Statistical Machine Learning: Exercise 2

Parametric and Non-parametric Density Estimation, Expectation Maximization

Group 82: Alper Gece, Jinyao Chen

TECHNISCHE UNIVERSITÄT DARMSTADT

Summer Term 2021

Task 1: Optimization

1a) Numerical Optimization

Derivative of the function:

$$\begin{aligned}\frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200(x_i - x_{i-1}^2)(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}) - 2(1 - x_{i-1})\delta_{i-1,j} \\ &= 200(x_j - x_{j-1}^2) - 400x_j(x_{j+1} - x_j^2) - 2(1 - x_j)\end{aligned}$$

but the first and the last x are exception:

$$\begin{aligned}\frac{\partial f}{\partial x_0} &= -400x_0(x_1 - x_0^2) - 2(1 - x_0) \\ \frac{\partial f}{\partial x_{N-1}} &= 200(x_{N-1} - x_{N-2}^2)\end{aligned}$$

This is complete code:

```
import numpy as np
import matplotlib.pyplot as plt

# Derivative of the function
def derrivation_rosenbrock(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der


def main():
    # set the learning rate first
    # this learning set is ad hoc corresponds to objective function
    lrate = 0.002
    # initialize a point
    x = np.array([-0.5, 0.2,-0.3, 0.1,-0.4, 0.5,-0.1, 0.1,-0.3, 0.4,-0.1, 0.2,-0.5, 0.4,-0.2,
        0.4,-0.3, 0.2,-0.3, 0.4])
    # set number of steps (can be changed as you want)
    steps = 10000
    # we have to record all tuples of the points and its function
    ai = []
```

```
    n = 20
    for i in range(steps):
        sum = 0
        for j in range(n-1):
            f = (100 * ((x[j+1] - x[j] ** 2) ** 2) + (x[j] - 1) ** 2)
            sum += f
        # sum = rosen(x)
        # append the point and its obj function to ai as 1D list
        ai.append([x, sum])
        # compute its derrivative on point a
        fi = np.array(derrivation_rosenbrock(x))
        # set the new point based on its
        x = x - np.dot(lrate, fi)

    # convert ai into a numpy array
    ai = np.array(ai)
    # print the last 10 of ai
    print(ai[-10:-1])
    # the minimum value of the function is just the last element of ai
    print(f'the minimum is: {ai[-1, 1]} at point: {ai[-1,0]}')


    # Plot argmin parameter
    x = np.arange(0, 10000, 1)
    plt.plot(x, ai[:,1])
    plt.xlabel("x axis label")
    plt.ylabel("y axis label")
    plt.title("Argmin")
    plt.legend(["Parameter"])
    plt.show()

if __name__ == '__main__':
    main()
```

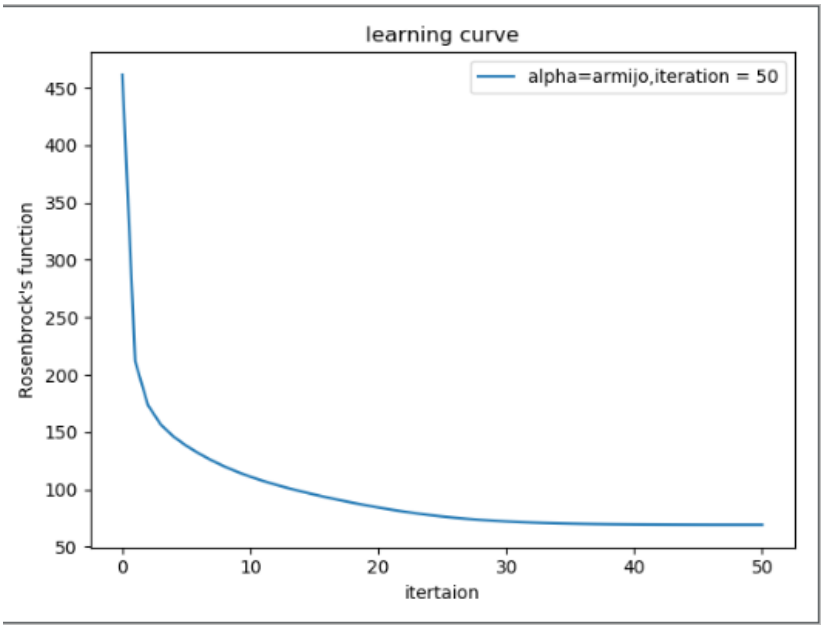


Figure 1: Learning Curve

1b) Gradient Descent Variants

(1)
vanilla:
pros:
For convex optimization problems, the algorithm will converge to global optima. In the case of non-convex questions, the final result will converge to local optimum. cons:
It can converge at local minima and saddle points.

stochastic:
pros:
The benefit of SGD is that it greatly reduces the time complexity. However, due to the random selection of a sample each time, it undoubtedly increases the randomness in the iteration process. Therefore, we can obviously observe the fluctuation in the graph of the function value on the number of iterations. cons:
Can veer off in the wrong direction due to frequent updates.

mini-batch:
pros:
On the one hand, we have greatly reduced the number of iterations and increased the efficiency; on the other hand, we have maintained some good properties of SGD, that is, we have reduced the Time Complexity when calculating the gradient and achieved better convergence performance.
cons:
But we have to configure the Mini-Batch size hyperparameter.

(2)
The momentum is an extension of the traditional gradient descent method (SGD), which is more efficient than SGD. Momentum method, also known as SGD with Momentum, is a method to accelerate gradient vectors to the relevant direction and finally achieve accelerated convergence. Momentum method is a very popular optimization algorithm and is used in many models today.

Using the stochastic gradient descent method, we will not calculate the exact derivative of the loss function. Instead, we estimate from a small set of data. This means that we are not always heading in the best direction because the results we get are "noisy". Therefore, a weighted average of the exponents can provide a better estimate that is closer to the derivative of the actual value than would be obtained by noisy computation. This is one reason why the momentum method may be better than traditional SGD.

Task 2: Density Estimation - MLE

2a) Maximization Likelihood Estimate of the Exponential Distribution

The probability density function of the exponential distribution is defined as:

$$f(n) = \begin{cases} \frac{1}{s} exp(-\frac{x}{s}) & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Its likelihood function is:

$$L(\lambda, x_1, x_2, \cdots, x_n) = \prod_{i=1}^{n} p(x|s) = \prod_{i=1}^{n} \frac{1}{s} exp(-\frac{x}{s}) = \frac{1}{s^n} exp(-\frac{1}{s} \sum_{i=1}^{n} x_i)$$

To calculate the MLE we need to solve the equation

$$\begin{gathered} \frac{dln(L(\lambda, x_1, x_2, \cdots, x_n))}{d\lambda} = 0 \\ \frac{dln(L(\lambda, x_1, x_2, \cdots, x_n))}{d\lambda} = \frac{dln(\lambda^n e^{-\lambda \sum_{i=1}^{n} x_i})}{d\lambda} \\ = \frac{d(nln(\lambda) - \lambda \sum_{i=1}^{n} x_i)}{d\lambda} \\ = \frac{n}{\lambda} - \sum_{i=1}^{n} x_i \end{gathered}$$

because

$$\begin{gathered} \lambda = \frac{1}{s} \Rightarrow \\ \frac{n}{\lambda} - \sum_{i=1}^{n} x_i = sn - \sum_{i=1}^{n} x_i \end{gathered}$$

Task 3: Density Estimation

3a) Prior Probabilities

$p(A) = \frac{count(A)}{count(A)+count(B)}$

The prior probability of each class from the dataset is computed as below python code.

```
import numpy as np

def priorOfTwo(data1, data2):
   length1 = 0
   length2 = 0
   for i in range(data1.shape[0]):
      length1 += 1
   for j in range(data2.shape[0]):
      length2 += 1
   return length1/(length1+length2), length2/(length1+length2)

densEst1 = np.loadtxt("./dataSets/densEst1.txt")
densEst2 = np.loadtxt("./dataSets/densEst2.txt")

print(prior1)
print(prior2)
```

The output of the code is given below.

$$P(C_1) = 0.24$$
$$P(C_2) = 0.76$$

3b) Biased ML Estimate

Define the bias of an estimator:

$$Bias_\theta[\widehat{\theta}] = E_\theta[\widehat{\theta}] - \theta = E_\theta[\widehat{\theta} - \theta]$$

Biased estimates of the conditional distribution:

$$s^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})}{N}$$

Unbiased estimates of the conditional distribution:

$$s^2 = \frac{\sum_{i=1}^{N}(x_i - \bar{x})}{N-1}$$

We need to calculate μ

```
import numpy as np

def meanOfTwo(data1, data2):
   mean11 = np.sum(data1[:, 0]) / data1.shape[0]
   mean12 = np.sum(data1[:, 1]) / data1.shape[0]

   mean21 = np.sum(data2[:, 0]) / data2.shape[0]
   mean22 = np.sum(data2[:, 1]) / data2.shape[0]

   return [mean11, mean12], [mean21, mean22]


def covarianceOfTwo(data1, data2):
   mean1, mean2 = meanOfTwo(densEst1, densEst2)
   bcov1 = (np.transpose(data1 - mean1) @ (data1 - mean1)) / data1.shape[0]
   bcov2 = (np.transpose(data2 - mean2) @ (data2 - mean2)) / data2.shape[0]
   ucov1 = (np.transpose(data1 - mean1) @ (data1 - mean1)) / (data1.shape[0] - 1)
   ucov2 = (np.transpose(data2 - mean2) @ (data2 - mean2)) / (data2.shape[0] - 1)
   return bcov1, bcov2, ucov1, ucov2


densEst1 = np.loadtxt("./dataSets/densEst1.txt")
densEst2 = np.loadtxt("./dataSets/densEst2.txt")

mean1, mean2 = meanOfTwo(densEst1, densEst2)
biasedcovariance1, biasedcovariance2, \
unbiasedcovarience1, unbiasedcovarience2 = covarianceOfTwo(densEst1, densEst2)

print(mean1)
print(mean2)
print(biasedcovariance1)
print(biasedcovariance2)
print(unbiasedcovarience1)
print(unbiasedcovarience2)
```

The output of the code is given below.

$$\text{mean1} = \begin{bmatrix} -0.7053709973916666 & -0.8135076184145833 \end{bmatrix}$$
$$\text{mean2} = \begin{bmatrix} 3.9879521086697367 & 3.9871418789315785 \end{bmatrix}$$
$$\text{biasedcovariance1} = \begin{bmatrix} 8.98244198 & 2.6617074 \\ 2.66170741 & 3.58135631 \end{bmatrix}$$
$$\text{biasedcovariance2} = \begin{bmatrix} 4.17569303 & 0.02214194 \\ 0.02214194 & 2.75079593 \end{bmatrix}$$
$$\text{unbiasedcovarience1} = \begin{bmatrix} 9.02002542 & 2.67284426 \\ 2.67284426 & 3.59634107 \end{bmatrix}$$
$$\text{unbiasedcovarience2} = \begin{bmatrix} 4.1811946 & 0.02217111 \\ 0.02217111 & 2.75442017 \end{bmatrix}$$

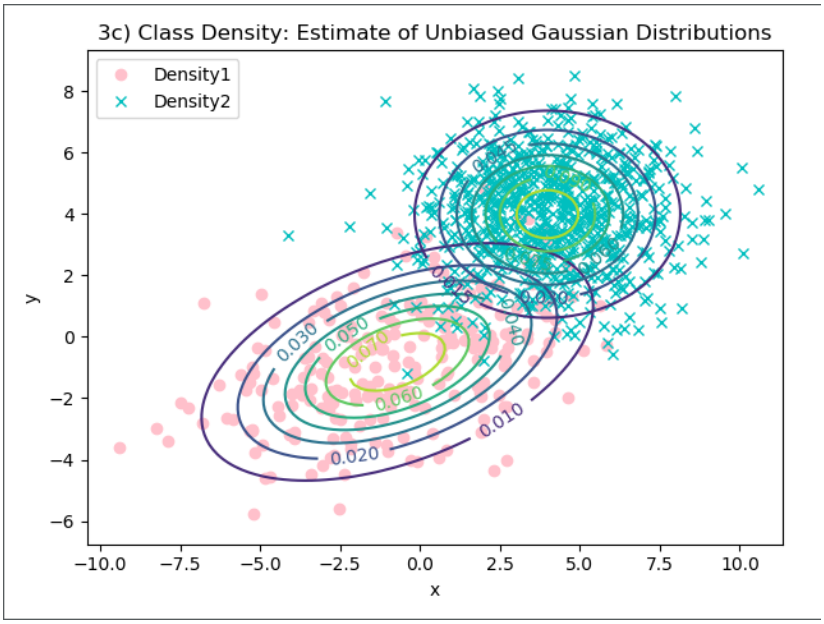
3c) Class Density

The unbiased estimates from the previous question are used to fit a Gaussian distribution to the data of each class. Codes are given below.

```
# 3c) Class Density
def multivariate_gaussian(data, mu, covar):
```

```
    """
    return likelihood for all given samples
    """
    out = np.empty(data.shape[0])
    # denominator
    y = np.sqrt((2 * math.pi) * np.linalg.det(covar))

    # compute for each datapoint
    for i, x in enumerate(data):
        diff = x - mu
        out[i] = np.exp(-0.5 * diff.T.dot(np.linalg.inv(covar)).dot(diff)) / y

    return out

# Visualization of class density
def visualize_class_density(mu, covar, data):
    steps = 100

    x_data = data[:, 0]
    y_data = data[:, 1]

    x_min = x_data.min()
    x_max = x_data.max()
    y_min = y_data.min()
    y_max = y_data.max()

    x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
    y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)

    Y, X = np.meshgrid(y, x)
    Z = np.empty((steps, steps))

    for i in range(n_models):
        for j in range(steps):
            # construct vector with same x and all possible y to cover the plot space
            points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
            Z[j] = multivariate_gaussian(points, mu[i], covar[i])
        c_plot = plt.contour(X, Y, Z)
        plt.clabel(c_plot, inline=1, fontsize=10)
```

The plot is given below showing the data points and the probability densities of each class.



3d) Posterior

In this question, the posterior distribution is found to make the classification.

```
# 3d) Posterior
def visualize_posterior(mu, covar, data, prior):
   steps = 100

   x_data = data[:, 0]
   y_data = data[:, 1]

   x_min = x_data.min()
   x_max = x_data.max()
   y_min = y_data.min()
   y_max = y_data.max()

   x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
   y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)

   Y, X = np.meshgrid(y, x)
   Z = np.empty((steps, steps))

   for i in range(n_models):
      for j in range(steps):
         # construct vector with same x and all possible y to cover the plot space
         points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
         Z[j] = multivariate_gaussian(points, mu[i], covar[i]) * prior[i]
      c_plot = plt.contour(X, Y, Z)
      plt.clabel(c_plot, inline=1, fontsize=10)

   for j in range(steps):
      # construct vector with same x and all possible y to cover the plot space
      points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
      Z[j] = multivariate_gaussian(points, mu[0], covar[0]) * prior[0] - \
           multivariate_gaussian(points, mu[1], covar[1]) * prior[1]
   c_plot = plt.contour(X, Y, Z, levels=[0])
```

```
plt.clabel(c_plot, inline=1, fontsize=10)
```

The below plot shows the posterior distribution of each class $P(C_i|x)$ and the decision boundary.

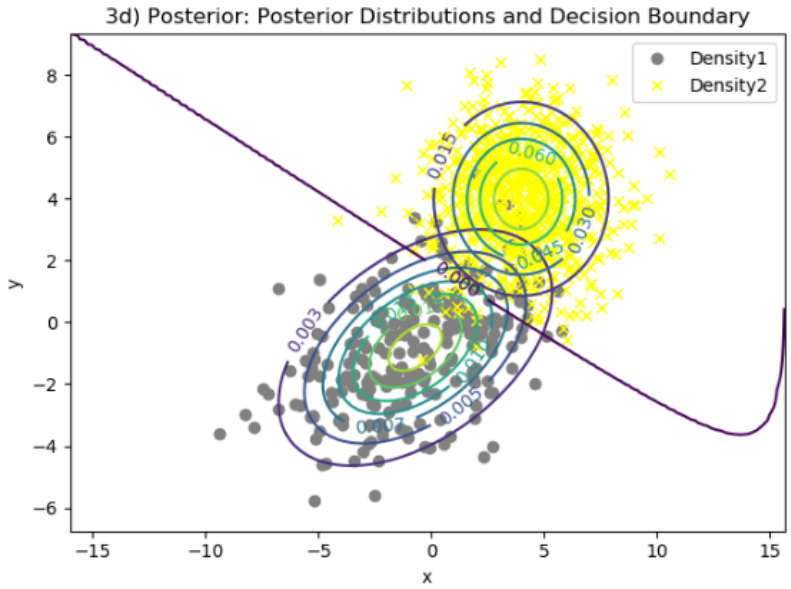


Figure 2: Posterior: Posterior Distributions and Decision Boundary

As shown in the figure, the decision boundary is not linear because the covariance matrix of the Gaussians is not identical.

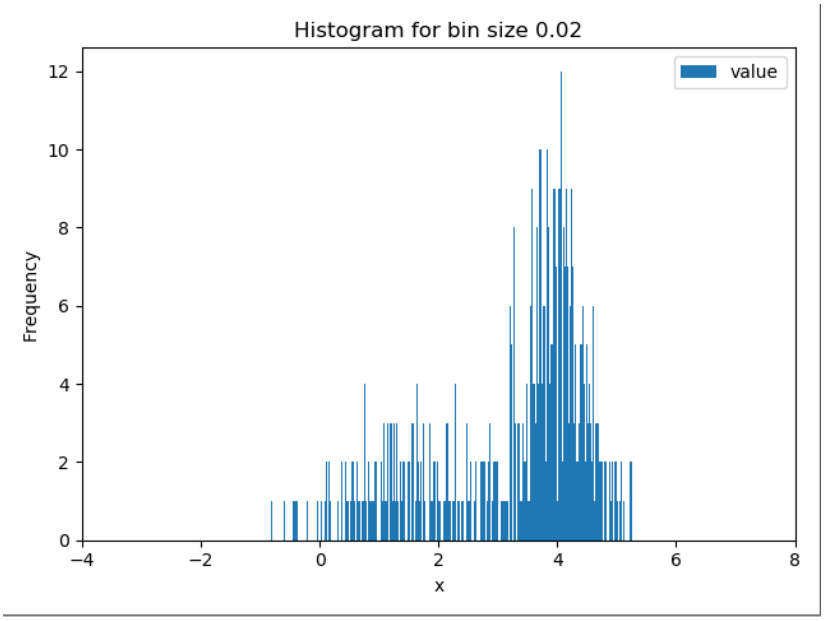
Task 4: Non-parametric Density Estimation

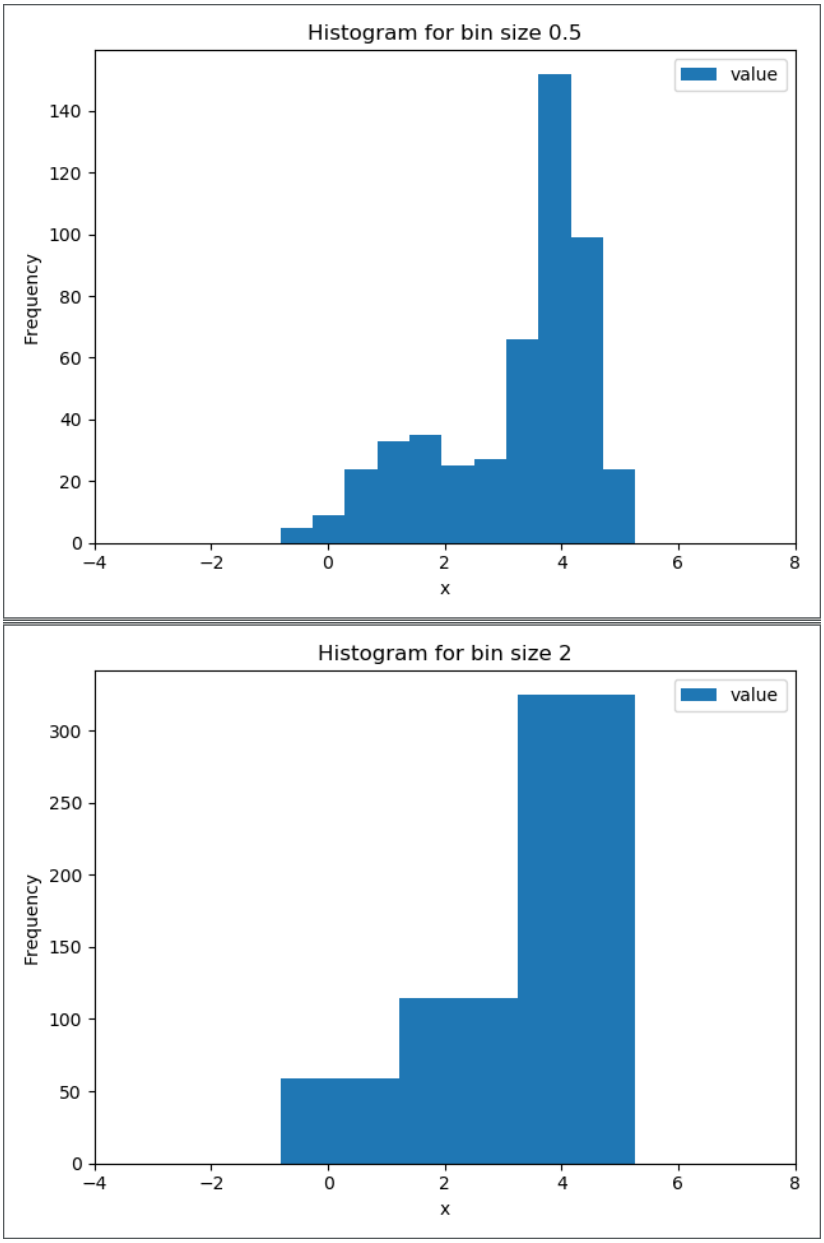
4a) Histogram

In this subtask, histograms are used to estimate the densities. The histograms using bins of size 0.02, 0.5, and 2.0 are computed in the below Python codes.

```
# 4a) Histogram
def plot_histo():
   histo_size = [0.02, 0.5, 2]

   for i, s in enumerate(histo_size):
      plt.figure(i)
      # number of bins = training_data.max().value / s
      training_data.plot.hist(by="value", bins=math.ceil(training_data.max().value / s))
      plt.xlabel("x")
      plt.title("Histogram with bin size {}".format(s))
      plt.xlim(x_min, x_max)
```

Histogram plots are given below.





As we can see from the plots, the bin size of 0.5 performs the best because there are no more gaps in the density estimation graph and the data is smoothly changing between bins.

When the bin size is 2.0, the densities can not be observable since the bins cover too many data point ranges and it is smoothing the density estimation too much. The characteristics of the distribution of data can not be seen in the histogram clearly.

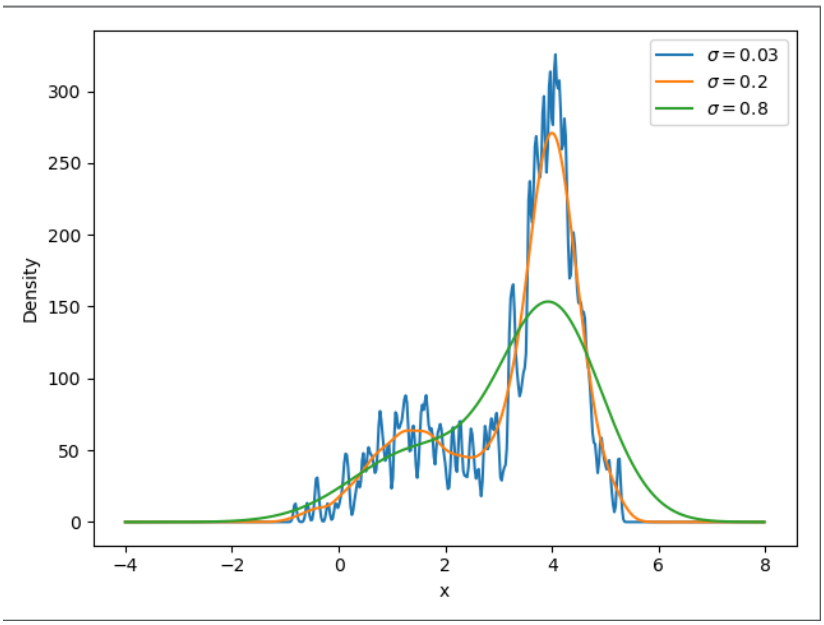
When the bin size is 0.02, it is not smooth enough and there are empty bins in the histogram.

Finally, we can not say that the bin size of 0.5 performs the best since it is possible to some bin values estimate better around different value which is close to 0.5.

4b) Kernel Density Estimate

In this question, the Gaussian kernel is used to estimate the density. The probability density estimate is computed using a Gaussian kernel with $\sigma = 0.03$, $\sigma = 0.2$, and $\sigma = 0.8$ in this code below.

```
# 4b) Kernel Density Estimate
def gaussian_KDE():
   sigmas = [0.03, 0.2, 0.8]
   steps = (x_max - x_min) / 500
   x = np.arange(x_min, x_max, steps)
   # x = np.sort(test_data.values, axis=0)
   plt.figure()
   for sigma in sigmas:

      # get log-likelihood
      # lecture05 slides48
      y = np.empty(training_data.values.shape[0])
      for i, val in enumerate(training_data.values):
         y[i] = gaussian_kernel(val, training_data.values, sigma)

      print("The train loglikelihood for sigma = {} is {}".format(str(sigma),
          str(np.sum(np.log(y)))))

      # get plots
      y = np.empty(x.shape)
      for i, val in enumerate(x):
         y[i] = gaussian_kernel(val, training_data.values, sigma)

      print("The test loglikelihood for sigma = {} is {}".format(str(sigma),
          str(np.sum(np.log(y)))))

      plt.plot(x, y, label="$\sigma=$" + str(sigma))
      plt.ylabel('Density')
      plt.xlabel('x')

   plt.legend()
   plt.show()
```



From the above figure, it can be seen that the $\sigma = 0.2$ performs best since it has low fluctuations with an acceptable Log-Likelihood.

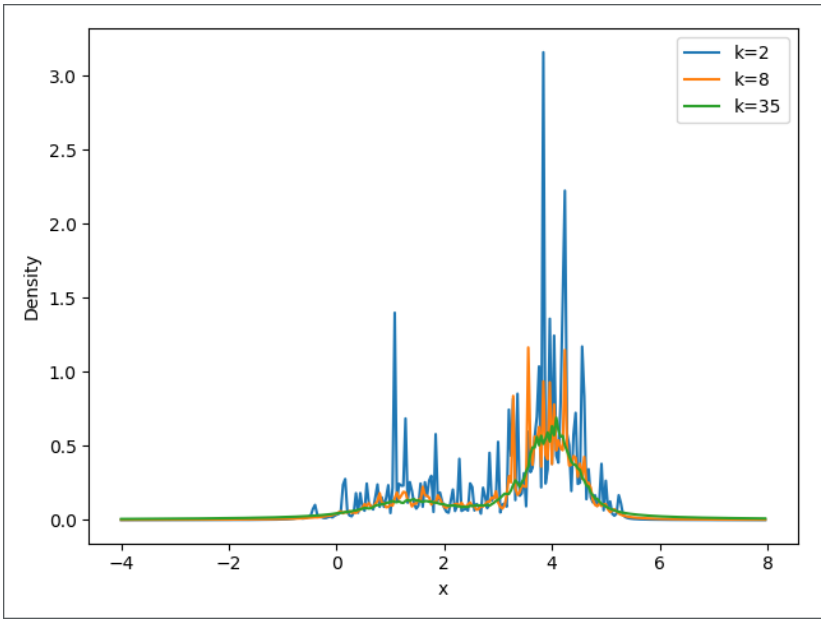
When the $\sigma = 0.8$, the density estimation is too much smoothed and does not represent the characteristics of the original distribution.

For the $\sigma = 0.03$, even though the Log-Likelihood is most desirable, the estimate fluctuates too much and the graph is too noisy. Over-fitting was observed in the plot.

4c) K-Nearest Neighbors

The probability density with the K-nearest neighbors method with $K = 2, K = 8, K = 35$ is estimated with the given python code below.

```
# 4c) K-Nearest Neighbour
def knn():
   ks = [2, 8, 35]
   steps = (x_max - x_min) / 300
   x = np.arange(x_min, x_max, steps)
   # calculate pairwise distances
   x_dist = cdist(x.reshape(x.shape[0], 1),
              training_data.values.reshape(training_data.values.shape[0], 1),
              metric="euclidean")

   for k in ks:
      y = np.empty(x.shape)
      for i, val in enumerate(x_dist):
         # find nearest k points and take point with greatest distance as Volume size
         # this assumes the distance matrix was computed with two different vectors
         # use k+1 for train data
         # np.argpartition(val, range(k))[:k] means top k element
         V = val[np.argpartition(val, range(k))[:k]][-1]
         # calculate density
         y[i] = k / (training_data.values.shape[0] * V * 2)

      print("The loglikelihood for k={} is {}"
           .format(k, np.sum(np.log(y))))

      plt.plot(x, y, label="k={}".format(k))
      plt.ylabel('Density')
      plt.xlabel('x')

   plt.legend()
   plt.show()
```



When K=2, it has a better log-likelihood value however it fluctuates too much and it is a sign of over-fitting. If we choose the K value as 35, density will be more accurate and smooth but it has the worst log-likelihood value. In our test case, the K-nearest neighbors method with K = 35 performs the best.

4d) Comparison of the Non-Parametric Methods

The log-likelihood of the testing data is estimated using the KDE estimators and the K-NN estimators. We need to test them on a different data set to avoid the problem of overfitting the model.

Log-likelihoods of the estimators for both training and testing sets are giving in the table below.

KDE Estimator			KNN Estimator		
	Training set	Testing set		Training set	Testing set
h = 0.03	2425.84	-inf	k = 2	-1256.04	-1672.38
h = 0.2	2383.64	-7853.36	k = 8	-1127.73	-1544.07
h = 0.8	2305.52	853.49	k = 35	-949.19	-1365.53

The test set is used to predict the performance of the model on previously unseen data. I would choose the KNN estimator with k = 2 because it has the highest log-likelihood on the testing set with good overfitting.

Task 5: Expectation Maximization

5a) Gaussian Mixture Update Rules

The model parameters are given below for Gaussian Mixture Update.

- $\boldsymbol{\mu}$ is the is the mean vector (2×1)
- Σ is the covariance matrix (2×2)
- $\boldsymbol{\pi}$ is the prior probability of the Gaussian distribution

The E- and M-steps of the algorithm are specified below.

- E-Step: Posterior responsibilities for each mixture component and all data points are evaluated. The specified values indicate which normal distributions fit best for data.

$$\alpha(z_{nk}) = \frac{\pi_k \mathcal{N}\left(\mathbf{x}_n|_k, \Sigma_k\right)}{\sum_{j=1}^{K} \pi_j \mathcal{N}\left(\mathbf{x}_n|_j, \Sigma_j\right)}$$

- M-Step: According to the current responsibilities the model parameters $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$ and $\boldsymbol{\pi}$ are re-estimated.

$$N_k = \sum_{n=1}^{N} \alpha(z_n k)$$

$$\overset{\text{new}}{k} = \frac{1}{N_k} \sum_{n=1}^{N} \alpha(z_{nk}) \mathbf{x}_n$$

$$\Sigma_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \alpha(z_{nk}) (\mathbf{x}_n - \overset{\text{new}}{k})(\mathbf{x}_n - \overset{\text{new}}{k})^T$$

$$\pi_k^{\text{new}} = \frac{N_k}{N}$$

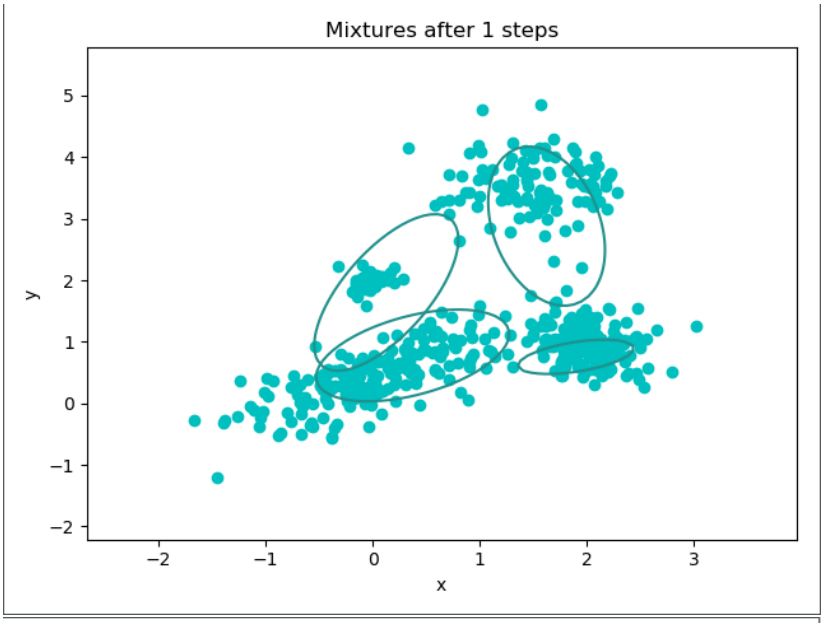
5b) EM

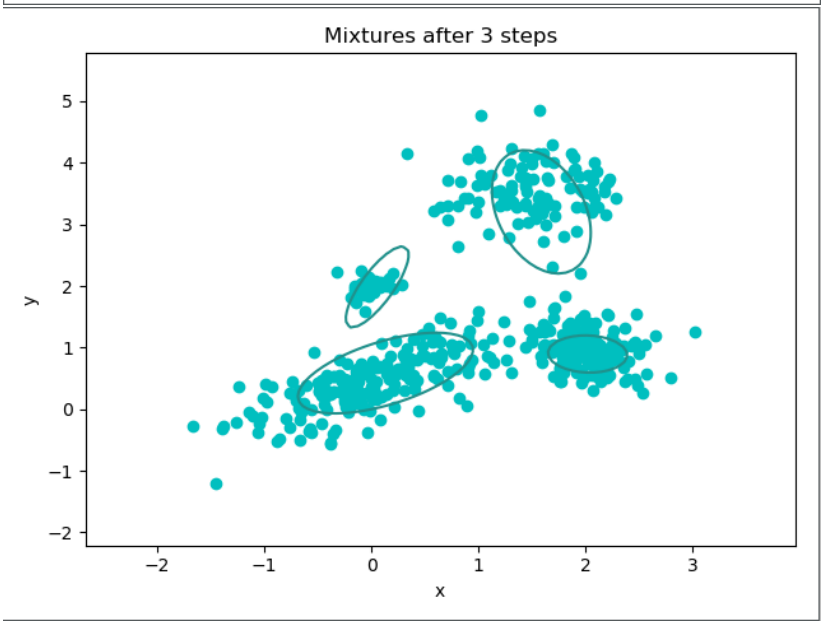
The Expectation-Maximization algorithm for Gaussian Mixture Models is implemented and initialized uniformly. The multivariate normal probability density function is computed with the help of `multivariate_normal` function from `scipy.stats`. The code is given below performs this operation.

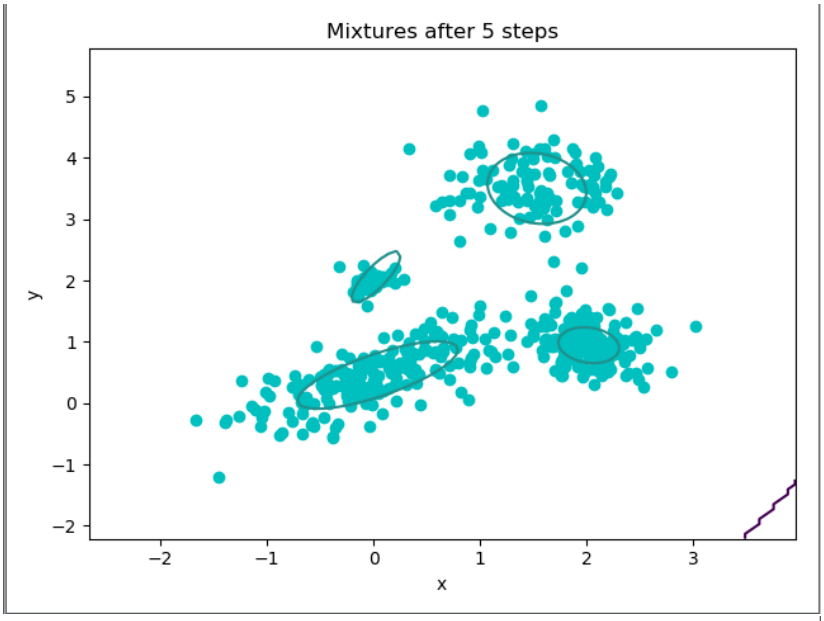
```
def e(x, mu, covar, pi):
   alpha = np.empty((k, x.shape[0]))
   for i in range(k):
      alpha[i] = pi[i] * multivariate_gaussian(x, mu[i], covar[i])

   # sum over all models per data point
   denominator = np.sum(alpha, axis=0)

   return alpha / denominator


def m(x, alpha):
   N = np.sum(alpha, axis=1) # sum over all data points per model

   mu = np.zeros((k, x.shape[1]))
```

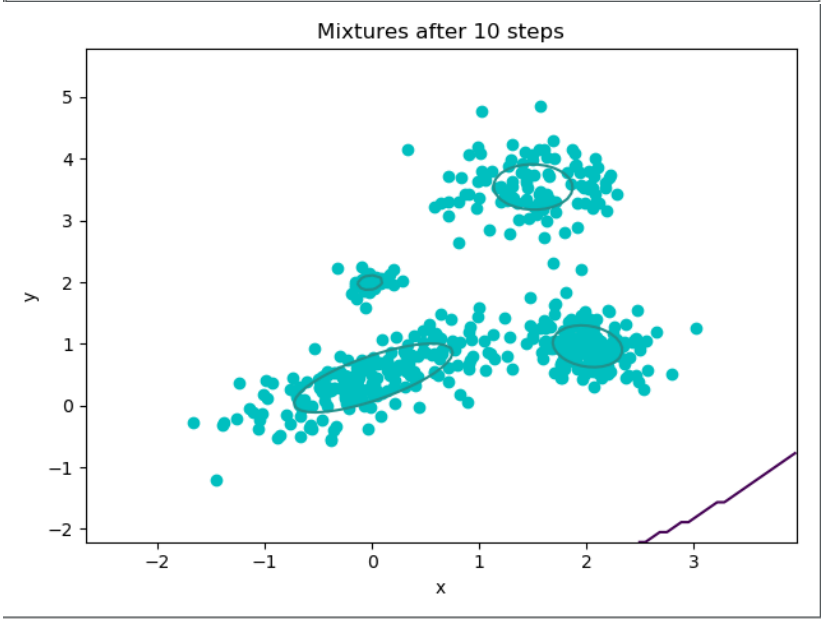
```
    covar = np.zeros((k, x.shape[1], x.shape[1]))

    for i in range(k):
        # update mu
        for j, val in enumerate(x):
            mu[i] += (alpha[i, j] * val)

        mu[i] /= N[i]

        # update covariance
        for j, val in enumerate(x):
            diff = val - mu[i]
            covar[i] += alpha[i, j] * np.outer(diff, diff.T)

        covar[i] /= N[i]

    # update pi
    pi = N / x.shape[0]

    return mu, covar, pi

def visualize(mu, covar, data, iteration):

    steps = 100

    x_data = data[:, 0]
    y_data = data[:, 1]

    x_min = x_data.min()
    x_max = x_data.max()
    y_min = y_data.min()
    y_max = y_data.max()

    x = np.arange(x_min - 1, x_max + 1, (x_max - x_min + 2) / steps)
    y = np.arange(y_min - 1, y_max + 1, (y_max - y_min + 2) / steps)

    Y, X = np.meshgrid(y, x)
    Z = np.empty((steps, steps))

    for i in range(k):
        for j in range(steps):
            # construct vector with same x and all possible y to cover the plot space
            points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
            Z[j] = multivariate_gaussian(points, mu[i], covar[i])
        plt.contour(X, Y, Z, 1)

    # plot the samples
    plt.plot(x_data, y_data, 'co', zorder=1)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title("Mixtures after {} steps".format(iteration + 1))
```

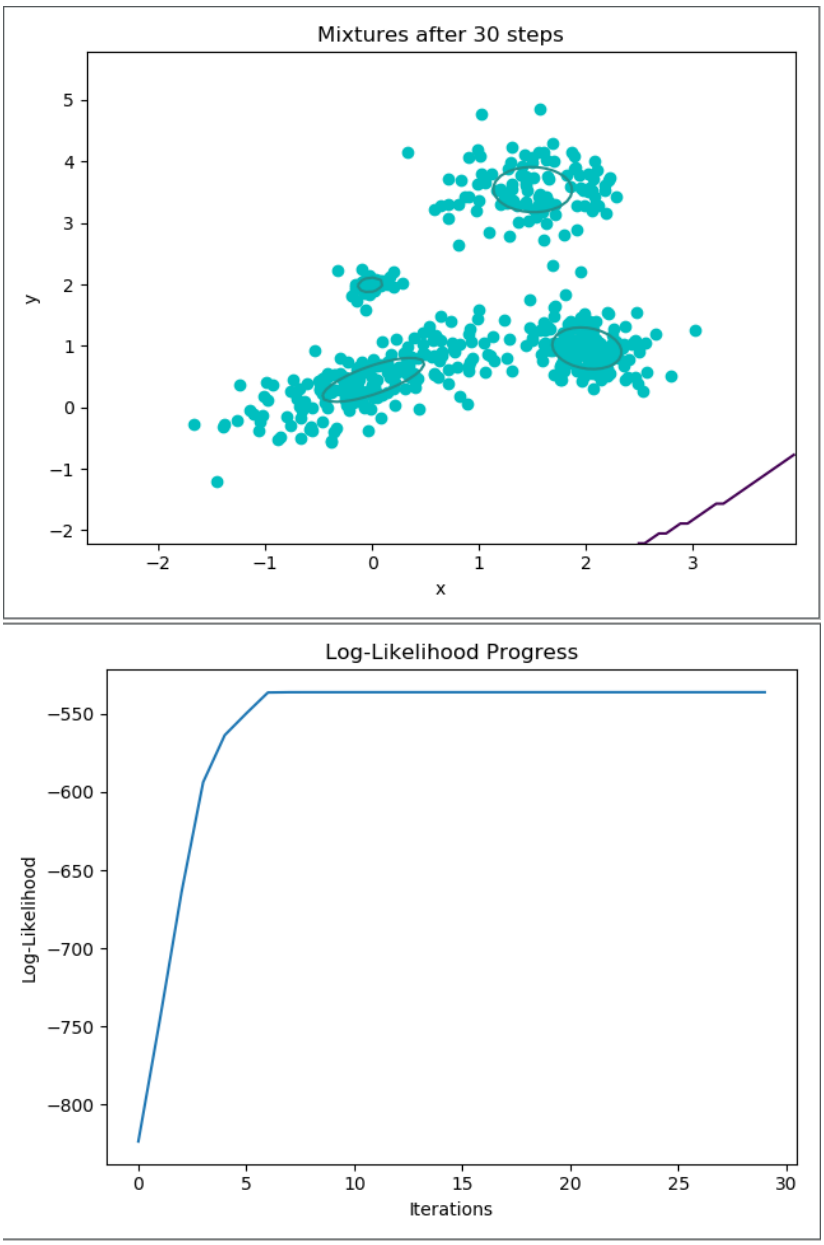
Plots at different iterations $t_i \in 1, 3, 5, 10, 30$ are generated for showing the data and the mixture components. The log-likelihood for every iteration is plotted as well.


Mixtures after 1 steps
5
4
3
2
1
0
−1
−2
y
−2
−1
0
1
2
3
x


Mixtures after 3 steps
5
4
3
2
1
0
−1
−2
y
−2
−1
0
1
2
3
x


Mixtures after 5 steps
y
x
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3


Mixtures after 10 steps
y
x
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3



It is obvious that the algorithm has already been achieved to cluster all data points in the 30th iteration. Besides that, according to the log-likelihood, there is no improvement in the estimation after the 8th iteration so that the stable state has already been achieved.