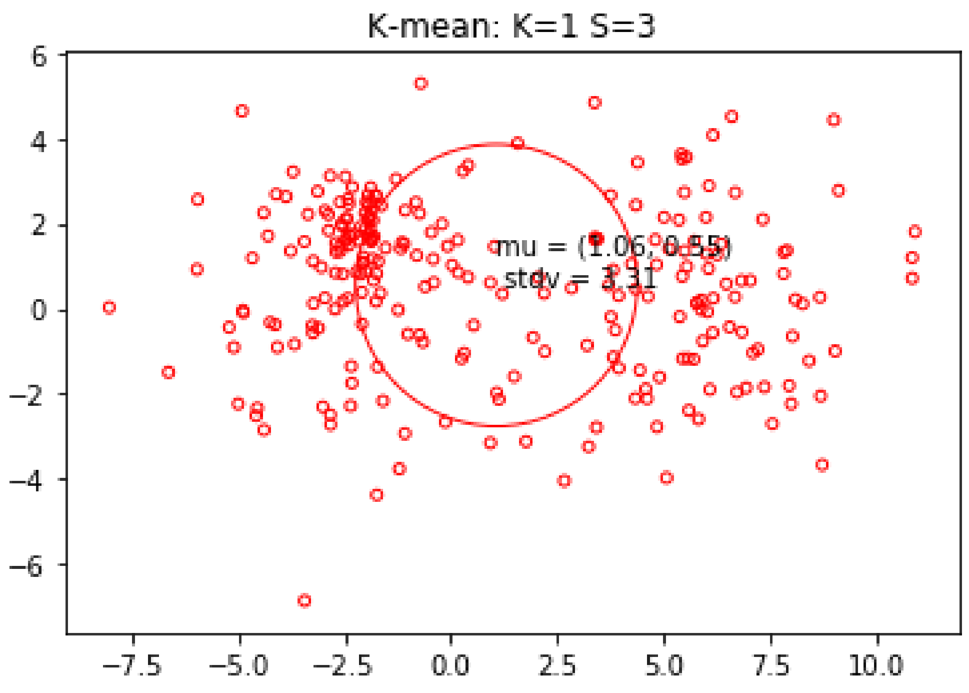
[13]:
```python
X = np.loadtxt("toy_data.txt")

def k_plot(X, K, S) -> dict:
    for k in K:
        for s in S:
            m, p = init(X, k, s)
            m, p, c = krun(X, m, p)
            title= 'K-mean: K=' + str(k) + ' S=' + str(s)
            plot(X, m, p, title)

def em_plot(X, K, S) -> dict:
    for k in K:
        for s in S:
            m, p = init(X, k, s)
            m, p, c = em_run(X, m, p)
            title= 'EM: K=' + str(k) + ' S=' + str(s)
            plot(X, m, p, title)

S = [0,1,2,3,4]
```

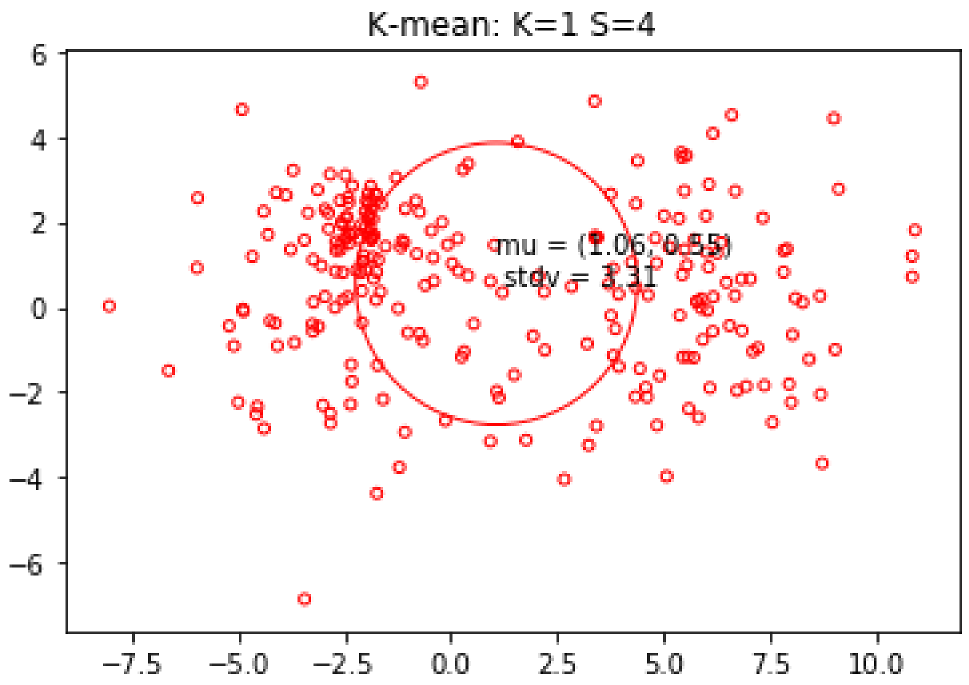
[14]:
```python
k_plot(X, [1], S)
```


K-mean: K=1 S=0
mu = (1.06, 0.55)
stdv = 3.31
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

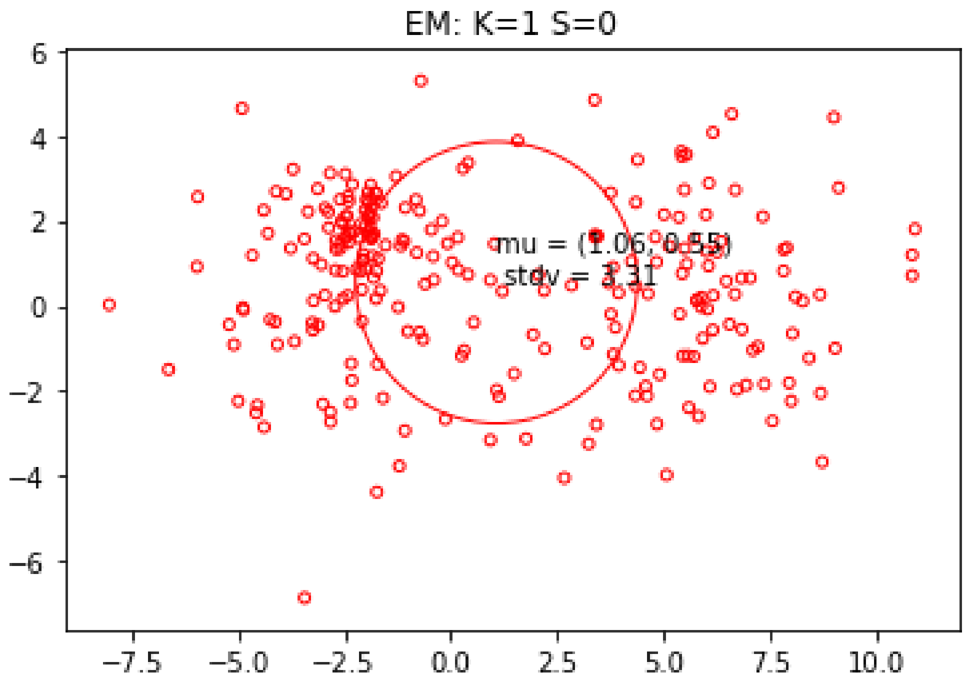

K-mean: K=1 S=1
mu = (1.06, 0.55)
stdv = 3.31
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

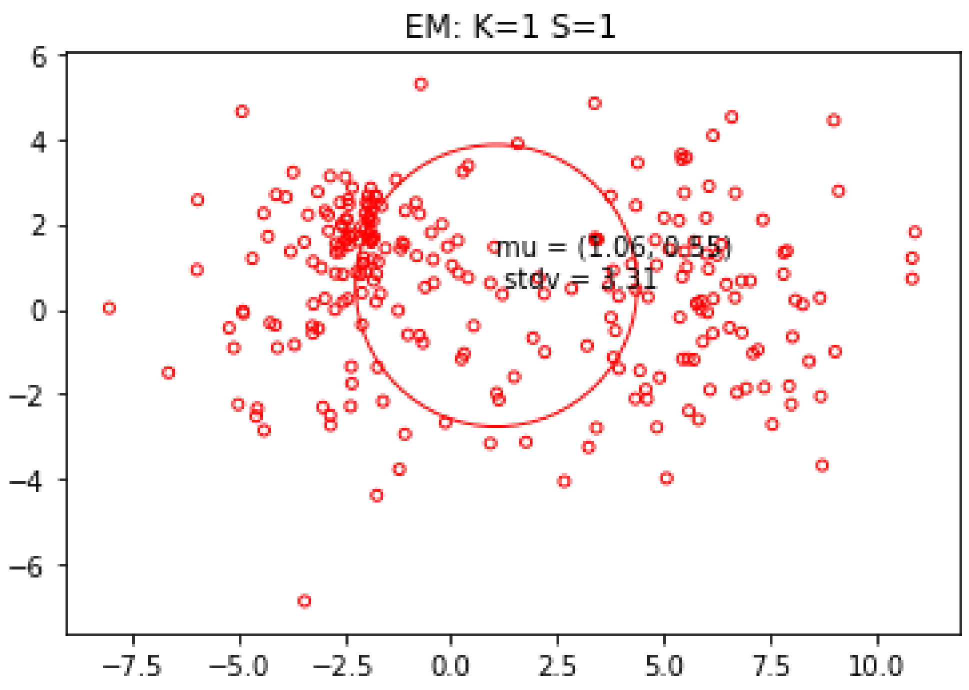

K-mean: K=1 S=2
mu = (1.06, 0.55)
stdv = 3.31
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

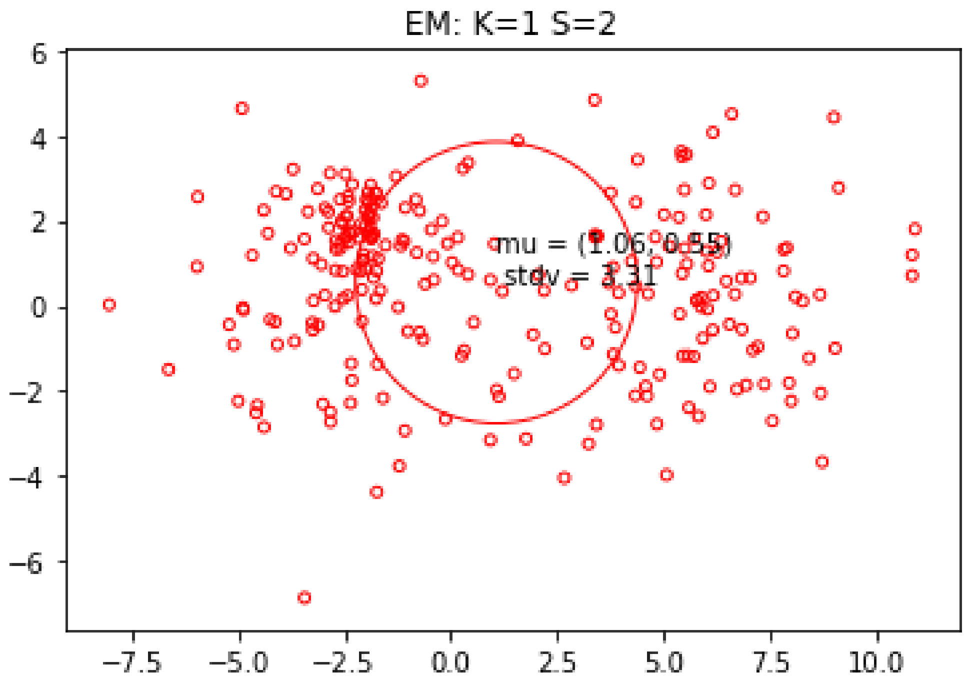

K-mean: K=1 S=3
mu = (1.06, 0.55)
stdv = 3.31
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

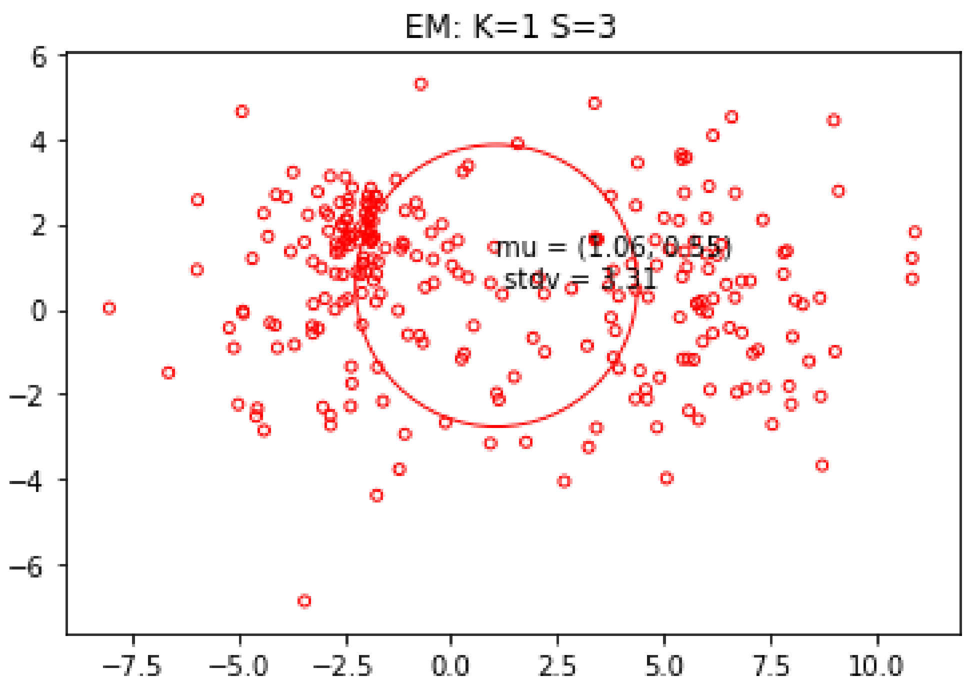


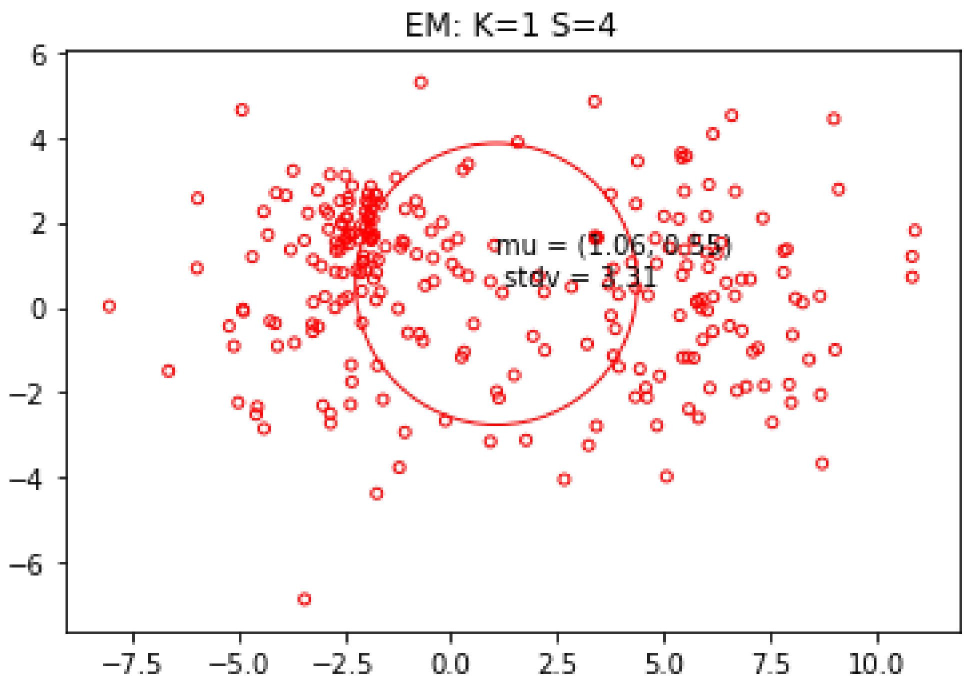
[15]:
```
em_plot(X, [1], S)
```




EM: K=1 S=1
mu = (1.06, 0.55)
stdv = 3.31


EM: K=1 S=2
mu = (1.06, 0.55)
stdv = 3.31





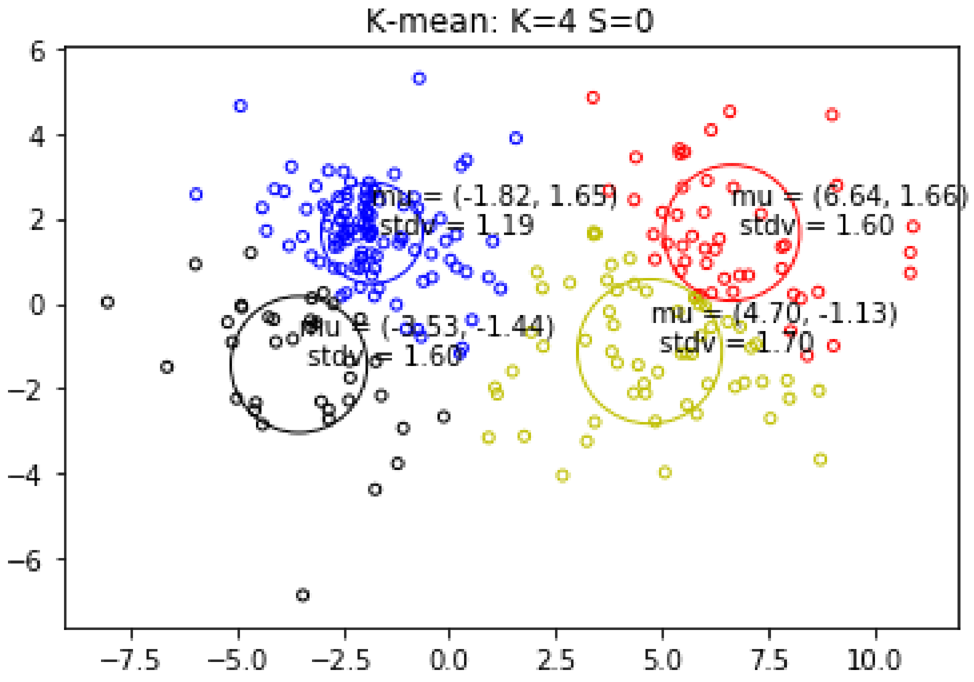
[ ]:
```
k_plot(X, [2], S)
```

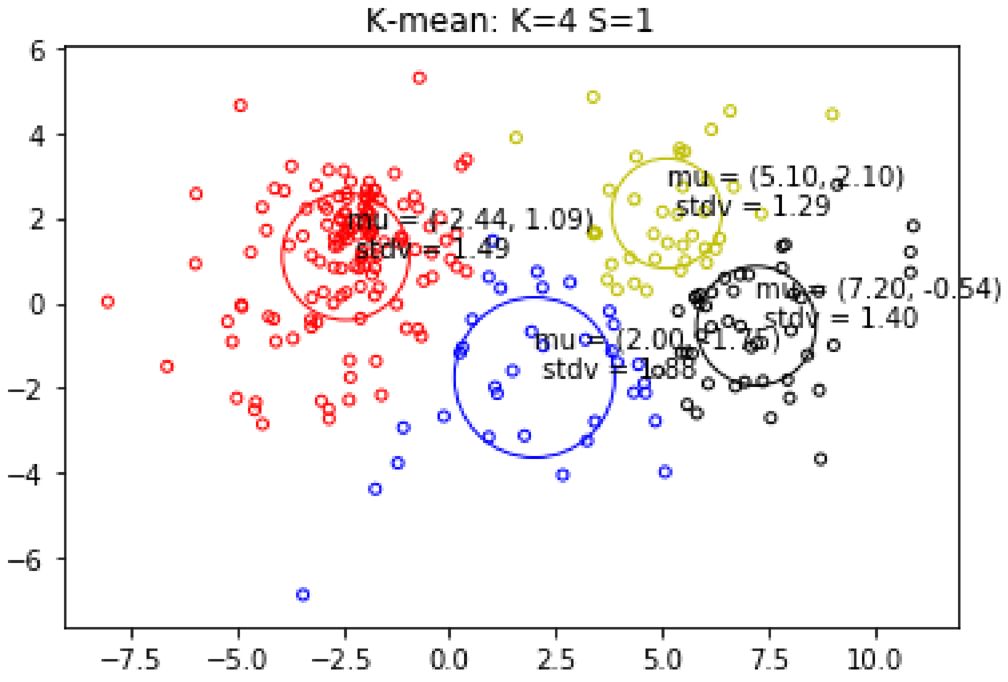
[ ]:
```
em_plot(X, [2], S)
```

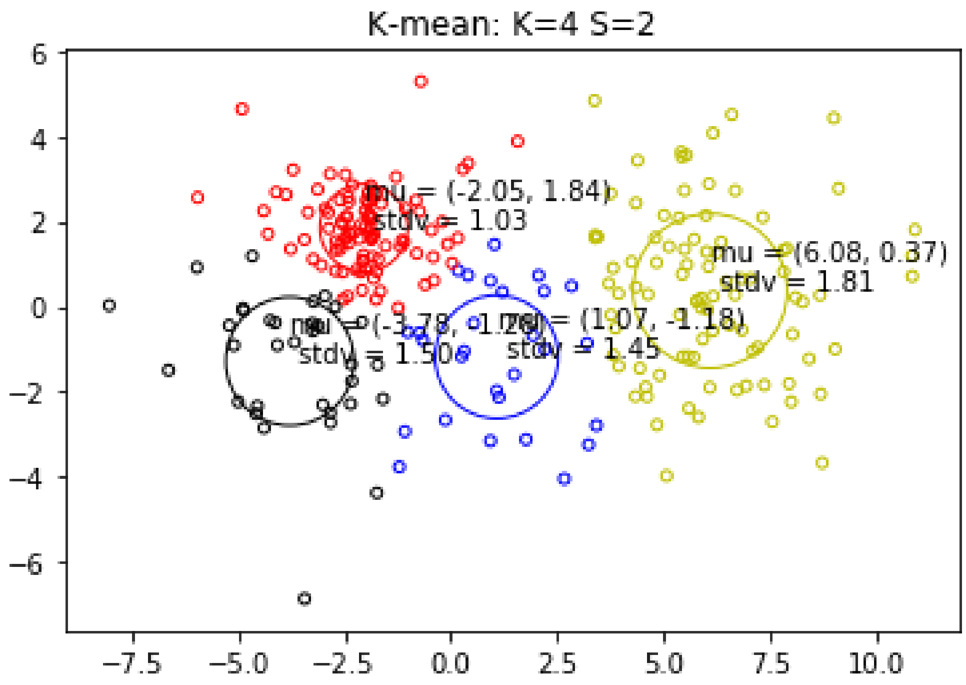
[ ]:
```
k_plot(X, [3], S)
```

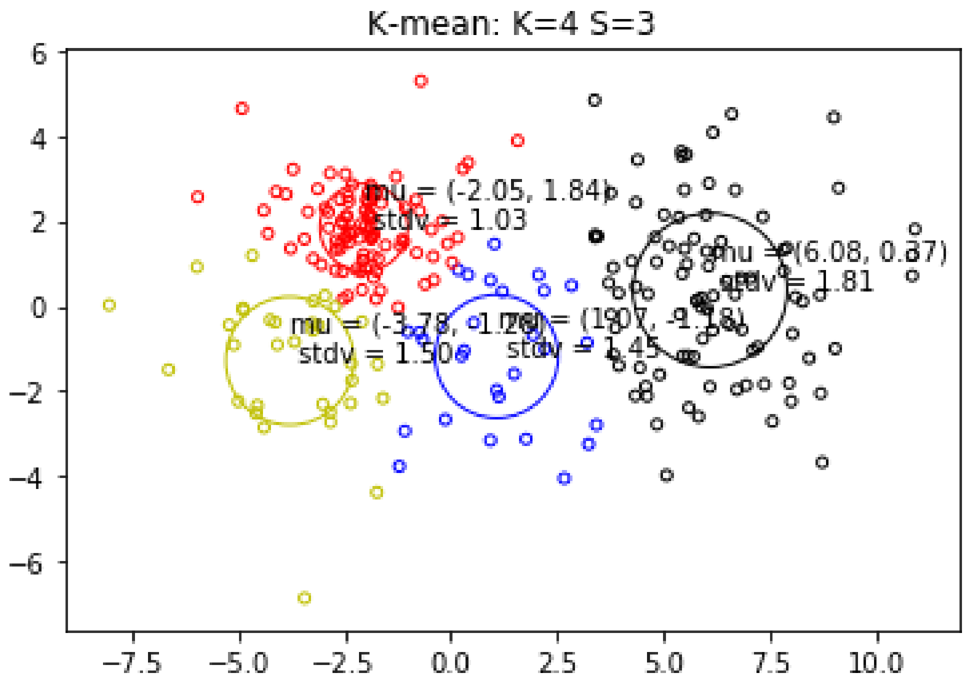
[ ]:
```
em_plot(X, [3], S)
```

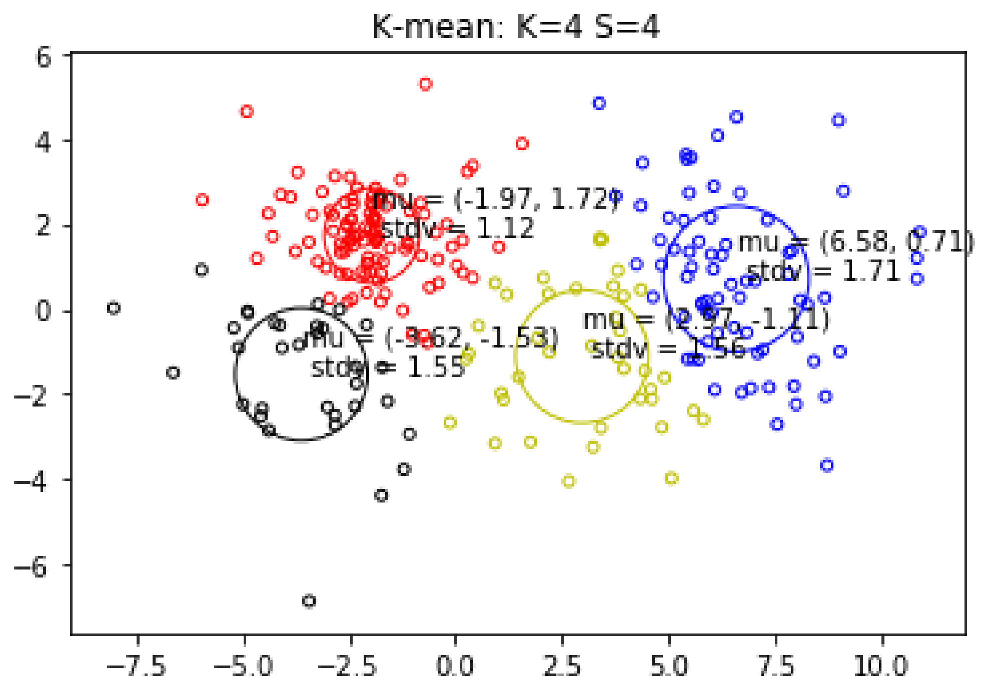
[16]:
```
k_plot(X, [4], S)
```



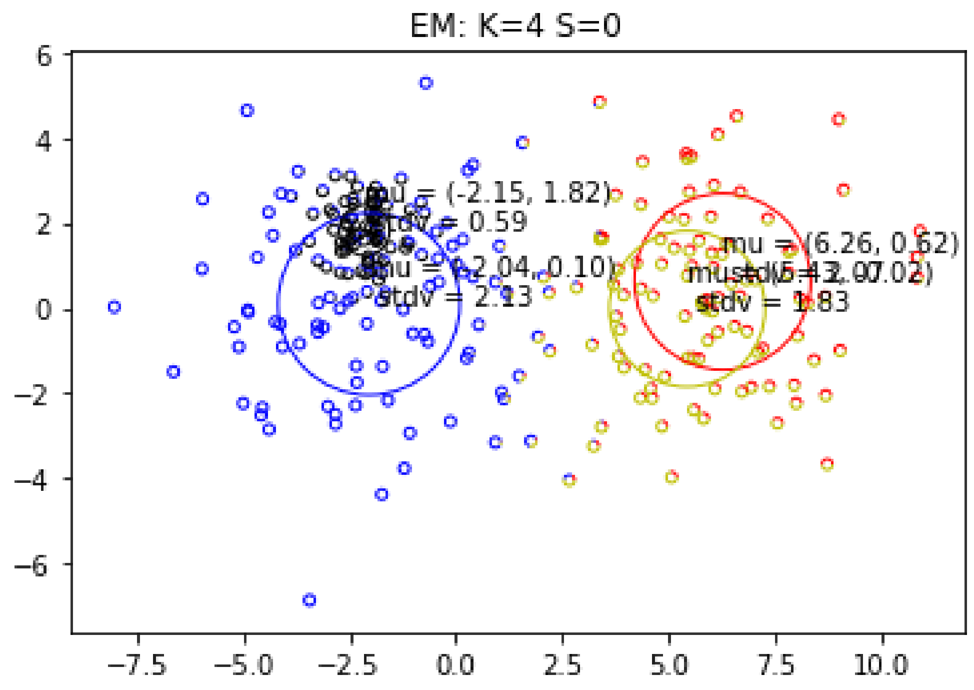


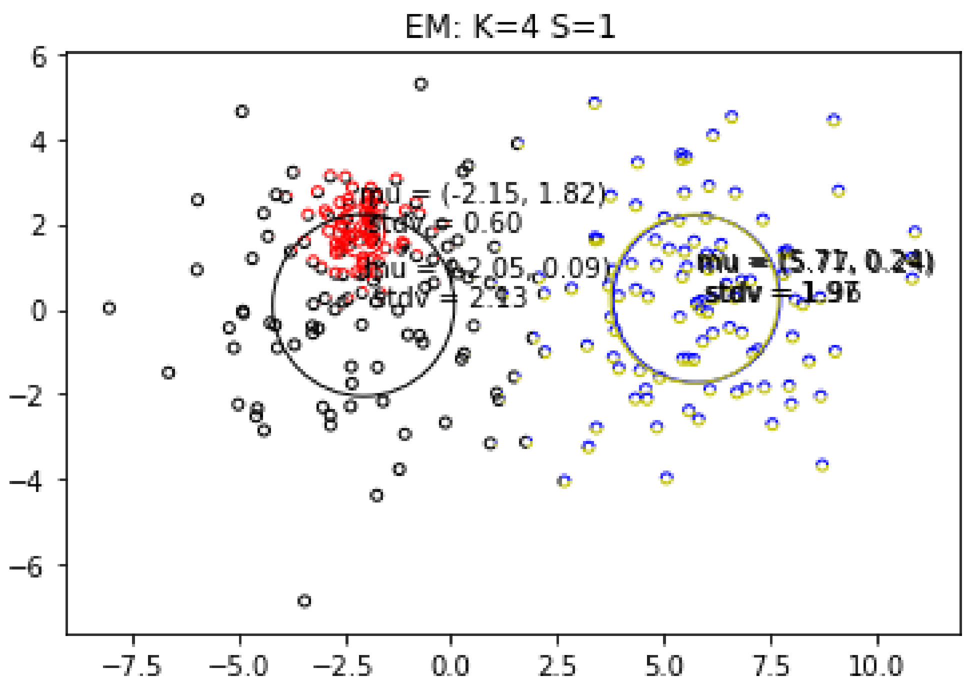

K-mean: K=4 S=2
mu = (-2.05, 1.84)
stdv = 1.03
mu = (6.08, 0.37)
stdv = 1.81
mu = (-3.78, -1.28)
stdv = 1.50
mu = (1.07, -1.18)
stdv = 1.45

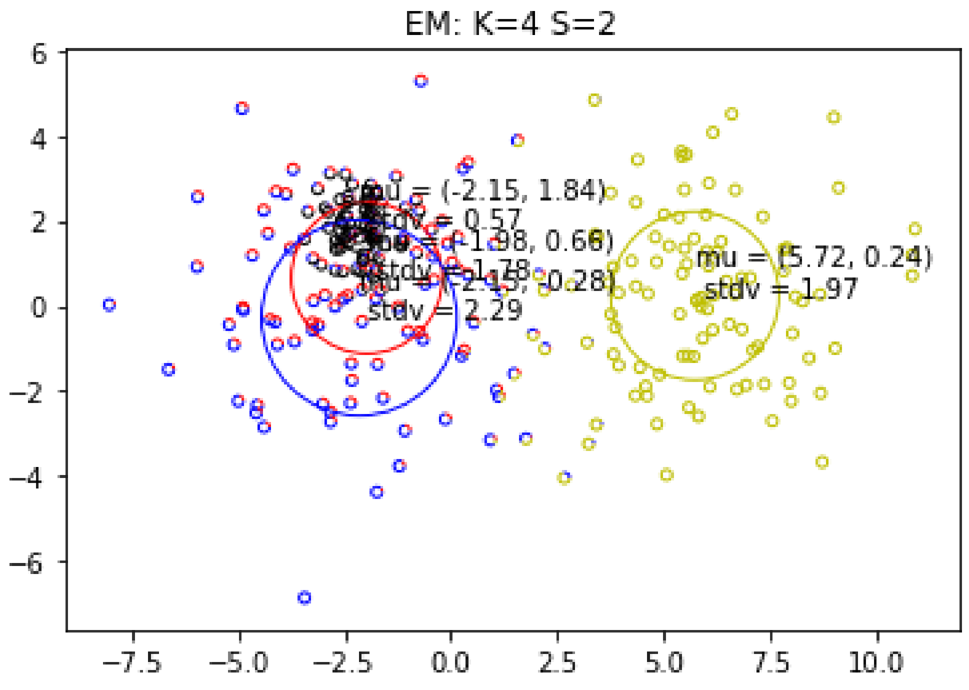

K-mean: K=4 S=3
mu = (-2.05, 1.84)
stdv = 1.03
mu = (6.08, 0.37)
stdv = 1.81
mu = (-3.78, -1.28)
stdv = 1.50
mu = (1.07, -1.18)
stdv = 1.45

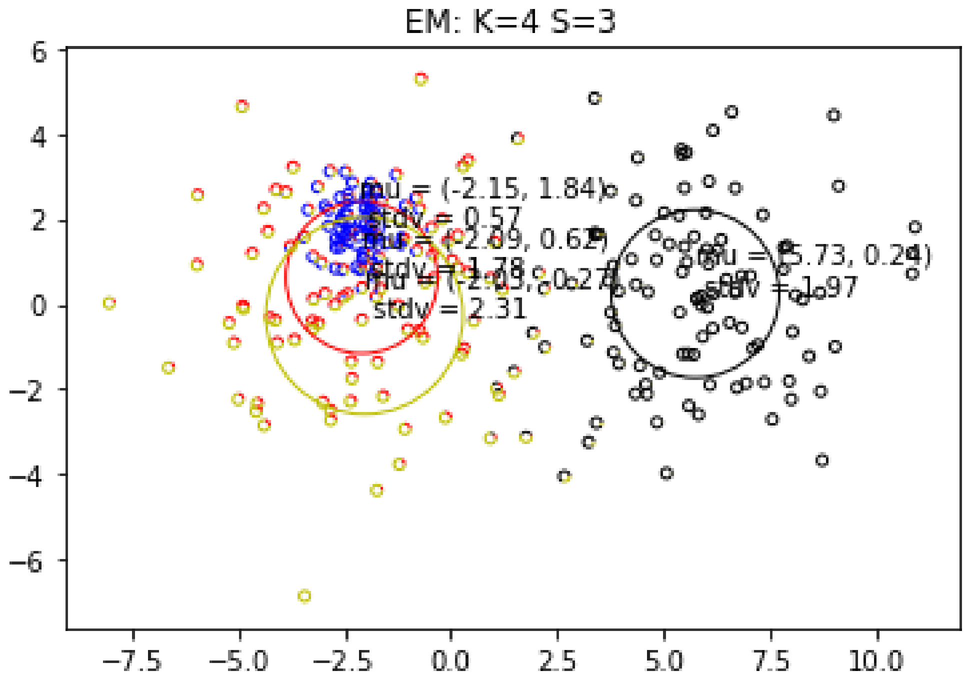


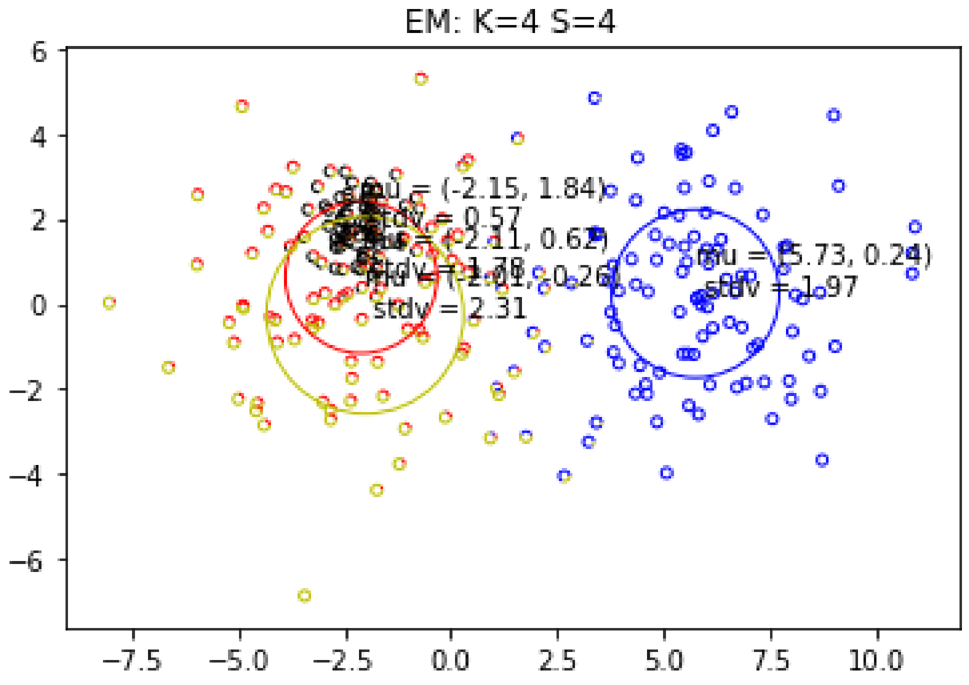
[17]:
```
em_plot(X, [4], S)
```




EM: K=4 S=1
mu = (-2.15, 1.82)
stdv = 0.60
mu = (-2.05, 0.09)
stdv = 2.13
mu = (5.71, 0.24)
stdv = 1.97
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


EM: K=4 S=2
mu = (-2.15, 1.84)
stdv = 0.57
mu = (-1.98, 0.66)
stdv = 1.78
mu = (-2.15, -0.28)
stdv = 2.29
mu = (5.72, 0.24)
stdv = 1.97
6
4
2
0
−2
−4
−6
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


EM: K=4 S=3
mu = (-2.15, 1.84)
stdv = 0.57
mu = (-2.09, 0.62)
stdv = 1.78
mu = (-2.03, -0.27)
stdv = 2.31
mu = (5.73, 0.24)
stdv = 1.97


EM: K=4 S=4
mu = (-2.15, 1.84)
stdv = 0.57
mu = (-2.11, 0.62)
stdv = 1.78
mu = (-2.01, -0.26)
stdv = 2.31
mu = (5.73, 0.24)
stdv = 1.97

0.6 5. Bayesian Information Criterion

So far we have simply set the number of mixture components K but this is also a parameter that we must estimate from data. How does the log-likelihood of the data vary as a function of K assuming we avoid locally optimal solutions?

To compensate, we need a selection criterion that penalizes the number of parameters used in the model. The Bayesian information criterion (BIC) is a criterion for model selection. It captures the tradeoff between the log-likelihood of the data, and the number of parameters that the model uses. The BIC of a model M is defined as:

$$BIC(M) = l - \frac{1}{2} p \log n$$

where l is the log-likelihood of the data under the current model (highest log-likelihood we can achieve by adjusting the parameters in the model), p is the number of adjustable parameters, and n is the number of data points. This score rewards a larger log-likelihood, but penalizes the number of parameters used to train the model. In a situation where we wish to select models, we want a model with the the highest BIC.

0.6.1 Implementing the Bayesian Information Criterion

Fill in the missing Bayesian Information Criterion (BIC) calculation (bic function) in common.py.

Available Functions: You have access to the NumPy python library as np, to the GaussianMixture class and to typing annotation typing.Tuple as Tuple.

[18]:
```
def bic(X: np.ndarray, mixture: GaussianMixture,
        log_likelihood: float) -> float:
    """Computes the Bayesian Information Criterion for a
    mixture of gaussians

    Args:
        X: (n, d) array holding the data
        mixture: a mixture of spherical gaussian
        log_likelihood: the log-likelihood of the data

    Returns:
        float: the BIC for this mixture
    """
    n, _ = X.shape
    K, d = mixture.mu.shape
    adjustable_parameters = K*d + (K-1) + K  #mu, var, p
    bic_value = log_likelihood - 1.0/2*adjustable_parameters*np.log(n)

    return bic_value
```

0.6.2 Picking the best K

Find the best K from [1,2,3,4] on the toy dataset. This will be the K that produces the optimal BIC score. Report the best K and the corresponding BIC score. Does the criterion select the correct number of clusters for the toy data?

Best K = 3
Best BIC = 1169.25893474
Grader note: While the best BIC should be a negative value, due to earlier grader error, we have corrected the grader to accept both the positive and the negative value.

[20]:
```
def em_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = init(X, k, s)
            _, _, c = em_run(X, m, p)
            seed.setdefault(s, []).append(c)
        log_likelihood = bic(X, m, max(seed.values()))
        cost.setdefault(k, log_likelihood)
    return cost

K = [1,2,3,4]
S = [0, 1, 2, 3, 4]
for k in K:
    print(em_cost(X, [k], S))
```

```
{1: array([[-1315.50562314]])}
{2: array([[-1195.03974258]])}
{3: array([[-1169.25893474]])}
{4: array([[-1180.01213258]])}
```

0.7 6. Mixture models for matrix completion

We can now extend our Gaussian mixture model to predict actual movie ratings. Let X again denote the (n,d) data matrix. The rows of this matrix correspond to users and columns specify movies so that X[u,i] gives the rating value of user u for movie i (if available). Both n and d are typically quite large. The ratings range from one to five stars and are mapped to integers {1,2,3,4,5}. We will set X[u,i]=0 whenever the entry is missing. In a realistic setting, most of the entries of X are missing.

For this reason, we define C_u as the set of movies (column indexes) that user u has rated and H_u as its complement (the set of remaining unwatched/unrated movies we wish to predict ratings for). We use $|C_u|$ to denote the number of observed rating values from user u. From the point of view of our mixture model, each user u is an example $x^{(u)} = X[u,:]$. But since most of the coordinates of $x^{(u)}$ are missing, we need to focus the model during training on just the observed portion. To this end, we use $x^{(u)}_{C_u} = \{x^{(u)}_i : i \in C_u\}$ as the vector of only observed ratings. If columns are indexed as $\{0,\ldots d-1\}$, then a user u with a rating vector $x^{(u)} = (5,4,0,0,2)$, where zeros indicate missing values, has $C_u = \{0,1,4\}$, $H_u = \{2,3\}$, and $x^{(u)}_{C_u} = (5,4,2)$.

In this part, we will extend our mixture model in two key ways.

- First, we are going to estimate a mixture model based on partially observed ratings. See notes below.

- Second, since we will be dealing with a large, high-dimensional data set, we will need to be more mindful of numerical underflow issues. To this end, you should perform most of your computations in the log domain. Remember, $log(ab) = log(a) + log(b)$. This can be useful to remember when a and b are very small – in these cases, addition should result in fewer numerical underflow issues than multiplication.

An additional numerical optimization trick that you will find useful is the LogSumExp trick. Assume that we wish to evaluate

$$y = log(exp(x_1) + \dots exp(x_n))$$

. We define

$$x^\star = max\{x_1 \dots x_n\}$$

. Then,

$$y = x^\star + log(exp(x_1 x^\star) + \dots exp(x_n x^\star))$$

. This is just another trick to help ensure numerical stability.

0.7.1 Marginalizing over unobserved coordinates

If $x^{(u)}$ were a complete rating vector, the mixture model from Part 1 would simply say that $P(x^{(u)}|\theta) = \sum_{j=1}^{K} \pi_j N(x^{(u)}; \mu^{(j)}, \sigma_j^2 I)$. In the presence of missing values, we must use the marginal probability $P(x_{C_u}^{(u)}|\theta)$ that is over only the observed values. This marginal corresponds to integrating the mixture density $P(x^{(u)}|\theta)$ over all the unobserved coordinate values. In our case, this marginal can be computed as follows.

The mixture model for a complete rating vector is written as:

$$P(x^{(u)}|\theta) = \sum_{j=1}^{K} p_j N(x^{(u)}; \mu^{(j)}, \sigma_j^2 I)$$

We can decompose the multivariate spherical Gaussian as a product of univariate Gaussians (since there is no covariance between coordinates).

$$P(x^{(u)}|\theta) = \sum_{j=1}^{K} p_j \prod_{i} N(x_i^{(u)}; \mu_i^{(j)}, \sigma_i^{2,(j)}) = \sum_{j=1}^{K} p_j \prod_{mC_u} N(x_m^{(u)}; \mu_m^{(j)}, \sigma_m^{2,(j)}) \prod_{m' \in H_u} N(x_{m'}^{(u)}; \mu_{m'}^{(j)}, \sigma_{m'}^{2,(j)})$$

For $m^{'} \in H_u$, we can marginalize over all of the unobserved values to get

$$\int N(x_{m'}^{(u)}; \mu_{m'}^{(j)}, \sigma_{m'}^{2,(j)}) dx_{m'}^{(u)} = 1$$

Thus, our mixture density can be written as

$$P(x_{C_u}^{(u)}|\theta) = \sum_{j=1}^{K} p_j N(x_{C_u}^{(u)}; \mu_{C_u}^{(j)}, \sigma_j^2 I_{C_u x C_u})$$

0.8 7. Implementing EM for matrix completion

We need to update our EM algorithm a bit to deal with the fact that the observations are no longer complete vectors. We use Bayes' rule to find an updated expression for the posterior probability $p(j|u) = P(y = j|x^{u)}_{C_u})$:

$$p(j|u) = \frac{p(u|j)p(j)}{p(u)} = \frac{p(u|j)p(j)}{\sum_{j=1}^{K} p(u|j)p(j)} = \frac{\pi_j N(x^{(u)}_{C_u}; \mu^{(j)}_{C_u}, \sigma_j^2 I_{C_u x C_u})}{\sum_{j=1}^{K} \pi_j N(x^{(u)}_{C_u}; \mu^{(j)}_{C_u}, \sigma_j^2 I_{C_u x C_u})}$$

This is the soft assignment of cluster u to data point j.

E Step To minimize numerical instability, you will be re-implementing the E-step in the log-domain, so you should calculate the values for the log of the posterior probability, $l(j,u) = log(p(j|u))$ (though the actual output of your E-step should include the non-log posterior).

Let

$$f(u,i) = log(\pi_i) + log(N(x^{(u)}_{C_u}; \mu^{(i)}_{C_u}, \sigma_i^2 I_{C_u x C_u}))$$

. Then, in terms of f, the log posterior is:

$$l(j,u) = log(p(j|u)) = log(\frac{\pi_j N(x^{(u)}_{C_u}; \mu^{(j)}_{C_u}, \sigma_j^2 I_{C_u x C_u})}{\sum_{j=1}^{K} \pi_j N(x^{(u)}_{C_u}; \mu^{(j)}_{C_u}, \sigma_j^2 I_{C_u x C_u})}) = f(u,j) - log(\sum_{j=1}^{K} exp(f(u,j)))$$

Log likelihood Once we have evaluated $p(j|u)$ in the E-step, we can proceed to the M-step. We wish to find the parameters π, μ, and σ that maximize $l(x;\theta)$, the expected complete $log-likelihood$:

$$l(X;\theta) = \sum_{u=1}^{n} log(\sum_{j=1}^{K} \pi_j N(x^{(u)}_{C_u} | \mu^{(i)}_{C_u}, \sigma_i^2 I_{|C_u| x |C_u|}))$$

.

M Step To maximize $l(X;\theta)$, we keep $p(j|u)$ (the soft-assignments) fixed, and maximize over the model parameters. Some of the parameters can be updated exactly as before with complete example vectors. For example,

$$\hat{\pi}_j = \frac{\sum_{u=1}^{n} p(j|u)}{n}$$

But we must be more careful in updating $\mu^{(j)}$ and σ_j^2. This is because the parameters appear differently in the likelihood depending on how incomplete the observation is. Notice that some coordinates of $\mu^{(j)}$ do not impact observation $x^{(u)}_{C_u}$ at all. But we can proceed to separately update each coordinate of $\mu^{(j)}$.

We will take the derivative with respect to the to the l^{th} movie coordinate for cluster k.

First, note that, by decomposing the multivariate spherical Gaussians into univariate spherical Gaussians as before, we can write, if $k \in C_u$:

$$\frac{\delta}{\delta\mu_l^{(k)}}N(x_{C_u}^{(u)}|\mu_{C_u}^{(k)},\sigma_k^2 I_{|C_u|x|C_u|}) = N(\dots)\frac{\frac{\delta}{\delta\mu_l^{(k)}}(\frac{1}{\sqrt{2\pi}\sigma_{l_{(k)}}}exp(-\frac{1}{2\sigma_{l_{(k)}}^2}(x_l^{(u)}-\mu_j^{(k)})^2))}{\frac{1}{\sqrt{2\pi}\sigma_{l_{(k)}}}exp(-\frac{1}{2\sigma_{l_{(k)}}^2}(x_l^{(u)}-\mu_j^{(k)})^2))} = N(\dots)\frac{x_l^{(u)}-\mu_l^{(k)}}{\sigma_{l_{(k)}}^2}$$

where $N(\dots) = N(x_{Cu}^{(u)}|\mu_{Cu}^{(k)},\sigma_k^2 I_{|C_u|x|C_u|})$

if $k \notin C_u$ that derivative is 0. To cover both cases, we can write:

$$\frac{\delta}{\delta_{\mu_j}^{(k)}}N(x_{C_u}^{(u)}|\mu_{C_u}^{(k)},\sigma_k^2 I_{|C_u|x|C_u|}) = N(x_{C_u}^{(u)}|\mu_{C_u}^{(k)},\sigma_k^2 I_{|C_u|x|C_u|})\delta(l,C_u)\frac{x_l^{(u)}-\mu_l^{(k)}}{\sigma_{l_{(k)}}^2}$$

where $\delta(i,C_u)$ is an indicator function: 1 if $i \in C_u$ and zero otherwise.

Therefore

$$\frac{\delta l(X;\theta)}{\delta\mu_l^{(k)}} = \sum_{u=1}^{n}\frac{\frac{\delta}{\delta\mu_j^{(k)}}\sum_{j=1}^{K}\pi_j N(x_{C_u}^{(u)}|\mu_{C_u}^{(i)},\sigma_i^2 I_{|C_u|x|C_u|})}{\sum_{j=1}^{K}\pi_j N(x_{C_u}^{(u)}|\mu_{C_u}^{(i)},\sigma_i^2 I_{|C_u|x|C_u|})} = \sum_{u=1}^{n}\frac{\pi_k N(x_{C_u}^{(u)}|\mu_{C_u}^{(i)},\sigma_k^2 I_{|C_u|x|C_u|})}{\sum_{j=1}^{K}\pi_j N(x_{C_u}^{(u)}|\mu_{C_u}^{(i)},\sigma_i^2 I_{|C_u|x|C_u|})}\delta(l,C_u)\frac{x_l^{(u)}-\mu_l^{(k)}}{\sigma_{l_{(k)}}^2}$$

$$0 = \sum_{u=1}^{n}p(k|u)\delta(l,C_u)\frac{x_l^{(u)}-\mu_l^{(k)}}{\sigma_{l_{(k)}}^2}$$

$$\hat{\mu_l}^{(k)} = \frac{\sum_{u=1}^{n}\delta(l,C_n)p(k|u)x_l^{(u)}}{\sum_{u=1}^{n}\delta(l,C_n)p(k|u)}$$

$$\hat{\mu_l}^{(j)} = \frac{\sum_{u=1}^{n}\delta(l,C_n)p(j|u)x_l^{(u)}}{\sum_{u=1}^{n}\delta(l,C_n)p(j|u)}$$

We do not compute the mean update in the log domain; we use $p(j|u)$ instead of $l(j,u)$. When you set $\mu_i^{(j)}$ and σ_j^2 in the implementation, it will be easier, and not lead to numerical underflow issues, to use $p(j|u)$ instead of the logarithm $l(j,u)$.

Finally, the update equation for the variance is not too different from before:

$$\hat{\sigma}_j^2 = \frac{1}{\sum_{u=1}^{n}|C_n|p(j|u)}\sum_{u=1}^{n}p(j|u)||x_{C_n}^{(u)}-\hat{\mu}_{Cn}^{(j)}||^2$$

Implementation guidelines: You may find LogSumExp useful. But remember that your M-step should return the new $P = \hat{\pi}$, not the log of $\hat{\pi}$.

- The following will not affect the update equation above, but will affect your implementation: since we are dealing with incomplete data, we might have a case where most of the points in cluster j are missing the i -th coordinate. If we are not careful, the value of this coordinate in the mean will be determined by a small number of points, which leads to erratic results. Instead, we should only update the mean when

$$\sum_{u=1}^{n}p(j|u)\delta(i,C_u) \geq 1$$

. Since $p(j|u)$ is a soft probability assignment, this corresponds to the case when at least one full point supports the mean.

- To also avoid the variances of clusters going to zero due to a small number of points being assigned to them, in the M-step you will need to implement a minimum variance for your clusters. We recommend a value of 0.25, though you are free to experiment with it if you wish. Note that this issue, as well as the thresholded mean update in the point above, are better dealt with through regularization; however, to keep things simple, we do not do regularization here.
- To debug your EM implementation, you may use the data files *test_incomplete.txt* and *test_complete.txt*. Compare your results to ours from test_solutions.txt.

0.8.1 Implementing E-step (2)

In em.py, fill in the estep function so that it works with partially observed vectors where missing values are indicated with zeros, and perform the computations in the log domain to help with numerical stability.

Available Functions: You have access to the NumPy python library as np, to the GaussianMixture class and to typing annotation typing.Tuple as Tuple. You also have access to scipy.special.logsumexp as logsumexp

Hint: For this function, you will want to use $log(mixture.p[j] + 1e^{-16})$ instead of $log(mixture.p[j])$ to avoid numerical underflow

[21]:
```
def estep(X: np.ndarray, mixture: GaussianMixture) -> Tuple[np.ndarray, float]:
    """E-step: Softly assigns each datapoint to a gaussian component

    Args:
        X: (n, d) array holding the data, with incomplete entries (set to 0)
        mixture: the current gaussian mixture

    Returns:
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the assignment

    """

    n, d = X.shape
    K, _ = mixture.mu.shape
    log_post = np.zeros((n, K))
    post = np.zeros((n, K))
    log_likelihood = np.zeros((n, 1))

    def log_normal_pdf(x: np.ndarray, mu: np.ndarray, var: float):
        d = len(x)
        x_mu = np.matrix(x-mu)
        norm_const = np.log((2*np.pi*var)) * (-d/2)
        norm_exp = -1.0 / (2*var) * np.linalg.norm(x_mu)**2
        return norm_const + norm_exp
    # calculate post: (n, k) array holding the soft counts
    for i in range(n):
        tiled_vector = np.tile(X[i, :], (K, 1))

        ind = np.where(tiled_vector.any(axis=0))[0]
        tiled_vector_Cu = tiled_vector[:,ind]
        mu_Cu = mixture.mu[:,ind]

        for j in range(K):
            fij = np.log(mixture.p[j]+1e-16) +␣
↪log_normal_pdf(tiled_vector_Cu[j,:], mu_Cu[j], mixture.var[j])
            log_post[i, j] = fij

        log_likelihood[i] = logsumexp(log_post[i,:])

        log_post[i,:] = log_post[i,:]-logsumexp(log_post[i,:])
        post[i,:] = np.exp(log_post[i,:])
        post[i,:] = post[i,:]/sum(post[i,:])

    return post, sum(log_likelihood)
```

0.8.2 Implementing M-step (2)

In em.py, fill in the mstep function so that it works with partially observed vectors where missing values are indicated with zeros, and perform the computations in the log domain to help with numerical stability.

Available Functions: You have access to the *NumPy* python library as np, to the *GaussianMixture* class and to typing annotation typing.Tuple as Tuple. You also have access to *scipy.misc.logsumexp* as *logsumexp*

[22]:
```
def mstep(X: np.ndarray, post: np.ndarray, mixture: GaussianMixture,
          min_variance: float = .25) -> GaussianMixture:
    """M-step: Updates the gaussian mixture by maximizing the log-likelihood
    of the weighted dataset

    Args:
        X: (n, d) array holding the data, with incomplete entries (set to 0)
        post: (n, K) array holding the soft counts
            for all components for all examples
        mixture: the current gaussian mixture
        min_variance: the minimum variance for each gaussian

    Returns:
        GaussianMixture: the new gaussian mixture
    """

    n, d = X.shape
    _, K = post.shape

    # 1. update hat_pi
    n_hat = post.sum(axis=0)
    p = n_hat / n

    #2. update mu and sigma (incomplete observation: some coordinates of mu do␣
↪not impact observation - missing observations)
    mu = np.copy(mixture.mu)
    var = np.zeros(K)

    ind = X>0
    ind.astype(np.int)

    for j in range(K):
        mu_j = np.copy(mu[j, :])
        #mu[j, :] = post[:, j] @ X / (post[:, j] @ ind)
        a = mu[j, :] = post[:, j] @ X
        b = (post[:, j] @ ind)
        mu[j, :] = np.divide(a, b, out=np.zeros_like(a), where=b!=0)

        pj = post[:, j] @ ind
        ind_pj = pj < 1
        for k, KI in enumerate(ind_pj):
            if KI:
                mu[j, k] = mu_j[k]

        tiled_vector = np.multiply(np.tile(mu[j], (n, 1)), ind)
        sse = ((tiled_vector - X)**2).sum(axis=1) @ post[:, j]
        c = sum( ind.sum(axis=1) * post[:, j] )
        var[j] = np.divide(sse, c, out=np.zeros_like(sse), where=c!=0)

    var[var < min_variance] = min_variance

    return GaussianMixture(mu, var, p)
```

test.py

[23]:
```
import numpy as np
import em
import common

X = np.loadtxt("test_incomplete.txt")
X_gold = np.loadtxt("test_complete.txt")

K = 4
```

```
n, d = X.shape
seed = 0

m, post = common.init(X, K, seed)
print('X = {0}\n K = {1}\n mu = {2}\n Var = {3}\n p = {4}\n'.format(X, K, m.mu,␣
↪m.var, m.p))

# first E-step
post, prev_log_likelihood = em.estep(X, m)
print('post = {0}\n LL = {1}\n'.format(post, prev_log_likelihood))

# first M-step
m = em.mstep(X, post, m)
print('mu = {0}\n Var = {1}\n p = {2}\n'.format(m.mu, m.var, m.p))


post, log_likelihood = em.estep(X, m)
print('post = {0}\n LL = {1}\n'.format(post, log_likelihood))

while (log_likelihood - prev_log_likelihood >= 1e-6 * abs(log_likelihood)):
    prev_log_likelihood = log_likelihood
    m = em.mstep(X, post, m)
    post, log_likelihood = em.estep(X, m)

print('mu = {0}\n Var = {1}\n p = {2}\n post = {3}\n LL = {4}\n'.format(m.mu, m.
↪var, m.p, post, log_likelihood))
```

```
X = [[2. 5. 3. 0. 0.]
 [3. 5. 0. 4. 3.]
 [2. 0. 3. 3. 1.]
 [4. 0. 4. 5. 2.]
 [3. 4. 0. 0. 4.]
 [1. 0. 4. 5. 5.]
 [2. 5. 0. 0. 1.]
 [3. 0. 5. 4. 3.]
 [0. 5. 3. 3. 3.]
 [2. 0. 0. 3. 3.]
 [3. 4. 3. 3. 3.]
 [1. 5. 3. 0. 1.]
 [4. 5. 3. 4. 3.]
 [1. 4. 0. 5. 2.]
 [1. 5. 3. 3. 5.]
 [3. 5. 3. 4. 3.]
 [3. 0. 0. 4. 2.]
 [3. 5. 3. 5. 1.]
 [2. 4. 5. 5. 0.]
 [2. 5. 4. 4. 2.]]
 K = 4
 mu = [[2. 4. 5. 5. 0.]
 [3. 5. 0. 4. 3.]
 [2. 5. 4. 4. 2.]
 [0. 5. 3. 3. 3.]]
 Var = [5.93 4.87 3.99 4.51]
 p = [0.25 0.25 0.25 0.25]

post = [[0.17713577 0.12995693 0.43161668 0.26129062]
 [0.08790299 0.35848927 0.41566414 0.13794359]
 [0.15529703 0.10542632 0.5030648  0.23621184]
 [0.23290326 0.10485918 0.58720619 0.07503136]
 [0.09060401 0.41569201 0.32452345 0.16918054]
 [0.07639077 0.08473656 0.41423836 0.42463432]
 [0.21838413 0.20787523 0.41319756 0.16054307]
 [0.16534478 0.04759109 0.63399833 0.1530658 ]
 [0.05486073 0.13290982 0.37956674 0.43266271]
 [0.08779356 0.28748372 0.37049225 0.25423047]
 [0.07715067 0.18612696 0.50647898 0.23024339]
 [0.16678427 0.07789806 0.45643509 0.29888258]
 [0.08544132 0.24851049 0.53837544 0.12767275]
 [0.17773171 0.19578852 0.41091504 0.21556473]
 [0.02553529 0.1258932  0.29235844 0.55621307]
 [0.07604748 0.19032469 0.54189543 0.1917324 ]
 [0.15623582 0.31418901 0.41418177 0.1153934 ]
 [0.19275595 0.13517877 0.56734832 0.10471696]
 [0.33228594 0.02780214 0.50397264 0.13593928]
 [0.12546781 0.05835499 0.60962919 0.20654801]]
 LL = [-152.16319226]

mu = [[2.38279095 4.64102716 3.73583539 4.28989488 2.17237898]
 [2.56629755 4.6686168  3.24084599 3.88882023 2.72874336]
 [2.45674721 4.72686227 3.55798344 4.05614484 2.5030405 ]
 [2.00305536 4.7674522  3.37388115 3.7905181  2.97986269]]
 Var = [0.71489705 0.64830186 0.73650336 0.85722393]
 p = [0.13810266 0.17175435 0.46575794 0.22438505]

post = [[0.11737048 0.20532903 0.44764518 0.22965531]
 [0.0969455  0.26670226 0.49465442 0.14169782]
 [0.16990327 0.14100071 0.51987021 0.16922582]
 [0.24383375 0.11791174 0.58874008 0.04951442]
 [0.05671393 0.25890771 0.4110296  0.27334875]
 [0.02458873 0.03962106 0.19606273 0.73972748]
 [0.2830109  0.09249982 0.51514382 0.10934546]
 [0.15806864 0.12158202 0.57784459 0.14250475]
 [0.04892659 0.27590349 0.39192455 0.28324536]
 [0.06415332 0.21227372 0.40787567 0.31569729]
 [0.05472879 0.34362655 0.41305216 0.1885925 ]
 [0.23729247 0.07548678 0.47177552 0.21544523]
 [0.07176612 0.32726516 0.4936478  0.10732092]
 [0.25718611 0.07848343 0.45859946 0.205731  ]
 [0.0048859  0.05467342 0.09916626 0.84127443]
 [0.07828671 0.31583151 0.46522763 0.14065415]
 [0.17127824 0.19831144 0.53700295 0.09340737]
 [0.29963468 0.10805485 0.54033075 0.05197972]
 [0.30229591 0.05763939 0.49821186 0.14185284]
 [0.20436195 0.13533675 0.53552727 0.12477404]]
 LL = [-103.43699945]

mu = [[2.00580495 4.99057824 3.13828416 4.00143976 1.16257667]
 [2.99393075 4.68354673 3.0052711  3.52424565 3.08961136]
 [2.54570695 4.20101006 4.56569215 4.55566605 2.31175294]
 [1.01534142 4.99975328 3.49252182 3.99998114 4.99986014]]
 Var = [0.25       0.25       0.44965163 0.2792997 ]
 p = [0.27682092 0.35433942 0.26728961 0.10155005]
 post = [[8.35114583e-01 1.26066023e-01 8.03346942e-03 3.07859243e-02]
 [2.29595284e-04 9.30406661e-01 6.93634633e-02 2.80840424e-07]
 [9.98723643e-01 1.34234094e-04 1.14212255e-03 1.65905887e-14]
 [1.85331147e-04 1.94115053e-03 9.97873518e-01 2.57285049e-14]
 [1.82091725e-08 8.82200084e-01 1.17730763e-01 6.91351811e-05]
 [2.13395201e-14 1.74763538e-08 1.23289877e-04 9.99876693e-01]
 [9.78452231e-01 2.41596929e-05 2.15236097e-02 2.05795060e-14]
 [1.95291523e-06 3.46537075e-03 9.96532634e-01 4.18625878e-08]
 [2.53995753e-04 9.99058306e-01 6.46220953e-04 4.14767958e-05]
 [1.39755279e-03 8.96199140e-01 1.02340131e-01 6.31761952e-05]
 [1.02964283e-05 9.98438589e-01 1.55110233e-03 1.18280899e-08]
 [9.99175360e-01 4.92298629e-07 8.24147990e-04 5.73816393e-13]
 [4.54696111e-06 9.96705586e-01 3.28986689e-03 1.91139775e-10]
 [4.13182467e-02 1.40457914e-05 9.58667653e-01 5.48560980e-08]
 [9.22358785e-14 4.78927600e-06 3.67220413e-07 9.99994844e-01]
 [2.36604822e-04 9.96136619e-01 3.62659186e-03 1.84275504e-07]
 [1.09042309e-01 2.42442342e-01 6.48515348e-01 8.68166867e-11]
 [9.62134995e-01 1.21159085e-04 3.77438456e-02 5.30337126e-16]
 [1.39885506e-04 2.34579872e-06 9.99672523e-01 1.85246074e-04]
 [6.05773445e-01 1.29236657e-02 3.81302856e-01 3.38895349e-08]]
 LL = [-84.98451993]
```

0.8.3 Implementing run

In em.py, fill in the run function so that it runs the EM algorithm. As before, the convergence criteria that you should use is that the improvement in the log-likelihood is less than or equal to 10^6 multiplied by the absolute value of the new log-likelihood.

Available Functions: You have access to the *NumPy* python library as np, to the *GaussianMixture* class and to typing annotation *typing.Tuple* as *Tuple*. You also have access to the estep and mstep functions you have just implemented

Correction note (Aug 8): Since the mstep function in previous problem has been defined differently since Aug 8, you will need to modify run function accordingly. Note that the grader will accept as correct a run function that works with either the earlier or current version of mstep.

[24]:
```python
def run(X: np.ndarray, mixture: GaussianMixture,
        post: np.ndarray) -> Tuple[GaussianMixture, np.ndarray, float]:
    """Runs the mixture model

    Args:
        X: (n, d) array holding the data
        post: (n, K) array holding the soft counts
            for all components for all examples

    Returns:
        GaussianMixture: the new gaussian mixture
        np.ndarray: (n, K) array holding the soft counts
            for all components for all examples
        float: log-likelihood of the current assignment
    """
    n, d = X.shape
    _, K = post.shape

    post, prev_cost = estep(X, mixture)
    mixture = mstep(X, post, mixture)
    post, cost = estep(X, mixture)

    while (cost - prev_cost >= 1e-6 * abs(cost)):
        prev_cost = cost
        mixture = mstep(X, post, mixture)
        post, cost = estep(X, mixture)

    return mixture, post, cost
```

[25]:
```python
#test run
X = np.loadtxt("test_incomplete.txt")
X_gold = np.loadtxt("test_complete.txt")

K = 4
n, d = X.shape
seed = 0

def em_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = common.init(X, k, s)
            _, _, c = em.run(X, m, p)
            seed.setdefault(s, []).append(c)
        cost.setdefault(k, max(seed.values()))
    return m, cost

print(em_cost(X, [K], [seed]))
```

```
(GaussianMixture(mu=array([[2., 4., 5., 5., 0.],
       [3., 5., 0., 4., 3.],
       [2., 5., 4., 4., 2.],
       [0., 5., 3., 3., 3.]]), var=array([5.93, 4.87, 3.99, 4.51]),
p=array([0.25, 0.25, 0.25, 0.25])), {4: [array([-84.98451993])]})
```

0.9 8. Using the mixture model for collaborative filtering

0.9.1 Reporting log likelihood values on Netflix data

Now, run the EM algorithm on the incomplete data matrix from Netflix ratings netflix_incomplete.txt. As before, please use seeds from [0,1,2,3,4] and report the best log likelihood you achieve with K=1 and K=12.

This may take on the order of a couple minutes for K=12.

Report the maximum likelihood for each K using seeds 0,1,2,3,4:

Log-likelihood | K=1= -1521060.95398525

Log-likelihood | K=12= -1390234.42234694

[26]:
```
import numpy as np
import em
import common

X = np.loadtxt("netflix_incomplete.txt")
X_gold = np.loadtxt("netflix_complete.txt")

def em_cost(X, K, S) -> dict:
    for k in K:
        for s in S:
            m, p = common.init(X, k, s)
            m, p, log_likelihood = em.run(X, m, p)
            BIC = common.bic(X, m, log_likelihood)
            print("k={0}, S={1}, log_likelihood={2}, BIC value={3}".format(k,␣
 ↪s, log_likelihood, BIC))
    return
#

K = [1, 12]
S = [0,1,2,3,4]

K = [1, 12]
for k in K:
    em_cost(X, [k], S)
```

```
k=1, S=0, log_likelihood=[-1521060.95398525], BIC value=[-1525318.54512513]
k=1, S=1, log_likelihood=[-1521060.95398525], BIC value=[-1525318.54512513]
k=1, S=2, log_likelihood=[-1521060.95398525], BIC value=[-1525318.54512513]
k=1, S=3, log_likelihood=[-1521060.95398525], BIC value=[-1525318.54512513]
k=1, S=4, log_likelihood=[-1521060.95398525], BIC value=[-1525318.54512513]
k=12, S=0, log_likelihood=[-1399803.04665691], BIC value=[-1450933.13575811]
k=12, S=1, log_likelihood=[-1390234.42234694], BIC value=[-1441364.51144814]
k=12, S=2, log_likelihood=[-1416862.40115128], BIC value=[-1467992.49025248]
k=12, S=3, log_likelihood=[-1393521.39298978], BIC value=[-1444651.48209098]
k=12, S=4, log_likelihood=[-1416733.80837635], BIC value=[-1467863.89747755]
```

0.9.2 Completing missing entries

Now that we have a mixture model, how do we use it to complete a partially observed rating matrix? Derive an expression for completing a particular row, say x_C where the observed values are iC.

In *em.py* implement the function $fill_matrix.py$ that takes as input an incomplete data matrix X as well as a mixture model, and outputs a completed version of the matrix X_pred.

Available Functions: You have access to the *NumPy* python library as np, to the *GaussianMixture* class and to typing annotation *typing.Tuple* as *Tuple*. You also have access to *scipy.special.logsumexp* as *logsumexp*

[27]:
```python
def fill_matrix(X: np.ndarray, mixture: GaussianMixture) -> np.ndarray:
    """Fills an incomplete matrix according to a mixture model

    Args:
        X: (n, d) array of incomplete data (incomplete entries =0)
        mixture: a mixture of gaussians

    Returns
        np.ndarray: a (n, d) array with completed data
    """

    n, d = X.shape
    K, _ = mixture.mu.shape

    pred = np.copy(X)
    log_post = np.zeros((n, K))
    post = np.zeros((n, K))

    def normal_pdf(x: np.ndarray, mu: np.ndarray, var: float):
        d = len(x)
        x_mu = np.matrix(x-mu)
        norm_const = np.log((2*np.pi*var)) * (-d/2)
        norm_exp = -1.0 / (2*var) * np.linalg.norm(x_mu)**2
        return norm_const + norm_exp

    # calculate post: (n, k) array holding the soft counts
    for i in range(n):
        tiled_vector = np.tile(X[i, :], (K, 1))

        ind = np.where(tiled_vector.any(axis=0))[0]
        tiled_vector_Cu = tiled_vector[:,ind]
        mu_Cu = mixture.mu[:,ind]

        for j in range(K):
            fij = np.log(mixture.p[j]+1e-16) + normal_pdf(tiled_vector_Cu[j,:],␣
 ↪mu_Cu[j], mixture.var[j])
            log_post[i, j] = fij

        log_post[i,:] = log_post[i,:]-logsumexp(log_post[i,:])
        post[i,:] = np.exp(log_post[i,:])
        post[i,:] = post[i,:]/sum(post[i,:])

    mask = pred == 0
    np.putmask(pred, mask, post @ mixture.mu)

    return pred
```

[28]:
```
import numpy as np
import em
import common

X = np.loadtxt("test_incomplete.txt")
X_gold = np.loadtxt("test_complete.txt")

K = 4
n, d = X.shape
seed = 0

def em_cost(X, K, S) -> dict:
    cost ={}
    for k in K:
        seed={}
        for s in S:
            m, p = common.init(X, k, s)
            _, _, c = em.run(X, m, p)
            seed.setdefault(s, []).append(c)
        cost.setdefault(k, max(seed.values()))
    return m, cost

m, _ = common.init(X, K, seed)
post, prev_log_likelihood = em.estep(X, m)
m = em.mstep(X, post, m)
post, log_likelihood = em.estep(X, m)
```

```python
while (log_likelihood - prev_log_likelihood >= 1e-6 * abs(log_likelihood)):
    prev_log_likelihood = log_likelihood
    m = em.mstep(X, post, m)
    post, log_likelihood = em.estep(X, m)

pred = em.fill_matrix(X, m)
print('X = {0}\n  Pred = {1}\n'.format(X, pred))

print(common.rmse(pred, X_gold))
```

```
X = [[2. 5. 3. 0. 0.]
 [3. 5. 0. 4. 3.]
 [2. 0. 3. 3. 1.]
 [4. 0. 4. 5. 2.]
 [3. 4. 0. 0. 4.]
 [1. 0. 4. 5. 5.]
 [2. 5. 0. 0. 1.]
 [3. 0. 5. 4. 3.]
 [0. 5. 3. 3. 3.]
 [2. 0. 0. 3. 3.]
 [3. 4. 3. 3. 3.]
 [1. 5. 3. 0. 1.]
 [4. 5. 3. 4. 3.]
 [1. 4. 0. 5. 2.]
 [1. 5. 3. 3. 5.]
 [3. 5. 3. 4. 3.]
 [3. 0. 0. 4. 2.]
 [3. 5. 3. 5. 1.]
 [2. 4. 5. 5. 0.]
 [2. 5. 4. 4. 2.]]
  Pred = [[2.         5.         3.         3.94568925 1.53287646]
 [3.         5.         3.11353798 4.         3.        ]
 [2.         4.98963525 3.         3.         1.        ]
 [4.         4.20209307 4.         5.         2.        ]
 [3.         4.         3.18901435 3.64570846 4.        ]
 [1.         4.9996548  4.         5.         5.        ]
 [2.         5.         3.16900392 4.01335718 1.        ]
 [3.         4.2026838  5.         4.         3.        ]
 [2.99330806 5.         3.         3.         3.        ]
 [2.         4.63461293 3.16518147 3.         3.        ]
 [3.         4.         3.         3.         3.        ]
 [1.         5.         3.         4.00189629 1.        ]
 [4.         5.         3.         4.         3.        ]
 [1.         4.         4.50669218 5.         2.        ]
 [1.         5.         3.         3.         5.        ]
 [3.         5.         3.         4.         3.        ]
 [3.         4.40409372 4.03173215 4.         2.        ]
 [3.         5.         3.         5.         1.        ]
 [2.         4.         5.         5.         2.31209197]
 [2.         5.         4.         4.         2.        ]]

0.31517185088634886
```

0.9.3 Comparing with gold targets

Test the accuracy of your predictions against actual target values by loading the complete matrix X_gold = np.loadtxt('netflix_complete.txt') and measuring the root mean squared error between the two matrices using common.rmse(X_gold, X_pred). Use your best mixture for K=12 from the first question of this tab to generate the results.

RMSE= 0.48047111268956827

Correction note (Aug 8): Since the mstep function has been defined differently since Aug 8, the root mean squared error will also be changed. But note that the grader will also accept RMSE resulting from from the earlier and current versions.

More Challenge: Collaborative Filtering Using Matrix Factorization and Neural Networks Now that you have solved use the Expectation Maximization (EM) algorithm to perform collaborative filtering, you may also want to try solving the same task using matrix factorization as discussed in earlier in the course. Here are some steps you could follow in that direction:

1. Implement and test matrix factorization directly. (You will actually find it easier to implement, and will likely work better.)
2. Reformulate the matrix factorization model as a neural network model where you feed in one-hot vectors of the user and movie and predict outcome for each pair.
3. Incorporate additional feature information available about the user/movie into the neural network model.

An Application of the EM algorithm: Black Hole Imaging You may wonder where else the EM algorithm is used.

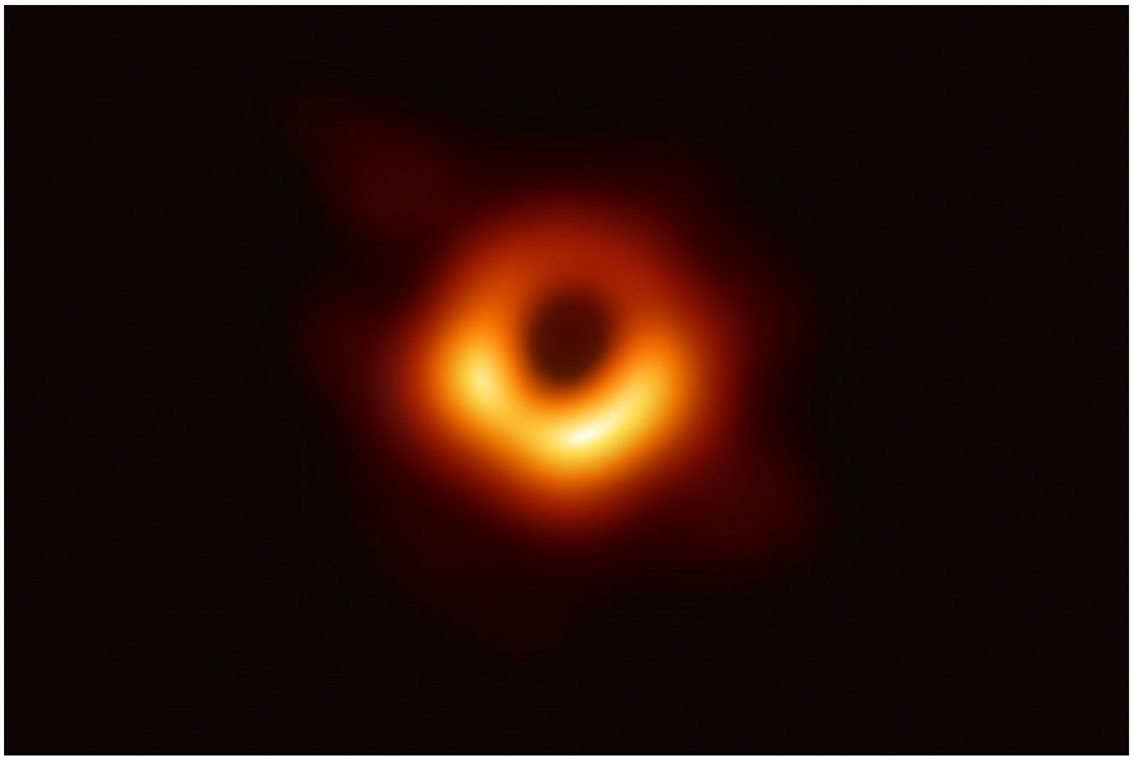
The following story will draw a connection between the EM algorithm and the first black hole "image" that was uncovered to the world on April 10, 2019. The material is mainly based on the work of Katherine L. Bouman during her PhD at MIT.

0.9.4 The sparse Reconstruction Problem in Black Hole Imaging

Say we want to obtain a picture of a black hole. Could we just take a "good" camera and directly take a "picture" of it? As you may have guessed, the task is not that easy.

[29]:
```
from IPython.display import Image
Image(filename='C:/root/workspace/MITx_6.86x/08_media/images_blackhole-image.
 ↪jpg')
```

[29]:



The main issue is that black holes are too far away from us human beings. In fact, the black hole in the picture above is roughly 55 million light years from earth: it is a supermassive black hole at the heart of Messier 87, or M87, a galaxy within the Virgo galaxy cluster (so even if you can travel in a spaceship that moves as fast as light, you will need 55 million years to get there from home). This long distance means that taking a picture of this blackhole is roughly as difficult of taking a picture of an orange on the Moon surface from earth.

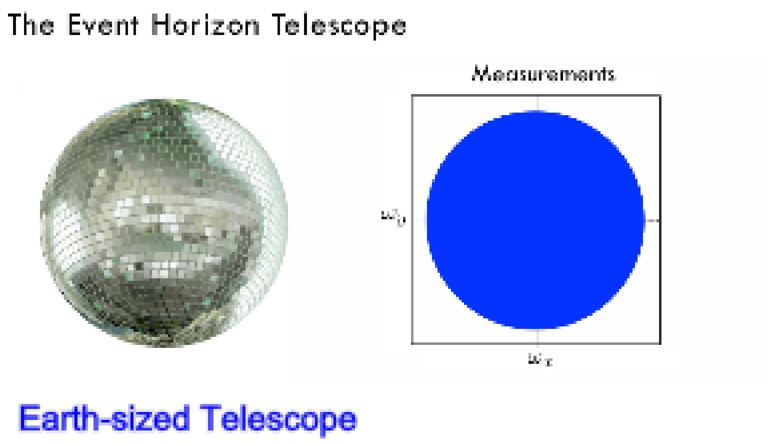
However, current photographing technology only allows us to take a picture of the Moon surface in which each pixel captures the size of millions of oranges. A high-resolution telescope that could directly take a picture of an object that "small" in terms of its distance to us would need be the size of the entire earth.

Let us assume for now we have turned our entire earth to a telescope - then could we really observe the black hole? The answer is yes. Even though the black hole, as its name suggests, absorbs everything including light, near its outer marginal surface there is something observable - this is the so-called "event horizon." And this is really what the telescopes are really observing, and that's also why those telescopes of observing the black hole is called the Event Horizon Telescope (EHT).

Back to reality, we do not have such large telescopes. Instead, what we can do is to use several much smaller telescopes, at different locations (Mexico, Hawaii, Chile, Spain, South Pole, and Arizona), to receive the measurements from the black hole, and then "piece together" these to reconstruct the full image. The technique behind this multiple-telescope combination for imaging is called Very Long Baseline Interferometry (VLBI).

[30]:
```
Image(filename='C:/root/workspace/MITx_6.86x/08_media/images_earthsize.png')
```

[30]:



[31]:
```
Image(filename='C:/root/workspace/MITx_6.86x/08_media/images_VLBI.png')
```

[31]:

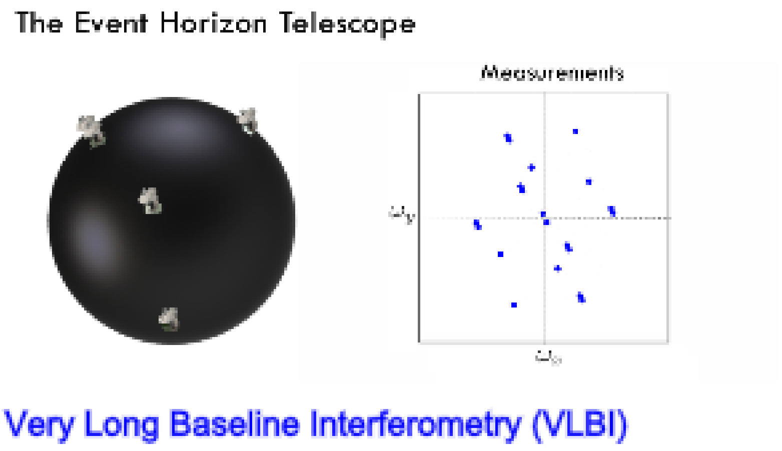

The Event Horizon Telescope
Measurements
ω_y
ω_x
Very Long Baseline Interferometry (VLBI)

Image credit:https://people.csail.mit.edu/klbouman/pw/papers_and_presentations/bouman_tedxslides.p

- (Up) Using an Earth-sized EHT, we would obtain an image with dense measurements
- (Down) Using VLBI, we obtain only sparse measurements

Using VLBI, we can only obtain sparse measurements from the black hole. (There are some ways to increase the amount of data we have. For example, since the Earth is rotating, the EHTs also move to different locations so as we are able to get more measurements.).

With the collected data, the problem becomes an algorithmic one: how do we reconstruct the "image" from these measurements, where by "image" we mean not the full and real one, but only one that is close to the true image with high probability.

The scale of this machine learning problem is enormous. The EHT teams took only 5 days to collect the roughly 3500TB of data. The researchers then took two years to clean the data, and train and adjust the machine learning algorithms in order to get the final image we now see.

Single Image VLBI Reconstruction of Static Sources For simplicity, we first consider the model of static sources, i.e., let us assume the image of the black hole would not change over time. Then, according to physics we know the image x and our measurements y has a functional dependence of f, that is:

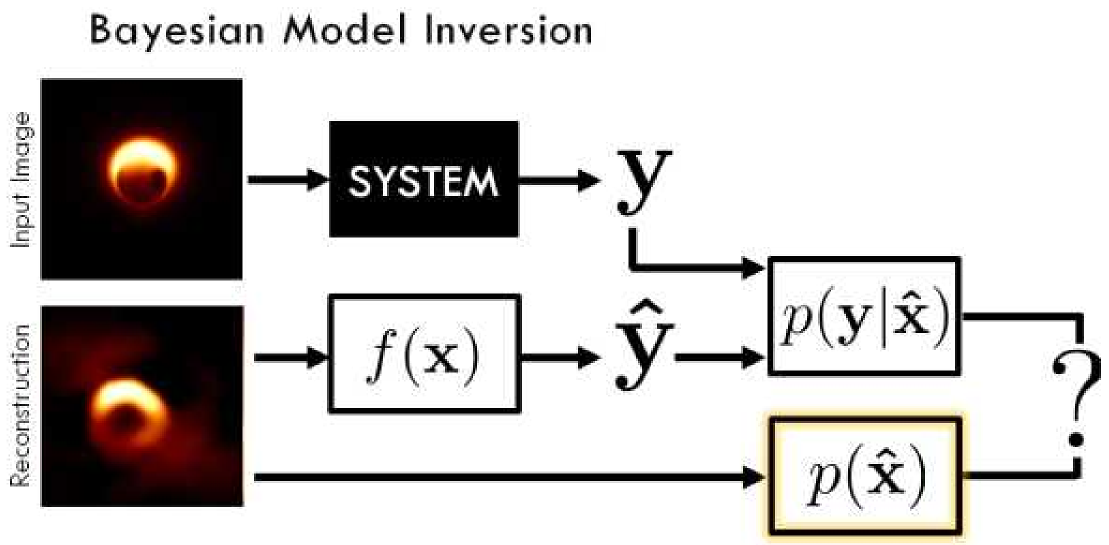
$$f(x) = y$$

If we have dense measurements, we could perform the inverse function operation $f^{-1}(y)$ to get the desired image. However, we only have sparse measurements, so the goal here is to get an approximation $\hat{x}$ of x from the measurements. Even for VLBI reconstruction of static sources, many of the traditional methods in image reconstruction does not work, because there are delays in the time when we get the measurements depending on the location of the EHTs. Since the EHTs are spread over the Earth, these delays can be viewed as random.

What about a Bayesian model? Following Katherine Bouman, we can start with the simplest model, the Gaussian Mixture Model (GMM), which we have seen in the lectures!

[32]:
```
Image(filename='C:/root/workspace/MITx_6.86x/08_media/images_Bayesian-inverse.
↪jpg')
```

[32]:



Specifically, we assume

$$y \sim N_y(f(x), R)$$

so that the measurements follows a Gaussian distribution of mean $f(x)$. We define each overlapping patch z of image x as being a sample from a GMM with C clusters. Here patch samples (patches of the image) include other planets, galaxies, simulated black hole images consistent with General Relativity, and even daily life pictures, and we denote them as the collection $\{z_n : n = 1 : \ldots, N\}$. We assume these samples are from a GMM model with C clusters, that is,

$$z_n = P_n x \sim \sum_{c=1}^{C} \pi_c N_p(\mu_c, \Lambda_c)$$

where matrix P_n extracts the n-th overlapping patch, z_n, from x, and π_c is the mixture component weight of the c-th cluster. From this relation we can write the probability of each patch z as

$$p(z) = \sum_{c=1}^{C} \frac{\pi_c}{\sqrt{|2\pi\Lambda}} exp(\frac{1}{2}(z - \mu_c)^T \Lambda^{-1} (z - \mu_c))$$

We use EM algorithm to train the GMM patch prior. Namely, given N patches $\{z_n : n = 1 : \ldots, N\}$, we iterate between the E-steps and M-steps as follows

- E-step: find the expected likelihood of a patch belonging to each cluster.

$$\xi_c(z_n) = \frac{\pi_c N_{z_n}(\mu_c, \Lambda_c)}{\sum_{k=1}^{C} \pi_k N_{z_n}(\mu_k, \Lambda_k)}$$

- M-step: find the model parameters that maximize the expected value of the log likelihood function.

$$\mu_x = \frac{\sum_{n=1}^{N} \xi_c(z_n) z_n}{\sum_{n=1}^{N} \xi_c(z_n)}$$

$$\Lambda_c = \frac{\sum_{n=1}^{N} \xi_c(z_n)(z_n - \mu_c)(z_n - \mu_c)^T}{\sum_{n=1}^{N} \xi_c(z_n)}$$

$$\pi_c = \frac{1}{N} \sum_{n=1}^{N} \xi_c(z_n)$$

Since we have defined the likelihood of an image patch, and not the likelihood of the image itself, we can no longer write a posterior distribution of x given y. However, we still are able to write an optimizing cost function in the form

$$\hat{x} = argmin_x[X(x, y) - \beta R(x)]$$

in which $X(x, y) = \frac{1}{2}(y - f(x))^T R^{-1} (y - f(x))$ denotes the deviation of y under conditional distribution $p(y|x)$ and $R(x) = \frac{1}{N} \sum_{n=1}^{N} log[p(P_n x)]$ denotes the regularization term. Here $\beta = \frac{1}{2}$.

Definitely, we then need to do some optimization to find the argmin. Again the usual EM steps won't follow directly here but a similar iterative procedure called "Half Quadratic Splitting" works for this. 1. Solve for $\{z^n\}$ given $\hat{x}$: we first find the most likely cluster by evaluating the log-likelihood of each cluster

$$\hat{c}^n = argmax_c[log\pi_c - \frac{1}{2} log|2\pi\Lambda_c| - \frac{1}{2}(P_n\hat{x} - \mu_c)^T \Lambda_c^{-1} (P_n\hat{x} - \mu_c)]$$

We introduce a weighting parameter γ^1, that indicates the variance of noise in the current estimate of $\hat{x}$. Under the posterior distribution

$$p(z^n|P_n\hat{x}; \mu_{\hat{c}^n}, \Lambda_{\hat{c}^n} \Rightarrow N_{x_n}(P_n\hat{x}, \gamma^{-1}1)N_{x^n}(\mu_{\hat{c}^n}, \Lambda_{\hat{c}^n})$$

we estimate the best patch using Wiener filtering

$$z^n = \mu_{\hat{c}^n} + \Lambda_{\hat{c}^n}(\gamma^{-1}1 + \Lambda_{\hat{c}^n})(P_n\hat{x} - \mu_{\hat{c}^n})$$

2. Solve for $\hat{x}$ given $\{z_n\}$: We re-define the regularization term as

$$R(x) = \frac{1}{N}\sum_{n=1}^{N}\sum_{n=1}^{N}[\frac{\gamma}{2}(P_nx - z^n)^T(Pnx - z^n) - logp(z^n)]$$

and then given $\{z_n\}$ and if we restrict to linear measurements such that $f(x) = Fx$ we would have a closed form of $\hat{x}$ as (you should be able to work this out yourself by doing the MAP minimization)

$$\hat{x} = (F^TR^{-1}F + \frac{\beta\gamma}{N}\sum_{n=1}^{N} P_n^TP_n)^{-1}\ (F^TR^{-1}F + \frac{\beta\gamma}{N}\sum_{n=1}^{N} P_n^TP_n))$$

3. Iterate between step (1) and step (2) with increasing γ values of 1, 4, 8, 16, 32, 64, 128, 256, and 512. Note when $\gamma \to \infty$ patches P_nx are restricted to be equal to their auxiliary patch z_n.

That's it! This is the methodology that lets you reconstruct nice-resolution astronomical images from VLBI measurements.

Endnotes Of course, the methodology presented in the former section is not what the scientists applied exactly for getting the black hole image. There are a lot of issues we have not considered, for example, the image of black hole might change over time due to astronomical movements. In Section 4 of Katherine L. Bouman's PhD thesis, she addresses the issue by proposing more advanced model and algorithms based on Gaussian Hidden Markov Models (GHMM). The Markov model here is used for capturing the dynamics of the object.

The main takeaway of the story is that mixture models and EM algorithms are very powerful tools for machine learning tasks in science. Besides the exciting photographing of black holes, it is also widely used in many other applications, including financial modeling, choice model in revenue management, predictive maintenance, and so on. Compared to deep learning, the mixture models often give your better interpretability especially when you have a good understanding of the underlying model dynamics.