SML - Exercise 2

Maximilian Nothnagel, Stefanie Martin

1 Task 1: Optimization

1.1 1a) Numerical Optimization

```
#Function for calculating f'(x)
def solveDerivative(x):
#Calculate Derivative f'(x)
result = 400 * math.pow(x, 3) + (2-400*(x+1)) * x-2
#print('f\'(', x,') =', result) #print result
return result

n = 20
#needs to be surprisingly small for decent results.
learningRate = 0.000001
#Randomly determine starting x
curX = random.randint(1, n-1)
for curIteration in range(1, 10000):
curResult = solveDerivative(curX)
print(curIteration, " : f\'(", curX,") =", curResult)
#negate to always move towards negative
diff = learningRate * -curResult
#apply to curX for next iteration
curX += diff
if curIteration % 2000 == 0:
input("Press Enter to continue...")

print("final X is", curX, "; with f'(",curX,")= ",
                              solveDerivative(curX) )
```

Learning rate impacts the size of the "steps" we take in each iteration. Since we are approximating, we generally step over the lowest point at some point, and then start stepping back and forth over the lowest point.
A smaller learning rate will lead to smaller steps while moving towards the lowest point, but to a larger gap created by stepping back and forth over the lowest point once it has been found.
Larger learning rates lead to fast, unaccurate conclusions, smaller rarely come to a conclusion within 10k iterations.
Choose the learning rate just right to get a accurate conclusion, while still arriving at the conclusion in most cases. (conclusion meaning the lowest point)
Samples taken from one running of the script:

$$\#0001 f'(18,0) = 2196034,0$$
$$\#1000 f'(1,8132) = 345,853$$
$$\#2000 f'(1,6567) = 59,694$$
$$\#3000 f'(1,6262) = 13,164$$
$$\#4000 f'(1,6193) = 3,050$$
$$\#5000 f'(1,6177) = 0,715$$
$$\#6000 f'(1,6173) = 0,1679$$
$$\#7000 f'(1,61721) = 0,0395$$
$$\#8000 f'(1,617187) = 0,00923$$
$$\#9000 f'(1,61718799) = 0,0021822$$
$$\#10000 f'(1,6171806) = 0,0005131$$

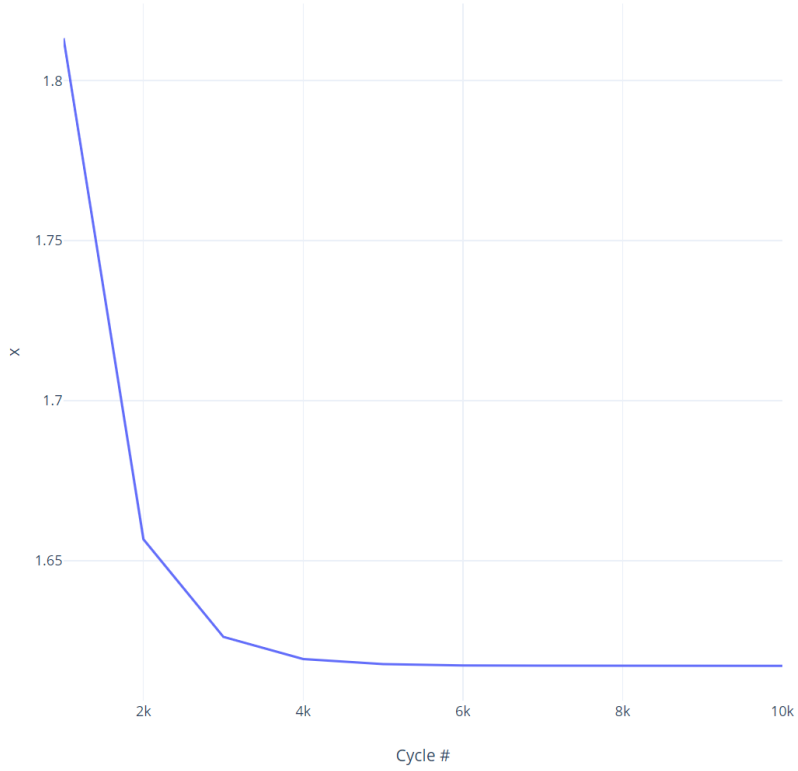


Figure 1: Graph for f'(x) throughout the cycles.
Cycle #1 excluded for scaling. We can observe a rapid descent of the gradient, slowing down as it approaches 0

1.2 1b) Gradient Descent Variants

1.2.1 1)

Batch Gradient Descent: From a random starting point, move toward the direction of lowest gradient a distance based on the Gradient. Repeat. Very accurate, but computationally expensive.
Stochastic Gradient Descent: From a random starting point, pick a random point P2 in the sorrounding. move roughly towards P2, following the direction of the lowest Gradient. Repeat. This reduces the computing in each step, opting instead for random generation.
Mini-Batch Gradient Descent: Create Mini-Batch of datapoints out of the whole set. Compute mean gradient for the Mini-batch, use it in selecting for the next Mini-Batch. Repeat. Saves Memory, by only ever working on small parts of the dataset.

1.2.2 2)

Momentum is the idea of taking greater or smaller steps based on the results of each step. This is easily recognized in Batch/Vanilla Gradient Descent, and also present in the other variants.

2 Task 2: Density Estimation - MLE

2.1 2a) Maximization Likelihood Estimate of the Exponential Distribution

Using Python we computed the maximum propabilities of the given function for x=[0..100] and s=[-40..40]
$\frac{1}{s} * e^{-\frac{x}{s}}$

$Maximum of s = -40:$	-0.025
$Maximum of s = -20:$	-0.05
$Maximum of s = -15:$	-0.06666666666666667
$Maximum of s = -10:$	-0.1
$Maximum of s = -5:$	-0.2
$Maximum of s = -4:$	-0.25
$Maximum of s = -3:$	-0.3333333333333333
$Maximum of s = -2:$	-0.5
$Maximum of s = -1:$	-1.0
$Maximum of s = 0.01:$	-100.0
$Maximum of s = 0:$	$Undefined$
$Maximum of s = 0.01:$	100.0
$Maximum of s = 1:$	1.0
$Maximum of s = 2:$	0.5
$Maximum of s = 3:$	0.3333333333333333
$Maximum of s = 4:$	0.25
$Maximum of s = 5:$	0.2
$Maximum of s = 10:$	0.1
$Maximum of s = 15:$	0.06666666666666667
$Maximum of s = 20:$	0.05
$Maximum of s = 40:$	0.025

When s converges towards 0, the likelihood rises to infinity.
When s rises towards $\pm\infty$, the likelihood approaches 0

3 Task 3: Density Estimation

3.1 3a) Prior Probabilities

Putting the data into a python script and averaging out the classes yields the following priors:
C1 Prior: 0.24
C2 Prior: 0.76

3.2 3b) Biased ML Estimate

An estimator's bias describes a tendency of the estimates to differ from the parameter being estimated in a consistent way.

Calculate the mean and deviation:

```
def CalcMean(data):
    total = 0
    entryCount = 0
    for entry in data:
        total += entry
        entryCount += 1
    return total / entryCount
```

```
def CalcDeviation(data, mean):
    distArray = []
    distTotal = 0
    for entry in data:
        distTotal += pow(mean - entry, 2)
    final = math.sqrt(distTotal / len(data))
    return final
```

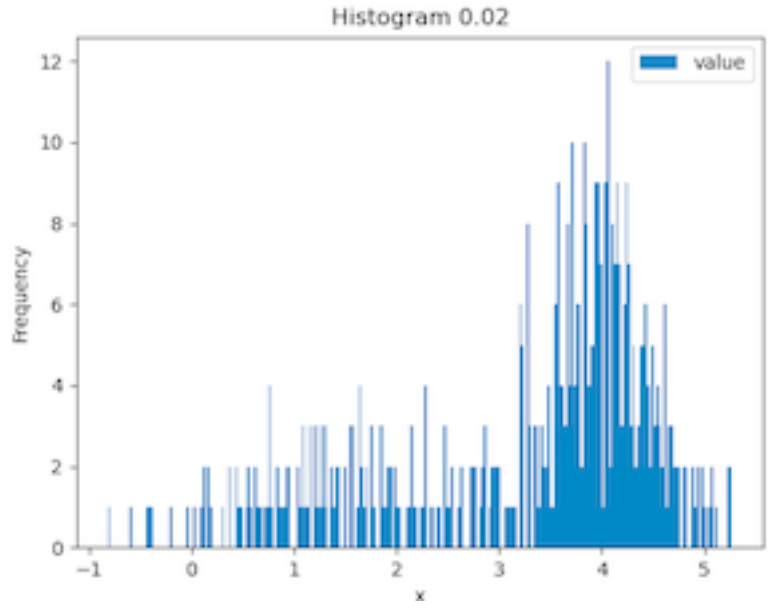
$C1Mean : -0.775; C2mean : 3.986$
$C1deviation : 2.531; C2deviation : 1.861$
Maximum likelihood function for Gaussian:
$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma}} * e^{\frac{-(x-\mu)^2}{2\sigma^2}}$
Plugged in:
$p(x|C_1) = 0,158e^{\frac{-x^2}{12,812}+0,121x+0,468}$
$p(x|C_2) = 0,214e^{\frac{-x^2}{6,927}-1,151x+2,294}$

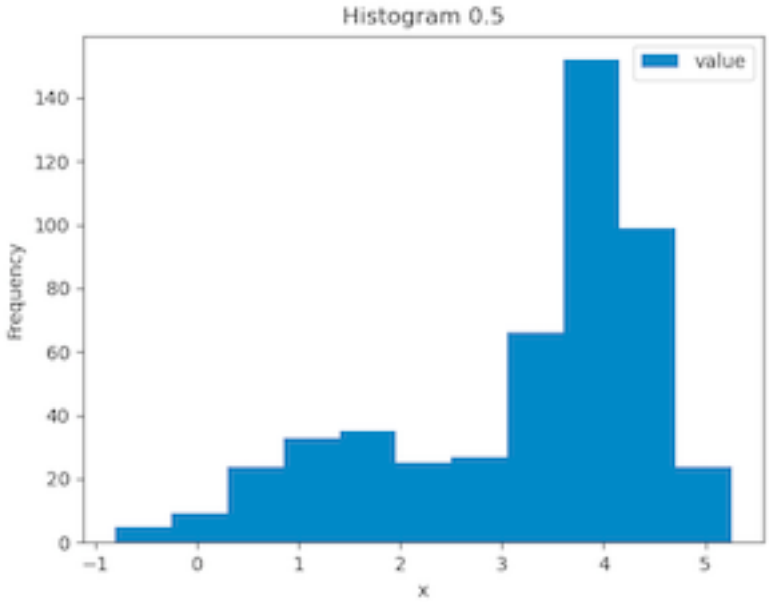
```
def CalcLikelihood(x, mean, deviation):
    r1 = 1/math.sqrt(2*math.pi*pow(deviation, 2))
    #print(r1)
    r2 = -pow(x-mean, 2)
    r2 = -pow(x, 2) +2*mean*x -pow(mean,2)
    #print(r2)
    r3 = 2*pow(deviation, 2)
    #print(r3)
    result = r1 * pow(math.e, r2/r3)
    return result

def CalcMaxLikelihood(data, mean, deviation):
    result = 0.0
    for entry in data:
        result += CalcLikelihood(entry, mean, deviation)
    return result
```

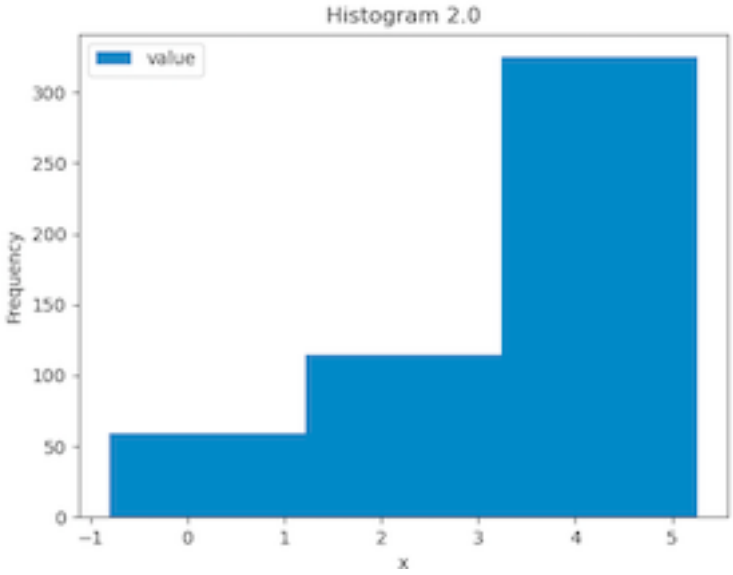
Max Likelihoods by summing up the results for each x:
$C1MaxLikelihood : 53.60138072754072; C2MaxLikelihood : 230.85569153085527$

4 Task 4: Non-parametric Density Estimation

4.1 4a) Histogram



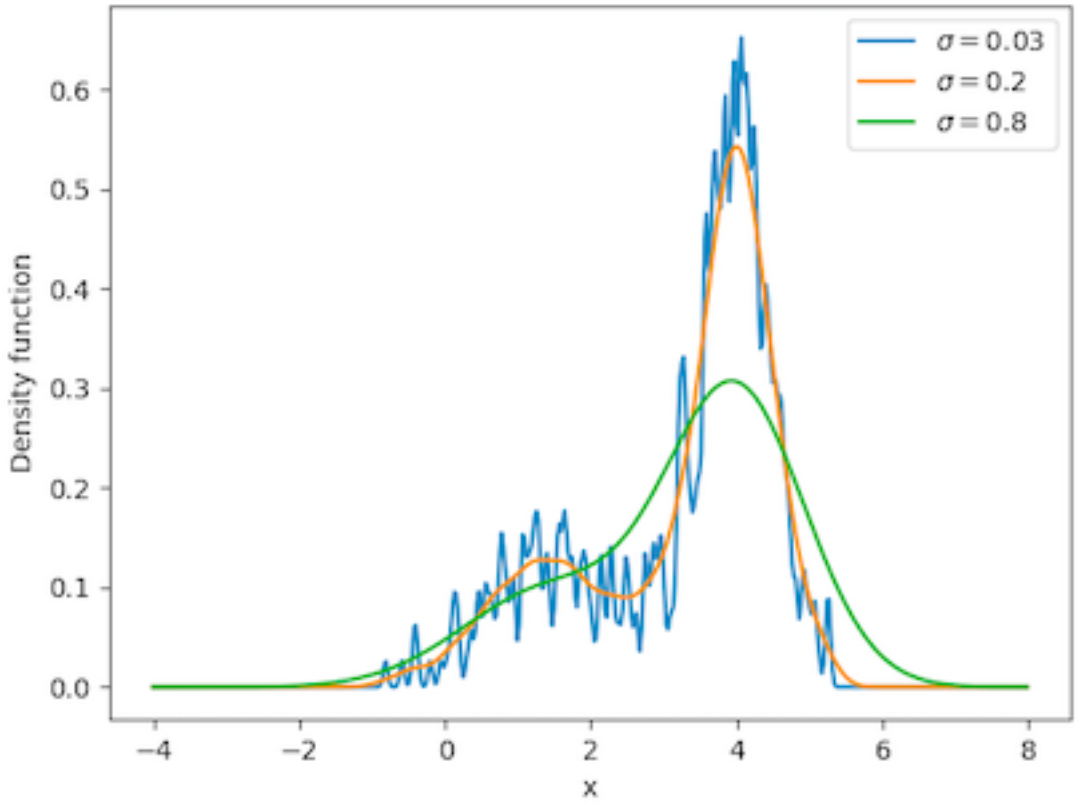




For a bin size of 0.02 the histogram is not smooth enough. For a bin size of 0.5 it seems to be about right, it is the best of the 3 histograms. The histogram for the bin size of 2.0 is too smooth. Since we only did 3 trials, there is a chance that there is an even better output possible, close to 0.5.

```
# Code for histogram
trainingData = p.read_csv("nonParamTrain.txt", sep="\s{2}")
testData = p.read_csv("nonParamTest.txt", sep="\s{2}")
trainingData.columns = testData.columns = ["value"]


def histo():
    size = [0.02, 0.5, 2.0]

    for i, s in enumerate(size):
        plt.figure(i)
        trainingData.plot.hist(by="value", bins=math.ceil(trainingData.max().val
        plt.xlabel("x")
        plt.title("Histogram {}".format(s))

histo()
plt.show()
```

4.2 4b) Kernel Density Estimate

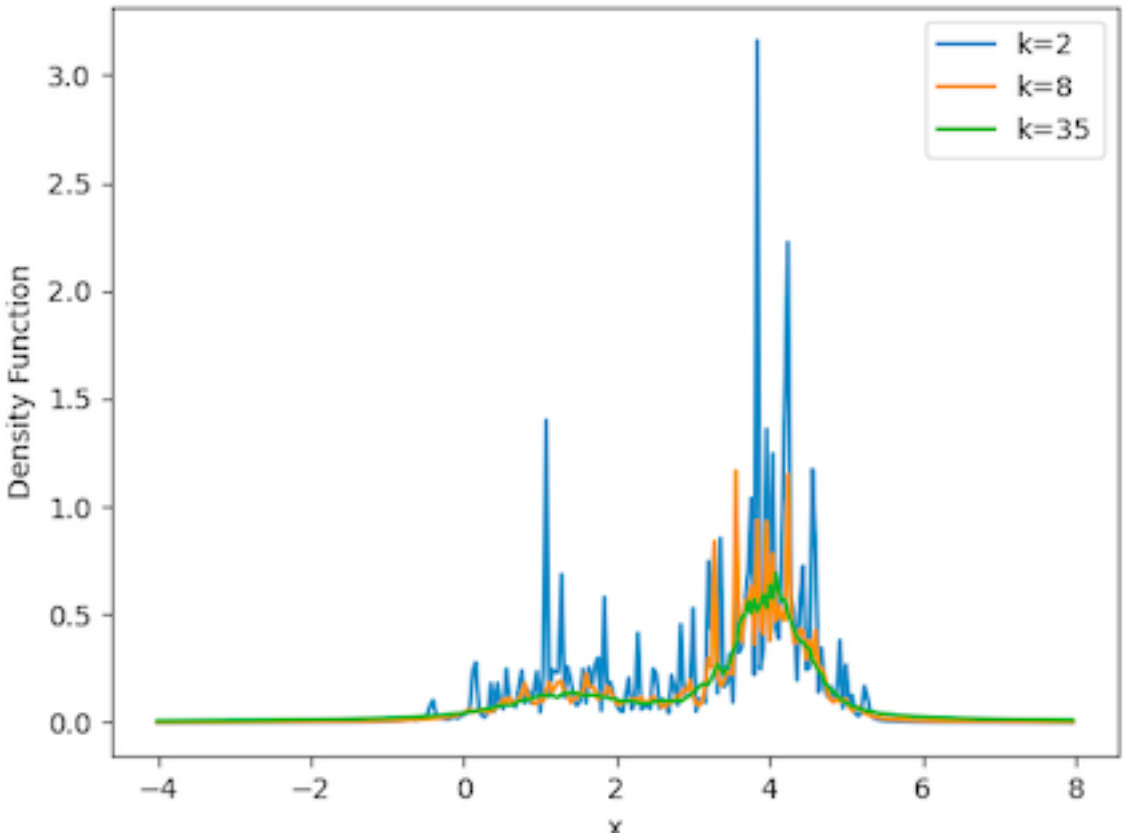


The log-likelihood for 0.03 is −674.25, for 0.2 it is −716.46 and for 0.8 −794.57.
There for it looks like 0.03 is performing the best.

```
# Code for kde
min = -4
max = 8

def kernel(x, data, sigma):
    return (np.sum(np.exp(-(x - data) ** 2 / (2 * sigma ** 2))))
            / (np.sqrt(2 * math.pi) * len(data) * sigma)


def kde():
    sigmas = [0.03, 0.2,.8]
    x = np.arange(min, max, ((max - min) / 500))
    plt.figure()
```

```
    for sigma in sigmas:

        y = np.empty(trainingData.values.shape[0])
        for i, val in enumerate(trainingData.values):
            y[i] = kernel(val, trainingData.values, sigma)

        y = np.empty(x.shape)
        for i, val in enumerate(x):
            y[i] = kernel(val, trainingData.values, sigma)

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

    plt.legend()
    plt.show()

kde()
```

4.3 4c) K-Nearest Neighbors



For k of 2 we get an log-likelihood of -1256.04. For k of 8 we get an log-likelihood of -1127.73 and for k of 35 we get -949.19. There for it looks like 35 is performing best.

```
# Code for knn
min = -4
max = 8

def knn():
    ks = [2, 8, 35]
    x = np.arange(min, max, ((max - min) / 300))

    d = cdist(x.reshape(x.shape[0], 1),
        trainingData.values.reshape(trainingData.values.shape[0], 1),
        metric="minkowski")
```

```
    for k in ks:
        y = np.empty(x.shape)
        for i, val in enumerate(d):
            V = val[np.argpartition(val, range(k))[:(k)]][-1]

            y[i] = k / (trainingData.values.shape[0] * V * 2)

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

    plt.legend()
    plt.show()

knn()
```

4.4 4d) Comparison of the Non-Parametric Methods

Sigma	KDE	
	Training	Test
0.02	-674.25	-inf
0.5	-716.46	-10959.67
2.0	-794.57	-2252.82
K	**K-NN**	
	Training	Test
2	-1256.04	-1358.73
8	-1127.73	-1270.44
35	-949.19	-1127.34

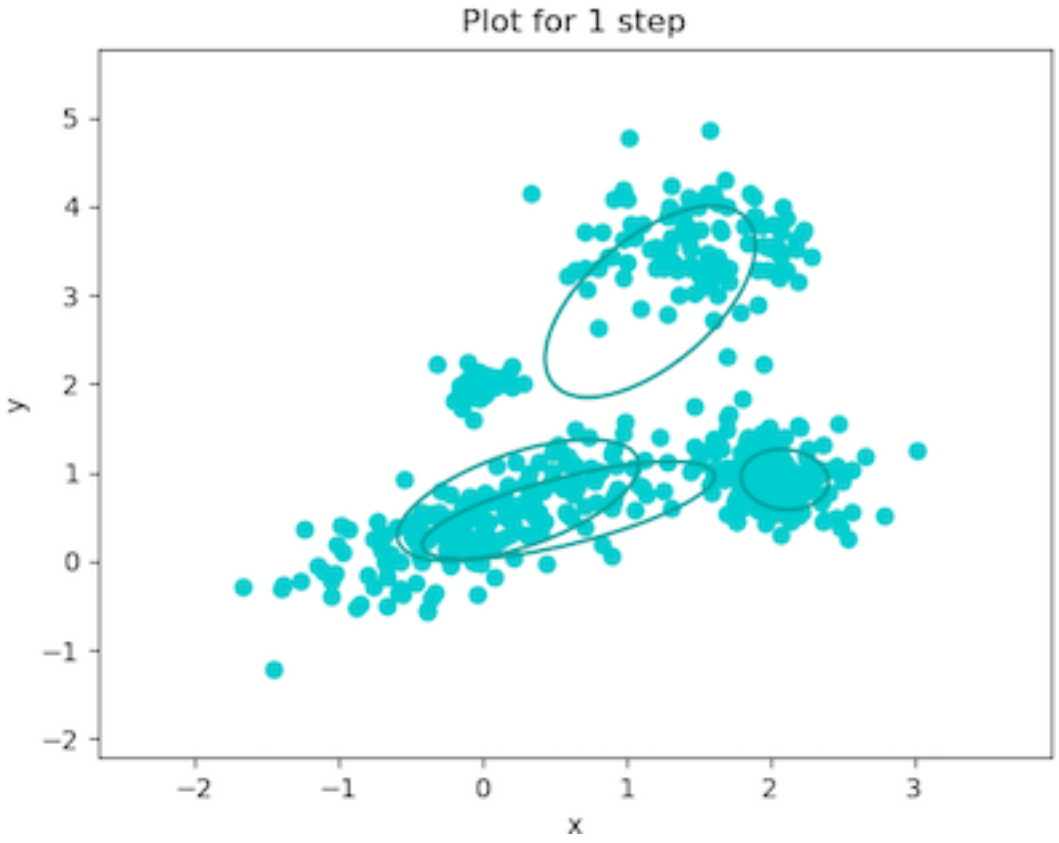
We use the test data to build up our model / to fit our model. Then later we use the test data to validate / test our model.

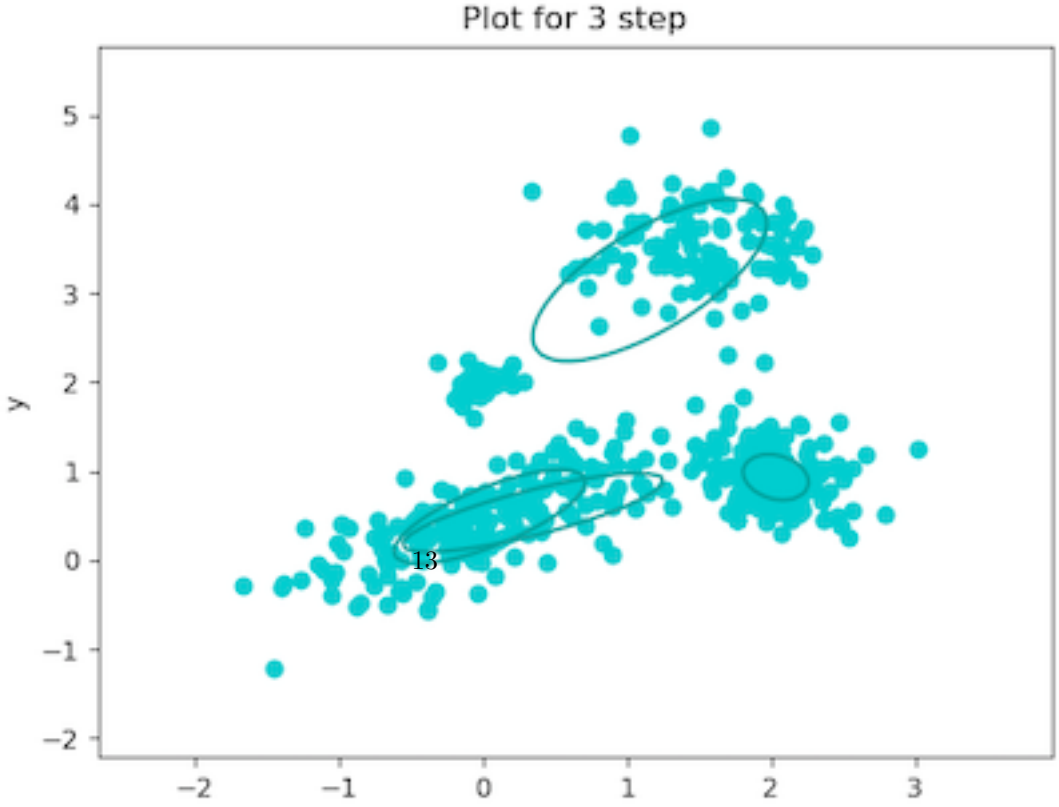
5 Task 5: Expectation Maximization

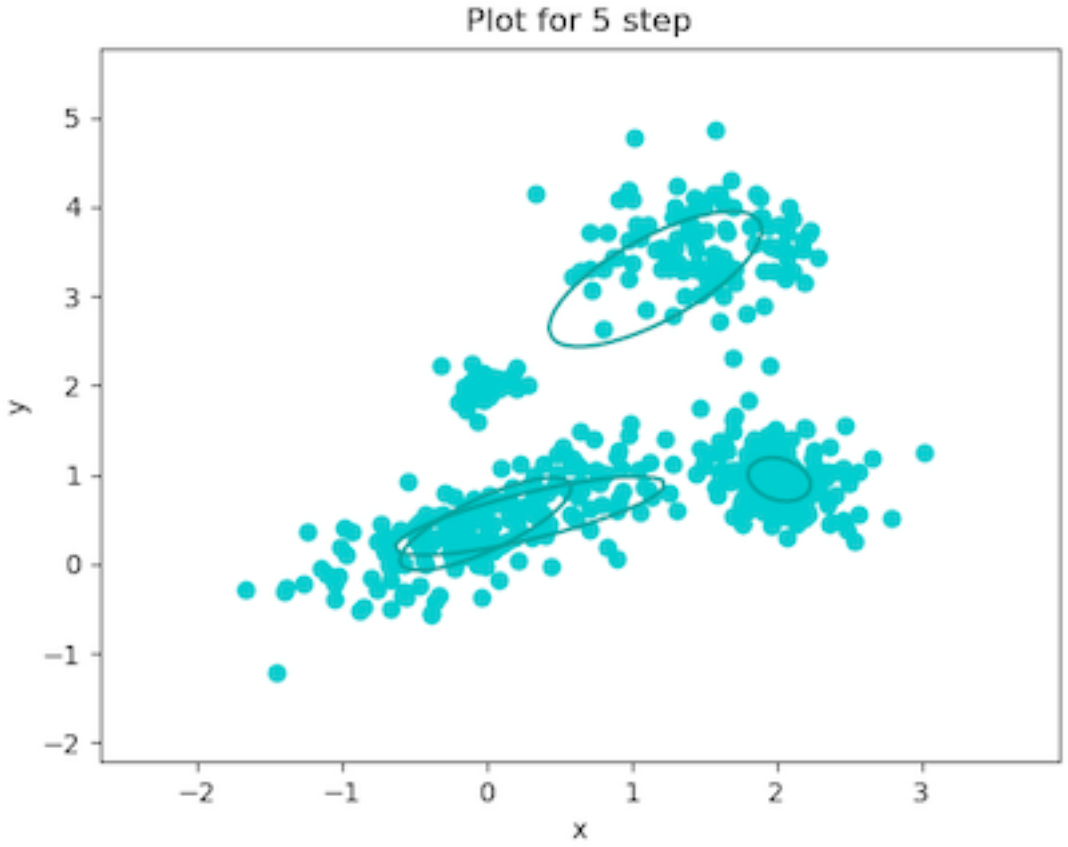
5.1 5a) Gaussian Mixture Update Rules

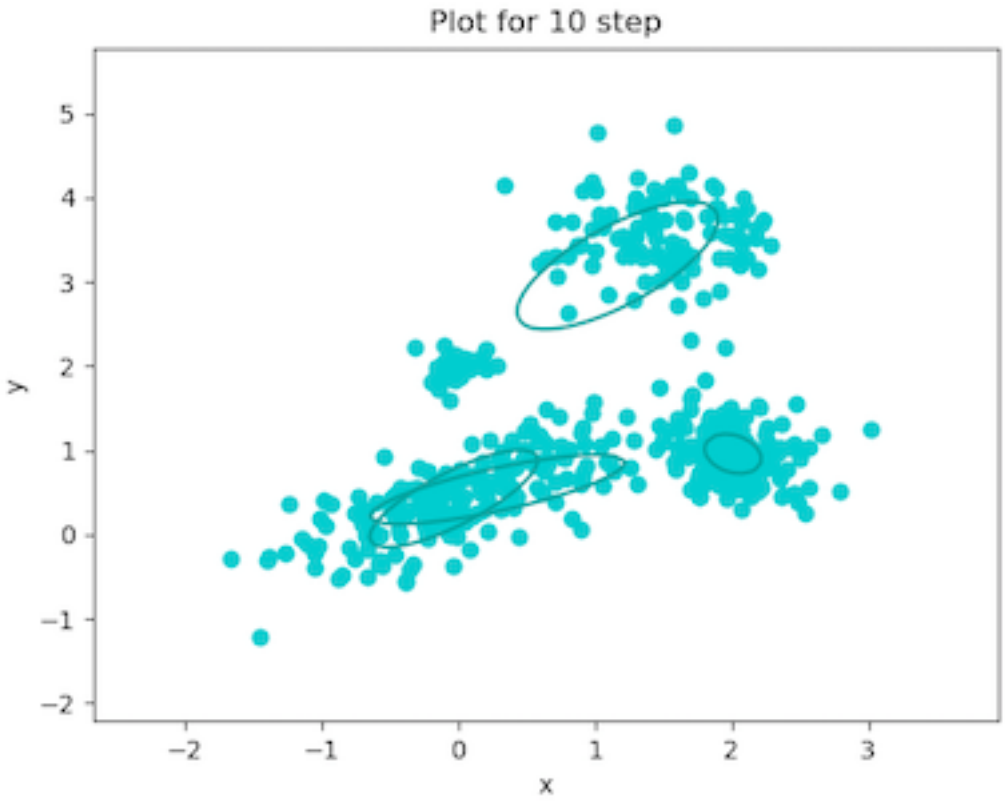
- Compute the E-Step using Gaussian $E = Q(\Theta, \Theta^{i-1}) = [\int]P(Y|X, \Theta^{i-1})logP(X, Y|\Theta)dy$ with X incomplete data and Y Hidden Data
- Update Rule $M = \Theta^i = arg[\max_\Theta]E$

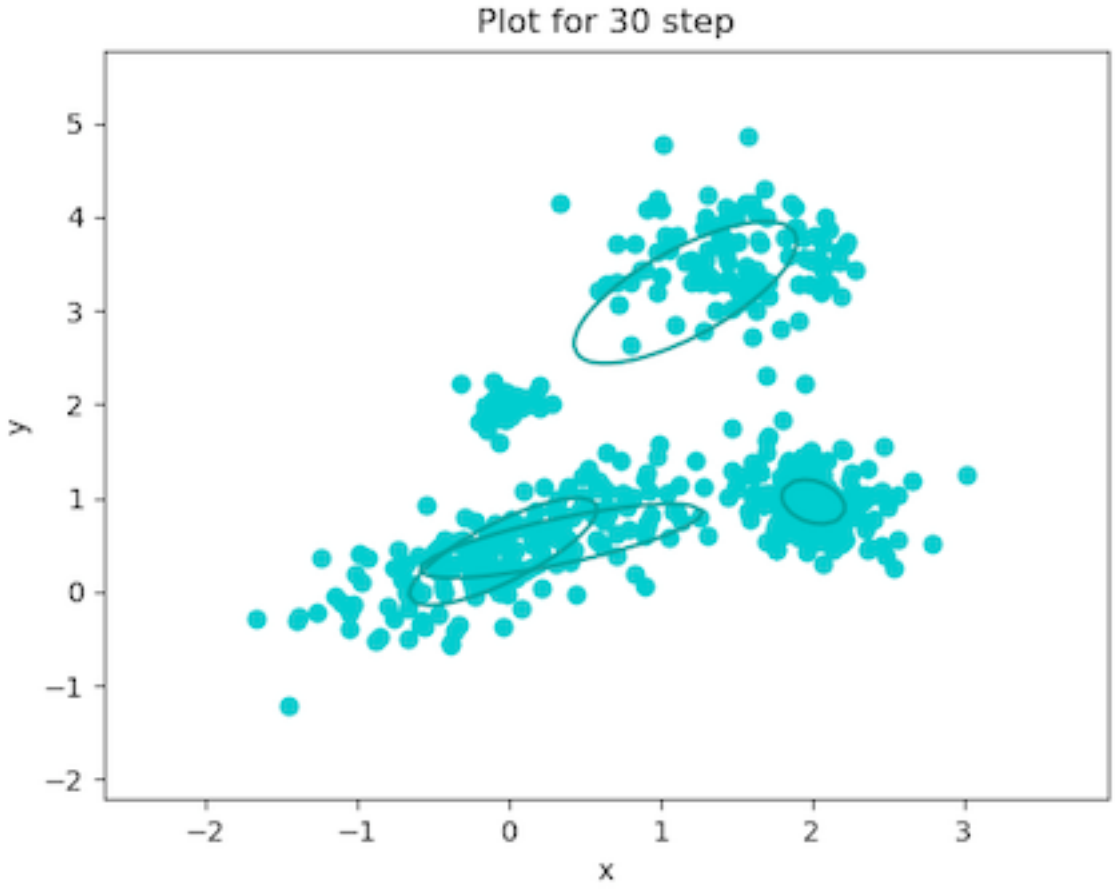
5.2 5d) EM

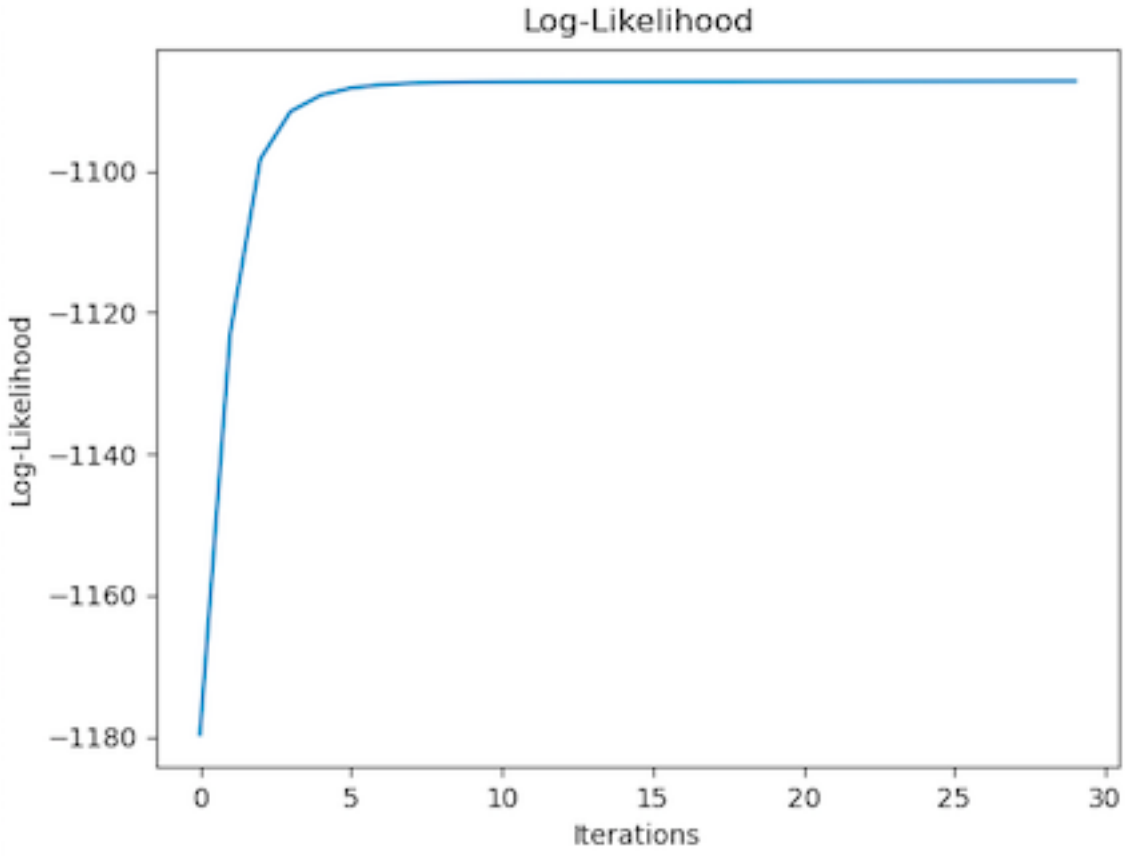

Plot for 1 step
y
x
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3


Plot for 3 step
y
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3


Plot for 5 step
y
x
5
4
3
2
1
0
−1
−2
−2
−1
0
1
2
3


Plot for 10 step
5
4
3
2
1
0
−1
−2
y
−2
−1
0
1
2
3
x


Plot for 30 step
5
4
3
2
1
0
−1
−2
y
−2
−1
0
1
2
3
x




```
#Code for exception maximization
k = 4

def em():

    covar = np.array(
        [np.identity(data.shape[1]) for _ in range(k)])
    mu = np.random.uniform(data.min().min(), data.max().max(),
                           size=(k, data.shape[1]))
    pi = np.random.uniform(size=(k,))
    likelihood = np.empty((30,))

    for i in range(30):
        alpha = e( covar, pi, mu, data.values)
        mu, covar, pi = m(data.values, alpha)
```

```
        if i + 1 in [1, 3, 5, 10, 30]:
            plt.figure(i)
            vis(data.values, i, mu, covar)
        likelihood[i] = logLikelihood(data.values, mu, covar, pi)

def m(x, alpha):
    N = np.sum(alpha, axis=1)

    m = np.zeros((k, x.shape[1]))
    c = np.zeros((k, x.shape[1], x.shape[1]))

    for i in range(k):
        for j, val in enumerate(x):
            m[i] += (alpha[i, j] * val)
        m[i] /= N[i]
        for j, val in enumerate(x):
            c[i] += alpha[i, j] * np.outer(val - m[i], (val - m[i]).T)
        c[i] /= N[i]
    pi = N / x.shape[0]

    return m, c, pi

def e(c, pi, m, x):
    alpha = np.empty((k, x.shape[0]))
    for i in range(k):
        alpha[i] = pi[i] * gaussian(x, m[i], c[i])

    return alpha / np.sum(alpha, axis=0)

def gaussian(data, m, c):
    res = np.empty(data.shape[0])
    for i, x in enumerate(data):
        diff = x - m
        res[i] = np.exp(-.5 * diff.T.dot(np.linalg.inv(c)).dot(diff)) \
        / np.sqrt((2 * math.pi) ** data.shape[1] * np.linalg.det(c))

    return res

def logLikelihood(x, m, c, pi):
    logLikelihood = np.empty((k, x.shape[0]))
    for i in range(k):
        logLikelihood[i] = pi[i] * gaussian(x, m[i], c[i])

    return np.sum(np.log(np.sum(logLikelihood, axis=0)))

def vis(data, iteration, m, c):
```

```
    x = np.arange(data[:, 0].min() - 1, data[:, 0].max() + 1,
        (data[:, 0].max() - data[:, 0].min() + 2) / 100)
    y = np.arange(data[:, 1].min() - 1, data[:, 1].max() + 1,
        (data[:, 1].max() - data[:, 1].min() + 2) / 100)
    Y, X = np.meshgrid(y, x)
    Z = np.empty((100, 100))

    for i in range(k):
        for j in range(100):
            points = np.append(X[j], Y[j]).reshape(2, x.shape[0]).T
            Z[j] = gaussian(points, m[i], c[i])
em()
```