SML Assignment 4

Casper Van Elteren, Daniel Mingers

January 12, 2017

Exercise 1 - Gaussian Processes

January 12, 2017

1.1

Not much to say it does what it says on the tin; implements equation (3) of the pdf.

```
In [1]: from pylab import *
        from numpy import *

        close('all')
        #  q1
        def kernel(x1, x2, theta):
            """
            Code for equation 3
            """
            sigma = np.zeros((len(x1), len(x2)))
            for idx, i in enumerate(x1):
                for jdx, j in enumerate(x2):
                    sigma[ idx, jdx ] = \
                        theta[ 0 ] *\
                        np.exp(- .5 * theta[ 1 ] * np.sqrt((i - j)**2)) + \
                        theta[ 2 ] + theta[ 3 ] * i * j
            return sigma
```

1.2

Define theta and the linearly spaced points and use the kernel function from 1.

```
In [2]: #q2
        theta = ones(4)
        N = 101
        x = linspace(-1,1, N)
        K = kernel(x, x, theta)
```

1.3

The Gramm matrix would be 101x101, since it should reflect the kernel between all the datapoints. In order to show that a matrix is semipositive definite we need to show that all eigenvalues are non-negative, $\lambda_i \geq 0$.

```
In [3]: print('Shape of K', K.shape)
        eigenValues, eigenVectors = linalg.eig(K)
        print(r'are all eigenvalues non-negative?', all(eigenValues >= 0))

Shape of K (101, 101)
are all eigenvalues non-negative? True
```

As we can see all the eigenvalues are non-negative. Hence, our K is semi-positive definite.

1.4

Please see figure 1.

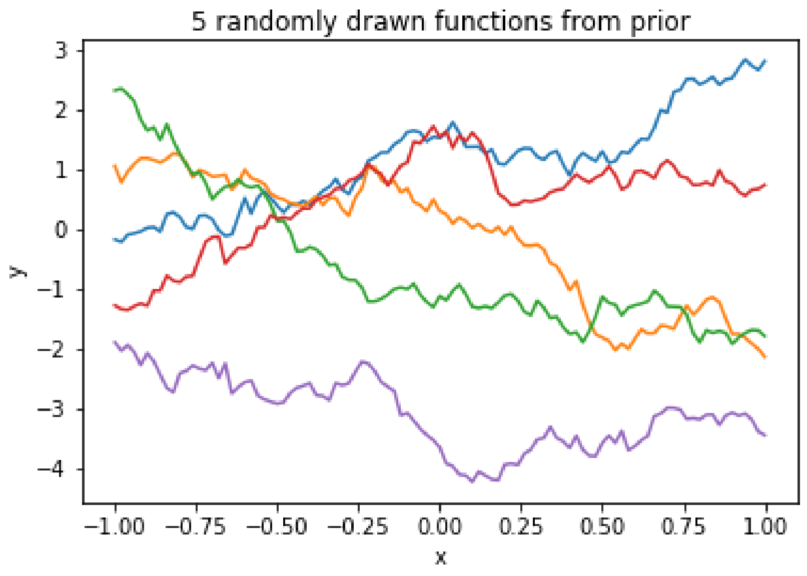


Figure 1: Results for 1.4; 5 randomly drawn functions drawn from the prior over the input space.

```
In [4]: # q4
        # import multivariate normal object
        from scipy.stats import multivariate_normal as mv
        # define mu / sigma
        mu = zeros(len(x))
        # prior object
        prior = mv(mu, K)
        # sample from prior
        samples = prior.rvs(5)


In [5]: # show samples from the prior
        fig, ax = subplots(1, 1)
        for i in samples:
            ax.plot(x, i)
        savefig('../Figures/ex1q5')
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_title('5 randomly drawn functions from prior')
        savefig('../Figures/1.4.png')
```

2.5

The code is self-explanatory; feed the kernel the training set with different theta values. The results are displayed in ??.

```
In [6]: # q5
        thetas = np.array([ [ 1, 4, 0, 0 ],\
                           [ 9, 4, 0, 0 ],\
                           [ 1, 64, 0, 0 ],
                         [ 1, 0.25, 0, 0 ],\
                           [ 1, 4, 10, 0 ],\
                           [ 1, 4, 0, 5 ] ])

        # we want at most 3 rows
        nRows = 3
        # get the the number of columns
        nCols = thetas.shape[ 0 ] // nRows

        # plot the prior for different theta
        fig, ax = subplots(nrows=nRows, ncols=nCols, sharex='all')
        for idx, thetai in enumerate(thetas):
            # compute the Gramm matrix
            K = kernel(x, x, thetai)
            # generate prior object
            prior = mv(mu, K, allow_singular=1)
            # sample from the prior
            samples = prior.rvs(5)
            for i in samples:
                ax.flatten()[ idx ].plot(x, i)
            ax.flatten() [ idx ].set_title(r'$\theta$ = {0}'.format(thetai))
            ax.flatten() [ idx ].set_xlabel('x')
            ax.flatten() [ idx ].set_ylabel('y')
        fig.suptitle(r'Samples from prior with different $\theta$ ')
        fig.tight_layout()
        fig.subplots_adjust(top=0.85)
        savefig('../Figures/1.5.png')
```

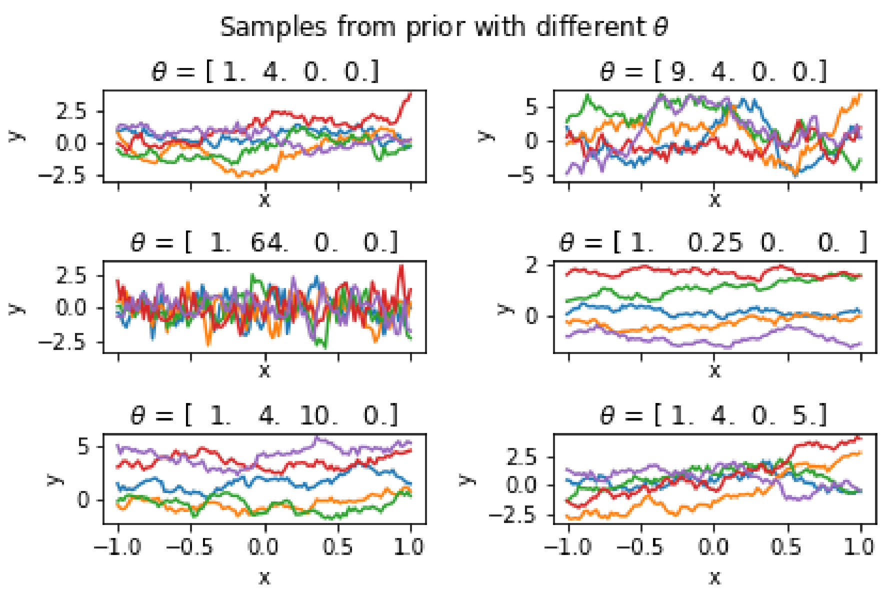


Figure 2: Results for 1.5. Samples drawn from the prior for different values of theta. An explanation for the different settings of θ is given in .

Bonus explanation

From the left column top two plots, and middle right we see that $\theta[1]$ regulates the amount of 'jitter' in the random process. From (3) we see that $\theta[1]$ weighs the euclidean distance between x, x'. Comparing the top row plots, we see that the non-linearity of the random process is controlled by increasing $\theta[0]$, i.e. (global) curvature of the lines. In other words, the gaussian part of the kernel favors clustered points, i.e. if the euclidean distance between them is small, it will tend to pull the process towards these points; creating these jitters we see here. The off-set of the random process is controlled by $\theta[2]$ (trivially) this can be seen in comparing the bottom and top plot in the left column. Finally, $\theta[3]$ controls the amount of linearity of change in the random process. This can be seen in bottom right plot, compared to the top row of plots. So in all we can summarize it as:

1. - Given the kernel function in (3):

   - $\theta[0]$ controls the non-linear component of the random process
   - $\theta[1]$ controls the 'jitterness' of the random process, i.e. weigh the euclidean distance in the random process
   - $\theta[2]$ controls the mean off set in the random process
   - $\theta[3]$ controls the linear component of change of the random process

2.6

In computing C we will use equation 6.62 from Bisschop.

```
In [7]: # q6
        # define training set; inputs, targets
        xTrain = array([[-.5, .2, .3 , -.1]]).T
        tTrain = array([[.5, -1, 3, -2.5]]).T
        KTrain = kernel(xTrain, xTrain, theta)
        beta   = 1
        C      = KTrain + 1/beta * eye(len(xTrain))
        print('C:\n ',C)

        # show the training points
        fig, ax = subplots()
        ax.scatter(xTrain, tTrain)
        ax.set_title('Scatter plot of training points')
        ax.set_ylabel('Targets')
        ax.set_xlabel('Input')
        savefig('../Figures/1.6.png')
```

```
C:
  [[ 3.25        1.60468809  1.52032005  1.86873075]
 [ 1.60468809  3.04        2.01122942  1.84070798]
 [ 1.52032005  2.01122942  3.09        1.78873075]
 [ 1.86873075  1.84070798  1.78873075  3.01      ]]
```

2.7

In order to compute the μ at x = 0, we will to compute the Gramm matrix using the training points and the new point (see Bisschop p.307); i.e. we compute K = K($\boldsymbol{X_n}, \boldsymbol{X_{N+1}}$), where $\boldsymbol{X_n}$ is the set $i = 1, ..., N$ and $X_{N+1} = 0$.

```
In [8]: xNew        = array([[0]])
        xTrainNew   = vstack((xTrain, xNew))
        c           = kernel(xNew, xNew, theta) + 1/beta
        k           = kernel(xTrain, xNew, theta)
        invC        = linalg.inv(C)
        # Bisschop 6.66
        mu_new      = k.T.dot(invC).dot(tTrain)
        # Bisschop 6.67
        sigma_new   = c - k.T.dot(invC).dot(k)
        print('mu at x = {0}:\n{1}'.format(xNew[0], mu_new) )
        print('sigma^2 at x = {0}:\n{1}'.format(xNew[0], sigma_new) )
```

```
mu at x = [0]:
[[-0.20721389]]
sigma^2 at x = [0]:
[[ 1.32520089]]
```

2.8

The mean of $p(t \mid (t)))$ will not go to zero as $x \to \pm\infty$. If x approaches $\pm\infty$ the gaussian part of (3) will go to zero but the linear part $(x^T x')$ will blow up to $\pm\infty$ (depending on the sign of x, x' (for more in depth explanation please see the bonus question above). Thus to make the kernel function to go to zero we would need to suppress $x^T x'$ by setting $\theta[3] = \theta[2] = 0$ (the bias, $\theta[2]$ will trivially cause non-zero if set to anything else than zero).

As an example, we take a large value for x and print the mean by setting $\theta[2] = \theta[3] = 0$.

```
In [9]: # take a very large number
        xNew         = array([[1e30]])
        xTrainNew    = vstack((xTrain, xNew))
        c            = kernel(xNew, xNew, theta) + 1/beta
        thetaEdit    = theta.copy()
        thetaEdit[-2:] = 0
        k            = kernel(xTrain, xNew, thetaEdit)
        invC         = linalg.inv(C)
        # Bisschop 6.66
        mu_new       = k.T.dot(invC).dot(tTrain)
        # Bisschop 6.67
        sigma_new    = c - k.T.dot(invC).dot(k)
        print('mu at x  = {0}:\n{1}\nand theta:\n {2}'\
              .format(xNew[0], mu_new, thetaEdit) )
```

```
mu at x  = [  1.00000000e+30]:
[[ 0.]]
and theta:
 [ 1.  1.  0.  0.]
```

Exercise 2 - Neural network regression

January 12, 2017

2.1

This is fairly trivial. Essentially, we make a meshgrid and use the object from scipy to compute the pdf over this meshgrid. Figure 1 shows the target distribution obtained from this process.

```
In [96]: import seaborn as sb
         from scipy.stats import multivariate_normal as mv
         from mpl_toolkits.mplot3d import Axes3D
         from pylab import *
         from numpy import *
         # q1
         # create the target distribution
         # sample at .1 interval
         tmp = arange(-2, 2, .1)
         x, y = meshgrid(tmp, tmp)

         mu = [0,0]
         sigma = eye(len(mu)) * 2/5
         dist = mv(mu, sigma)

         X = vstack((x.flatten(), y.flatten())).T
         Y = dist.pdf(X).reshape(x.shape)  * 3
         targets = array(Y.flatten(), ndmin = 2).T

         fig, ax = subplots(1, 1, subplot_kw = {'projection': '3d'})
         ax.plot_surface(x, y, targets.reshape(x.shape))
         ax.set_xlabel('$x_1$', labelpad = 20)
         ax.set_ylabel('$x_2$', labelpad = 20)
         ax.set_zlabel('pdf', labelpad =20)
         sb.set_context('poster')
         sb.set_style('white')
         fig.suptitle('Target distribution')
         savefig('../Figures/2.1.png')
         show()
```

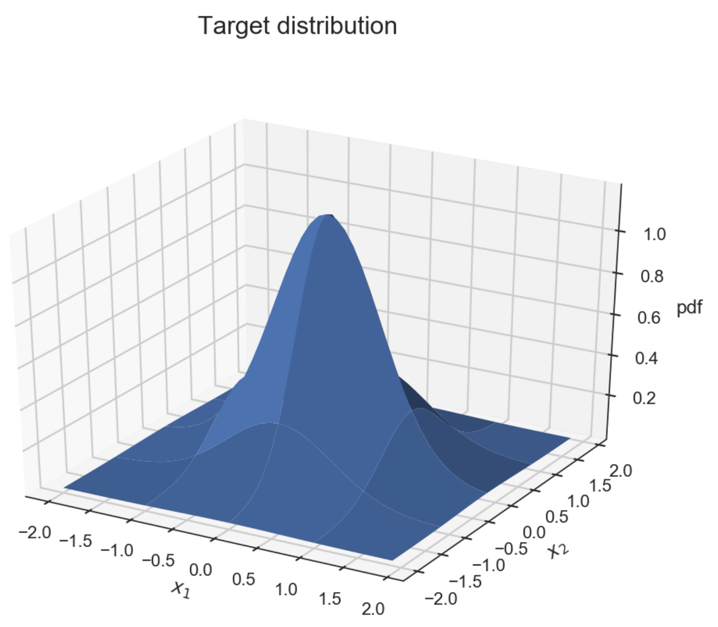


Figure 1: Target distribution for the MLP in the later exercises

2.1

The code provided below allows for plotting the progress over time. The naive output of the network is depicted in figure 2. The algorithm works by performing a forward pass, then feeding the error back through the network.

The forward pass uses bisschop eq 5.62, 5.63, 5.64. The backward pass uses the output to generate a delta (eq 5.65), and update the hidden layer weights for both the output connections and input connections (eq 5.67).

```
In [97]: class mlp(object):
             '''
             Multi-layered-pereptron
             K = output nodes
             M = hidden nodes
             Assumes the input data X is samples x feature dimension
             Returns:
                 prediction and error
             '''
             def __init__(self, X, t,\
                          eta = 1e-1,\
                          gamma = .0,\
                          M = 8,\
                          K = 1):
                 # learning rate / momentum rate
                 self.eta          = eta
                 self.gamma        = gamma
```

```
        # Layer dimensions; input, hidden, output
        self.D              = D =  X.shape[1] + 1
        self.M              = M
        self.K              = K
        # add bias node to input
        self.X              = hstack( (X, ones(( X.shape[0], 1 ) ) ) )
        self.targets        = t
        # weights; hidden and output
        wh                  = random.rand(D, M) - 1/2
        wo                  = random.rand(M, K) - 1/2

        self.layers         = [wh, wo]
        # activation functions:
        self.func           = lambda x: tanh(x)
        self.dfunc          = lambda x: 1 - x**2


    def forwardSingle(self, xi):
        ''' Performs a single forward pass in the network'''
        layerOutputs = [ []  for j in self.layers ]
        #forward pass
        a = xi.dot(self.layers[0])
        z = self.func(a)
        y = z.dot(self.layers[1])

        # save output
        layerOutputs[0].append(z);
        layerOutputs[1].append(y)
        return layerOutputs

    def backwardsSingle(self, ti, xi, forwardPass):
        '''Backprop + update of weights'''
        # prediction error
        dk  = forwardPass[-1][0] - ti
        squaredError = dk**2
        # compute hidden activation; note elementwise product!!
        dj  = \
        self.dfunc(forwardPass[0][0]) * (dk.dot(self.layers[-1].T))

        # update the weights
        E1  = forwardPass[0][0].T.dot(dk)
        E2  = xi.T.dot(dj)

        # update weights of layers
        self.layers[-1]  -= \
        self.eta * E1 + self.gamma * self.layers[-1]

        self.layers[0]   -= \
        self.eta * E2 + self.gamma * self.layers[0]
        return squaredError

    def train(self, num, plotProg = (False,)):
        #set up figure
        if plotProg[0]:
            fig, ax = subplots(subplot_kw = {'projection':'3d'})


        num   = int(num) # for scientific notation
        SSE   = zeros(num) # sum squared error
        preds = zeros((num, len(self.targets))) # predictions per run
        for iter in range(num):
            error = 0 # sum squared error
            for idx, (ti, xi) in enumerate(zip(self.targets, self.X)):
                xi = array(xi, ndmin = 2)

                forwardPass = self.forwardSingle(xi)
                error += self.backwardsSingle(ti, xi, forwardPass)
                preds[iter, idx] = forwardPass[-1][0]
            # plot progress
            if plotProg[0]:
                if not iter % plotProg[1]:
                    x, y = plotProg[2]
                    ax.cla() # ugly workaround
                    ax.plot_surface(x, y, preds[iter, :].reshape(x.shape))
                    ax.set_xlabel('$x_1$', labelpad = 20)
                    ax.set_ylabel('$x_2$', labelpad = 20)
                    ax.set_zlabel('pdf', labelpad =20)
                    ax.set_title('Cycle = {0}'.format( iter ))
                    pause(1e-10)
            SSE[iter] = .5 * error
        return SSE, preds

# perform a single forward pass and show the results
model = mlp(X, targets)
# perform a single pass
preds = array([\
              model.forwardSingle(\
              array(hstack( ( xi, 1) ),\
                    ndmin = 2))[-1]\
              for xi in X]).flatten()
# plot the results
fig, ax = subplots(subplot_kw  = {'projection': '3d'})
ax.scatter(x, y, preds.reshape(x.shape))
ax.set_xlabel('$x_1$', labelpad = 20)
ax.set_ylabel('$x_2$', labelpad = 20)
ax.set_zlabel('pdf', labelpad =20)
```

```
        ax.set_title('Network output without training')
        sb.set_context('poster')
        savefig('../Figures/2.2.png')
        show()
```

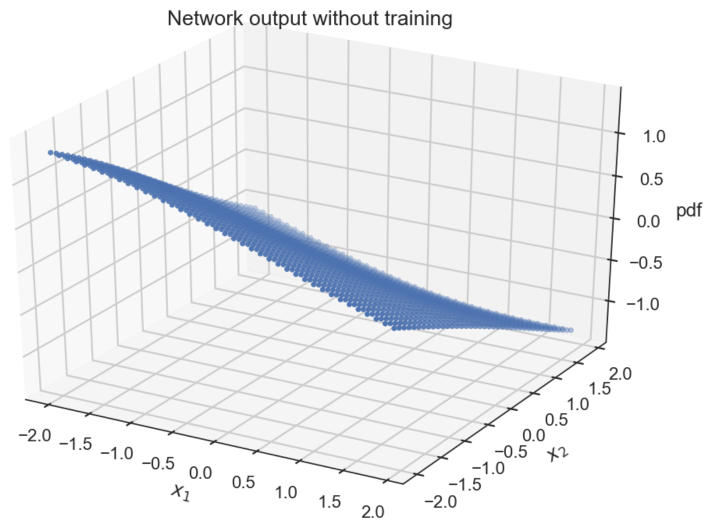


Figure 2: Results for 2.2; output of the network without any training. The output is a 2D plane since the weight have been sampled uniformly.

2.3

If one wants to please parse the correct parameter, i.e. plotProg = (True, interval, [x,y]). As summary of the training progress is depicted in figure 3. It is interesting to see that even after 1 full training cycle (top left plot in 3), we already can see a gaussian shape, it only get more and more refined as a function of training cycles. The MLP learns the most when its output is very wrong compared to the target. Hence, the 2D shee from figure 2, will be adjusted a lot after the first pass, but consecutive passes will learn less at the output is already farely close to the desired output. Please note that cylce = 0 is actually one fully completed training cycle as python starts indexing from 0. The zero training is shown in the previous question, i.e. figure 2.

```
In [98]: # run at atleast 500 cycles
         num = int(5e2) + 1
         model = mlp(X, targets)
         # train the model
         SSE, preds = model.train(num)

In [82]: # plot every 250 cycles
         cycleRange = arange(0,  num, num // 5)
```

```
# use at most 3 rows
nRows = 3
nCols = int(ceil(len(cycleRange)/nRows))
nCols = 2

fig, axes  = subplots(nRows,\
                      nCols,\
                      subplot_kw = {'projection': '3d'})


for axi, i in enumerate(cycleRange):
    ax = axes.flatten()[axi]
    ax.plot_surface(\
                    x,\
                    y,\
                    preds[i,:].reshape(x.shape),\
                    cstride = 1,\
                    rstride = 1)
    # formatting of plot
    ax.set_xlabel('$x_1$', labelpad = 20)
    ax.set_ylabel('$x_2$', labelpad = 20)
    ax.set_zlabel('pdf', labelpad =20)
    ax.set_title('cycles = {0}'.format(i))
    sb.set_style('white')
fig.delaxes(axes.flatten()[-1])
fig.suptitle('Output of MLP as a function of complete cycles')
subplots_adjust(top=0.8)
# fig.tight_layout()
savefig('../Figures/2.3.png')
show()
```

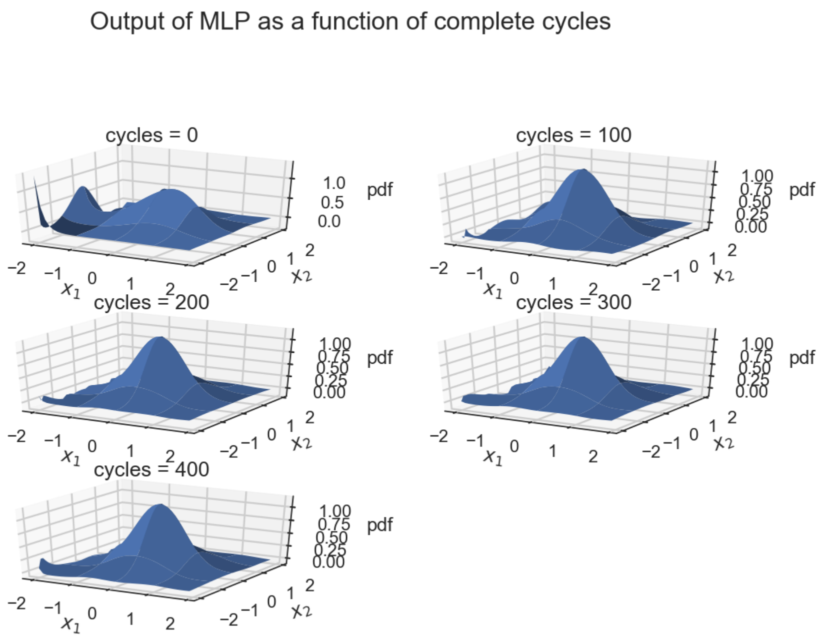


Figure 3: This figure shows the output of the network as a function of training cycles. Note that cycle = 0 is after 1 full pass of the network. Python starts indexing from 0. This if for the non-shuffled data.

2.4

Permute the indices and plot the sum squared error for the training. We expect the shuffled condition to have faster convergence. Results are shown in figure 4.

```
In [87]: # shuffle the indices
         idx = random.permutation(len(targets))

         # shuffle the data
         shuffleX = X[idx,:]
         # shuffle the targets as well (same indices)
         shuffleTargets = targets[idx, :]
         shuffleModel = mlp(shuffleX, shuffleTargets)
         shuffleSSE, shufflePreds = shuffleModel.train(num)
```

```
In [90]: # plot the results: compare with error from [3]
         fig, ax = subplots()
         ax.plot( range(num),SSE,\
                 range(num), shuffleSSE)

         ax.set_xlabel('Iterations')
         ax.set_ylabel('Sum squared error')
         ax.legend(['shuffled','non-shuffled'],loc = 0)
         fig.suptitle('Training error')
```

```
          savefig('../Figures/2.4.png')
          show()
```

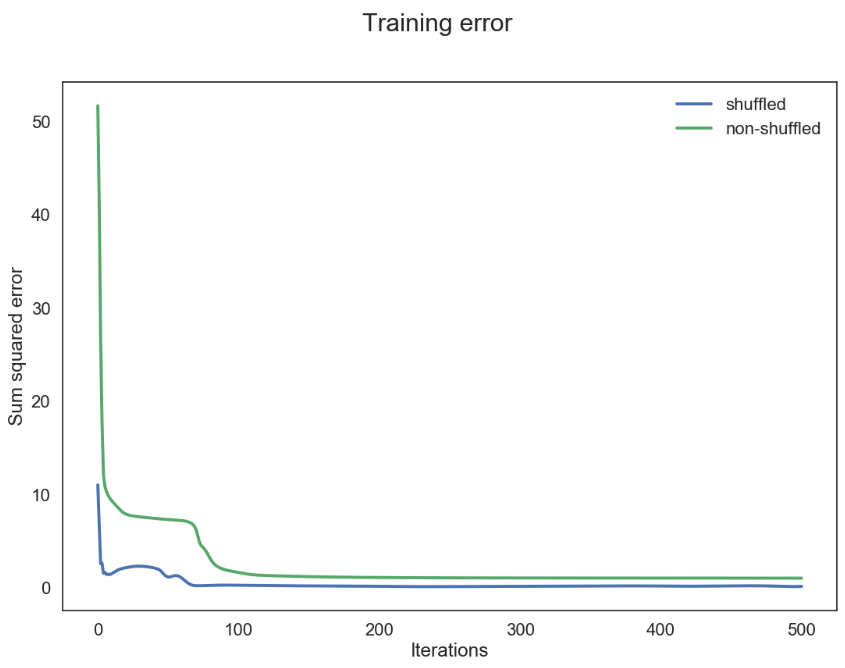


Figure 4: The sum squared error as a function of training cycles. Note a full cycle is after seeing all the training samples, weights are updated after seeing a datapoint. Noticeably, we see that the shuffled data is faster in converging, for explanation see text.

Since the grid is linearly spaced, this will mean that nearby points will yield the same gradient in the error. This 'local' correlation will yield that the algorithm will change the weights in similar direction for a while, hence shuffling the data removes this 'local' correlation structure, yielding more likely to move in the different directions, constraining the algorithm, yielding faster convergence. To summarize, the linearly spaced inputs will yield smaller gradients and thus slower learning as the network 'locally' overtrains. Randomizing inputs will yield larger gradients and thus the network will be better able to adjust the weights such that the prediction is generalized over the entire dataset.

2.5

Same as in section 1. The target distribution is shown in figure 5.

```
In [99]: # load the data
         X, Y, target = list(np.loadtxt('../Data/a017_NNpdfGaussMix.txt').T)
         tmp = int(np.sqrt(target.shape[0]))
         # convert in shape for it to be plottable
         x = X.reshape(tmp,tmp)

         y = Y.reshape(tmp, tmp)
         # stack to create input data
         X = np.vstack((X,Y)).T

         # target vector
         target = np.array(target, ndmin = 2).T

In [92]: # visualize the target distribution
         fig, ax = subplots(1,1, subplot_kw = {'projection': '3d'})
         ax.plot_surface(x, y, target.reshape(tmp, tmp))
         ax.set_xlabel('$x_1$', labelpad = 20)
         ax.set_ylabel('$x_2$', labelpad = 20)
         ax.set_zlabel('pdf', labelpad = 20)
         fig.suptitle('Target distribution')
         savefig('../Figures/2.5.png')
         show()
```

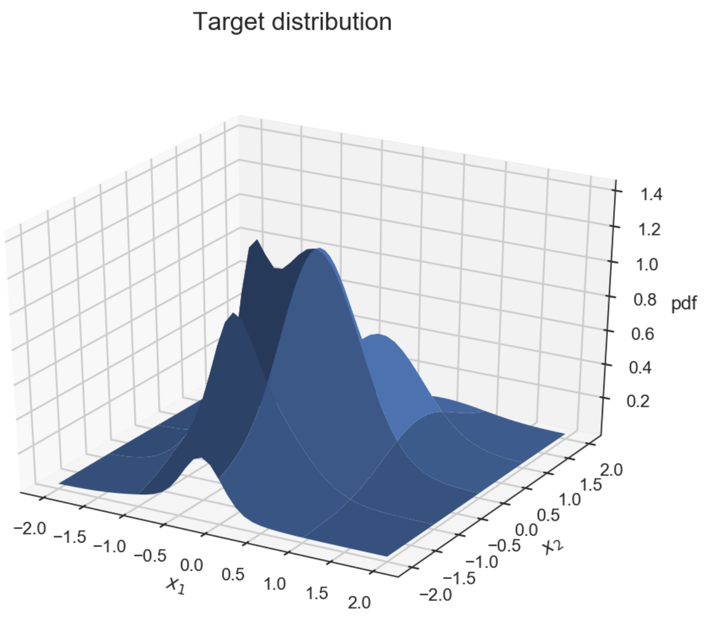


Figure 5: Target distribution loaded from the provided data files

2.6

The training error and final prediction are shown in figures 7, 6. Explanation is below the code block.

```
In [100]: #randomly permute indices
          idx = np.random.permutation(range(len(target)))
          # keep track of the changes
          shuffX = X[idx,:]; shuffTarget = target[idx]
```

```
            # run mlp with eta = .01
            model = mlp(shuffX, shuffTarget,\
                                  eta = 1e-2,\
                                  M = 40)

            # run for 2000 complete cylcles
            num = int(2e3)
            errors, preds = model.train(num = num)
            # map back to original space
            orgIdx = argsort(idx)
            # get final prediction
            finalPred = preds[-1, orgIdx]
```

```
In [94]: # plot the final prediction and the target distribution
         fig, ax = subplots(subplot_kw = {'projection': '3d'})
         ax.scatter(x, y, target.reshape(x.shape), label = 'target')
         ax.scatter(x, y, finalPred.reshape(x.shape), label = 'estimation')
         ax.set_xlabel('$x_1$', labelpad = 20)
         ax.set_ylabel('$x_2$', labelpad = 20)
         ax.set_zlabel('pdf', labelpad   = 20)
         ax.legend(loc = 0)
         fig.suptitle('Final prediction after {0} cycles'.format(num))
         savefig('../Figures/2.61.png')

         fig, ax = subplots()
         ax.plot(errors)
         ax.set_xlabel('iterations')
         ax.set_ylabel('Sum squared error')
         ax.set_title('Training error')
         savefig('../Figures/2.62.png')

         show()
```

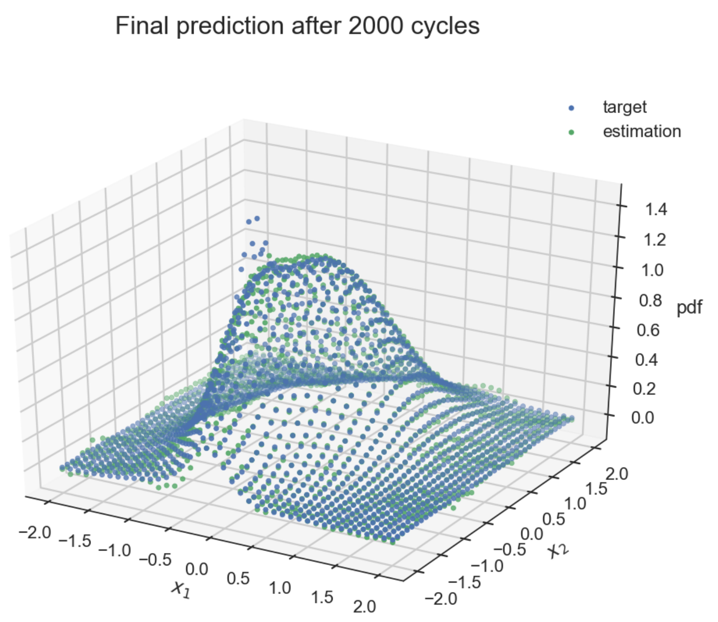


Figure 6: In green we gave the prediction generated from our network, for parameters please see the code block in this exercise. The performance of training is shown in figure 7. Note that the little hump is not captured, but the sum squared error remains low.

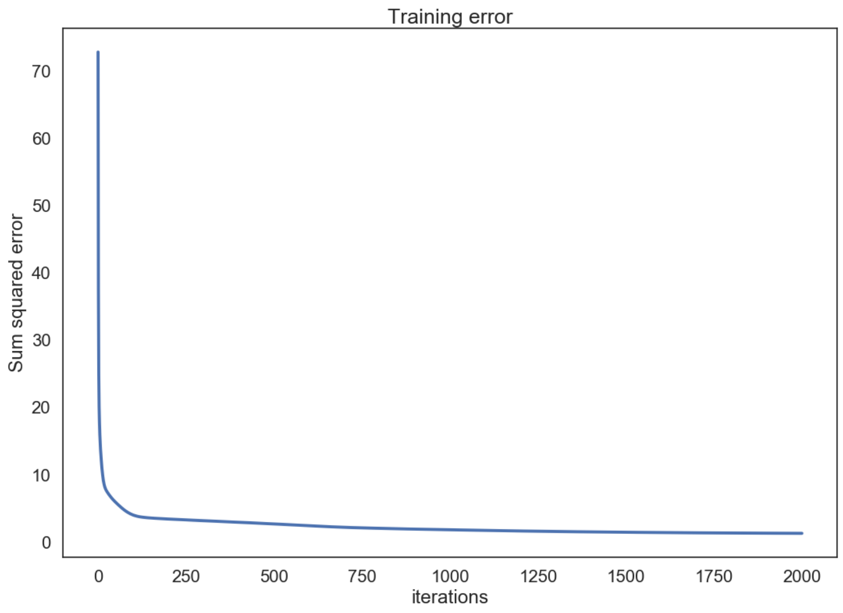


Figure 7: Sum squared error as a function of completed training cycles.

One might improve the performance by adding more hidden nodes to the network. The hidden nodes essentially represent the degrees of freedom in the model. Increasing the number of hidden nodes might increase the fit on a trainingset, however it will also increase the modelling noise (i.e overfitting). Improvements might also be found in presenting the inputs / targets in a different feature space. Multi-layered perceptrons are notorious for being sensitive to how the data is represented. Another might be instead of taking a global learning rate, is make it adaptive (see conjugate gradient descent, momentum etc).

2.7

We are using python hence netlab toolbox is not available to us. We opted for the neurolab toolbox which has a conjugate gradient method. The same parameters were used as in 7 to improve comparison. The final output for the neurolab is represented in figure 8. Comparisons of the training errors are displayed in figure 9. Note, that the conjugate gradient stopped after 25 cycles, the output stated that the algorithm didn't converge. Hence, we zoomed in on the plot. The error was lower than our implementation however.

```
In [101]: import neurolab as nl
          from scipy.optimize import fmin_ncg
          # inputs and targets
          inp = shuffX
          tar = shuffTarget

          # specify same network structure as we have
          # i.e. M = 40, D = 3, K = 1
          # this function takes min/max of input space

          # Specify activation functions;
          # input - to hidden  is hyperbolic tangent
          # hidden- to out is linear
          tranfs = [nl.trans.TanSig(), nl.trans.PureLin()]
          net = nl.net.newff(\
                              [ [np.min(X), np.max(X)] ] * inp.shape[1],\
                              [40, 1],\
                              transf= tranfs,\
                             )
          # use conjugate gradient as method
          net.trainf = nl.train.train_ncg # conjugate gradient
          # net.trainf = fmin_ncg
          net.errorf  = nl.error.SSE()     # same as above

          # init weight matrix between -.5,.5
          for l in net.layers:
              l.initf = nl.init.InitRand([-.5, .5], 'wb')
          net.init()

          # Train network; show output ever 500 iterations
          errorNL = net.train(inp, tar, epochs= num, show = 500)
```

```
           # Simulate network
           outNL = net.sim(inp)
           # sort back to original indices
           outNL = outNL[argsort(idx)]

Warning: Warning: CG iterations didn't converge.  The Hessian is not positive defi
         Current function value: 2.973533
         Iterations: 25
         Function evaluations: 36
         Gradient evaluations: 6893
         Hessian evaluations: 0
```

```
In [105]: # plot the performance versus our algorithm
           # plot the final prediction and the target distribution
           fig, ax = subplots(subplot_kw = {'projection': '3d'})
           ax.scatter(x, y,\
                      target.reshape(x.shape), label = 'target')
           ax.scatter(x, y,\
                      outNL.reshape(x.shape), label = 'estimation')

           ax.set_xlabel('$x_1$', labelpad = 20)
           ax.set_ylabel('$x_2$', labelpad = 20)
           ax.set_zlabel('pdf', labelpad = 20)
           ax.legend(loc = 0)
           fig.suptitle('Neurolab prediction')
           print(\
           'final SSE neruolab :\n {0}'\
           .format(errorNL[-1]))
           print(\
           'final SSE our alogorithm:\n{0}'.format(errors[-1]))
           savefig('../Figures/2.7.png')
           show()

           # plot training errors
           fig, ax = subplots()
           ax.plot(errorNL, label = 'conjugate gradient')
           ax.plot(errors, label = 'our MLP')
           ax.set_xlabel('Training cycles')
           ax.set_ylabel('Sum squared error')
           fig.suptitle('Conjugate gradient convergence')
           # savefig('../Figures/2.71.png')
           show()
```

```
final SSE neruolab :
 2.9735334308131867
```

```
final SSE our alogorithm:
1.3464563919674712
```

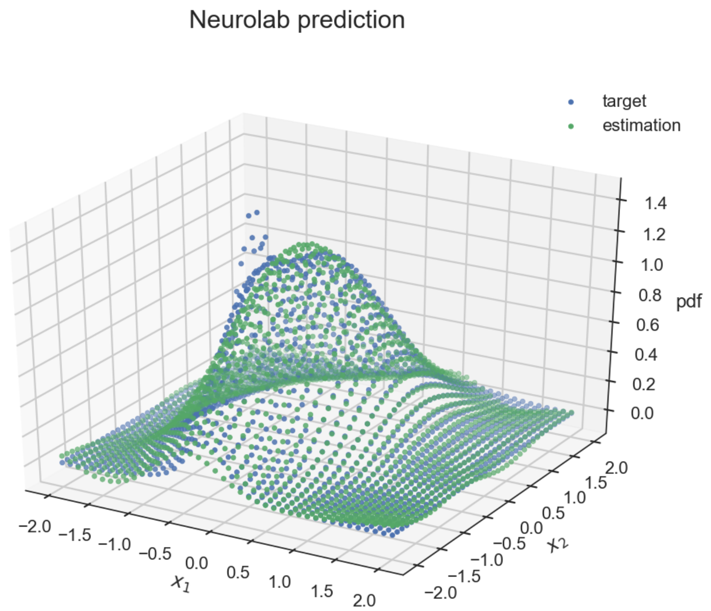


Figure 8: Output of conjugate gradient from neurolab toolbox.

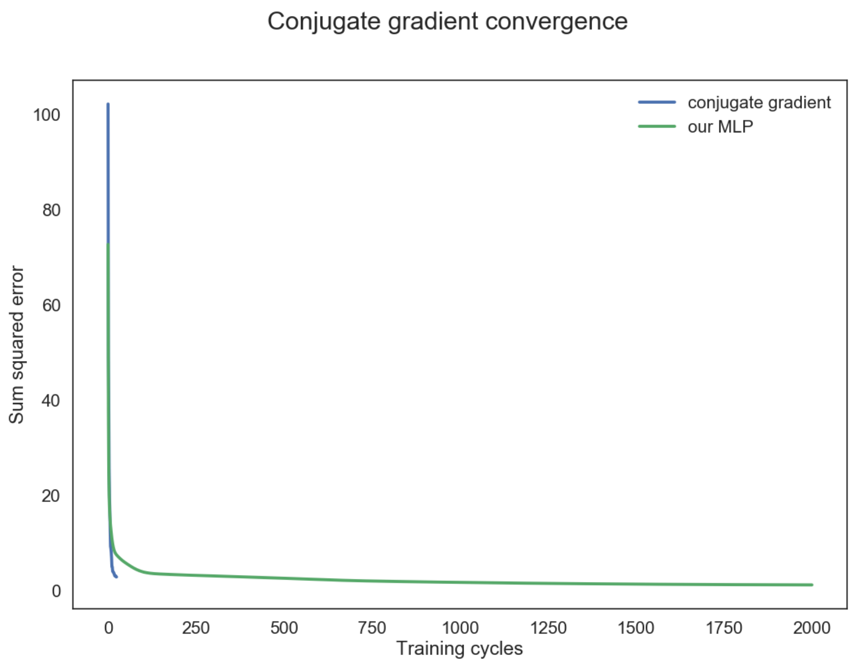


Figure 9: Training error of the neurolab implementation versus ours. Note that the rate of convergence if faster than our implementation. However, it did halt after 25 iterations due to a failure to converge. It uses the methods from scipy.

3 EM and Doping

3.1 Dataset visual inspection

It is always a good idea to get a feeling for the raw data itself. Suppose we were an expert witness, we would also have to present some graphs to show that we know the data. So, here we go: let us first look at 3D-scatterplots to see, that the first two features of each sample are correlated across the set. The other features are not that obviously correlated.

```
X = np.loadtxt('a011_mixdata.txt')
N, D = X.shape
# n = number of datapoints
# D = number of features
fig = plt.figure()
ax = fig.add_subplot(211, projection='3d')
ax.scatter(X[ :, 0 ], X[ :, 1 ], X[ :, 2 ], c='c', marker='o')
ax.set_xlabel('1st variable')
ax.set_xlabel('2nd variable')
ax.set_xlabel('3rd variable')
ax2 = fig.add_subplot(212, projection='3d')
ax2.scatter(X[ :, 0 ], X[ :, 2 ], X[ :, 3 ], c='c', marker='o')
ax2.set_xlabel('1st variable')
ax2.set_xlabel('2th variable')
ax2.set_xlabel('4th variable')
plt.show()
```

This will yield the following figures:

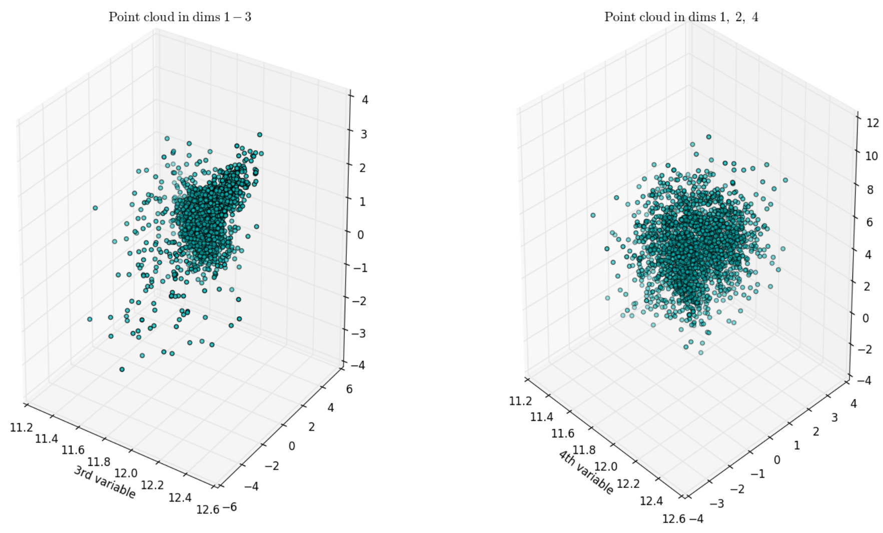


Figure 1: Point clouds do not show significant features, other than correlation between x_1, x_2.

Maybe histograms of the single dimensions tell us a bit more:

```
fig31 = plt.figure()
ax1 = fig31.add_subplot(221)
n1, bins1, patches1 = plt.hist(X[ :, 0 ], 25, normed=1, alpha=0.75)
plt.xlabel('frequency (normed)')
plt.ylabel('value')
plt.title(r'$\mathrm{Histogram\ in\ dim\ 1}$')
ax2 = fig31.add_subplot(222)
n2, bins2, patches2 = plt.hist(X[ :, 1 ], 25, normed=1, alpha=0.75)
plt.xlabel('frequency (normed)')
plt.ylabel('value')
plt.title(r'$\mathrm{Histogram\ in\ dim\ 2}$')
ax3 = fig31.add_subplot(223)
n3, bins3, patches3 = plt.hist(X[ :, 2 ], 25, normed=1, alpha=0.75)
plt.xlabel('frequency (normed)')
plt.ylabel('value')
plt.title(r'$\mathrm{Histogram\ in\ dim\ 3}$')
ax4 = fig31.add_subplot(224)
n4, bins4, patches4 = plt.hist(X[ :, 3 ], 25, normed=1, alpha=0.75)
plt.xlabel('frequency (normed)')
plt.ylabel('value')
plt.title(r'$\mathrm{Histogram\ in\ dim\ 4}$')
plt.show()
```

This will yield the following figures:

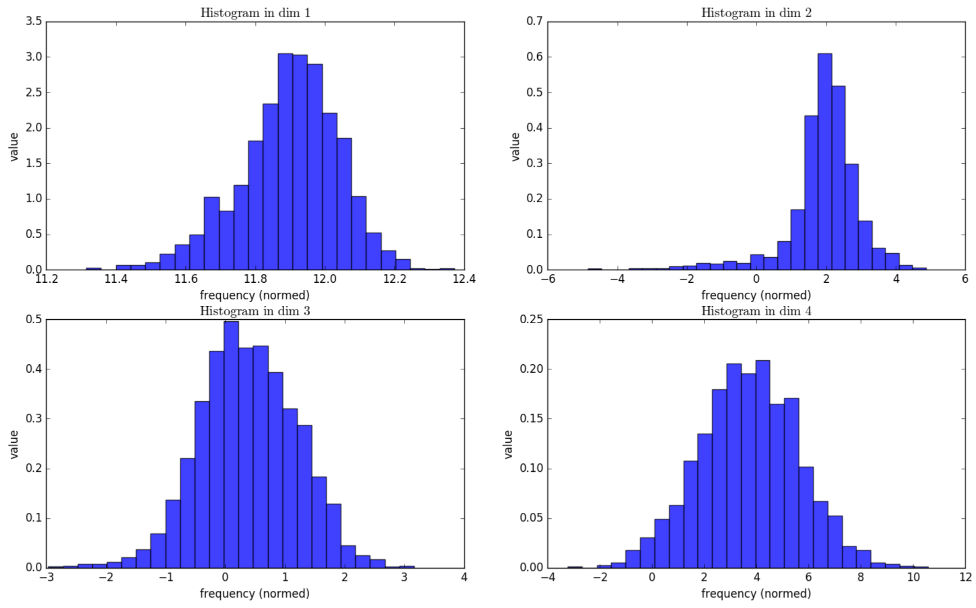


Figure 2: Histograms reveal a little more information.

Firstly, we notice that the distributions across dimensions 1 and 2 are both left-skewed. This does not imply a correlation, but it could be the one, we saw in the first graph.

Secondly, dimension 1 shows a little bump on the left side of the mean. This could be a mini-cluster.

3.2 Setting up the EM-algorithm

This is a pure coding exercise. All the requested features will become obvious in the next parts. The code of the EM-class is attached in the Appendix. I will only copy the relevant parts or parts from the executable script `Ex3.py` here. The EM-class is contained in `mixture_models.py`. It was originally based on the Gaussian mixture models from the scipy-package, but completely rewritten for this exercise and exercise 4. For convenience and readability, the general structure and some method names remain similar. Formulas implemented from CB06 are marked with the respective number. To make the review easier, I will point to where the specific requested features can be found in the code.

- Initialisation variable setup, i.e. means, weights and covariances:

```
n_iterations = 100
K = 2
# initialisation
init_means = np.repeat(np.mean(X, axis=0), K,
axis=0).reshape(K, D)
init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
init_weights = np.ones(K, dtype=float) / K
init_covars = np.zeros((K, D, D))
for k in np.arange(K):
    # the initialisation cannot lead to singularity.
    Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
    init_covars[ k, :, : ] = np.diag(Sigma_k)
init_covars = init_covars[ :, :, : ]
```

- After the initialisation of the class instance, the number of iterations of the EM-steps are dynamically increased until reaching 100:

```
loglikelihoods = np.zeros(n_iterations)
criterions = np.zeros(n_iterations)
convergence_print = False
for i in np.arange(1, n_iterations):
    gmm = mixture_models.MixtureModel(
            n_components=K,
            means_init=init_means,
            weights_init=init_weights,
            covars_init=init_covars,
            random_state=np.random.seed(1),
            n_iter=i)
    gmm = gmm.fit(X)
    if gmm.converged_ and not convergence_print:
        print('converged at step {0}'.format(i))
```

```
                convergence_print = True
            loglikelihoods[ i ] = np.sum(gmm.score_samples(X)[ 0 ])
            criterions[ i ] = gmm.bic(X)

        # The final data labels:
        labels = gmm.score_samples(X)[ 1 ].argmax(axis=1)
```

This already includes an example run in the line containing `gmm = gmm.fit(X)`, which is part of subsection 3.3.

- The plots are displayed here for the initial run, required in 3.3.

We will get the following figures:

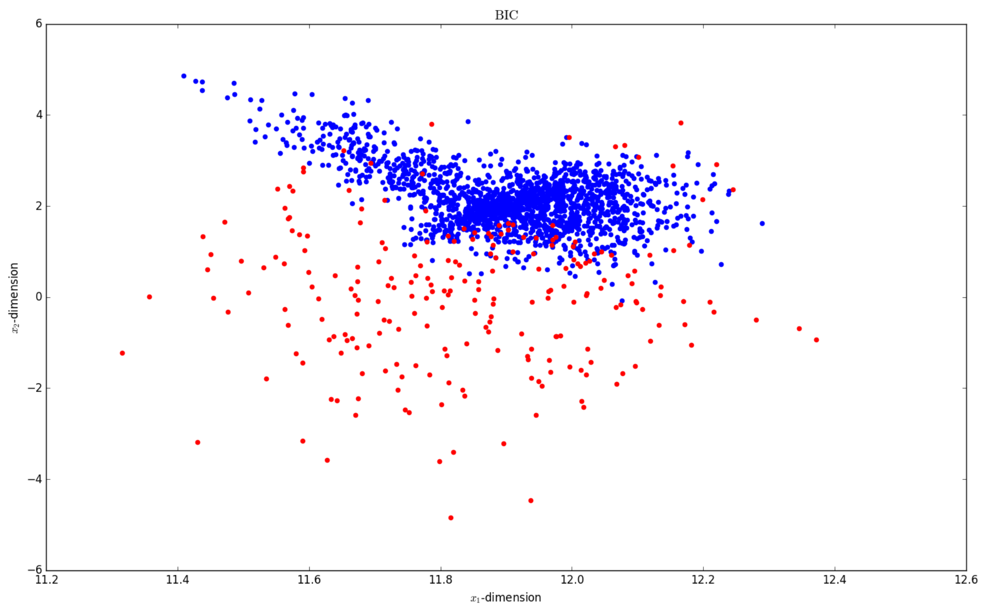


Figure 3: It is obvious that one cluster is significantly bigger than the other. This may either be due to the 1-in-5 rumors or an unfortunate RNG initialisation. As it is unclear which cluster is which, there are no labels designated.

Now have a look at the course of the classification procedure: We will get the following figures:

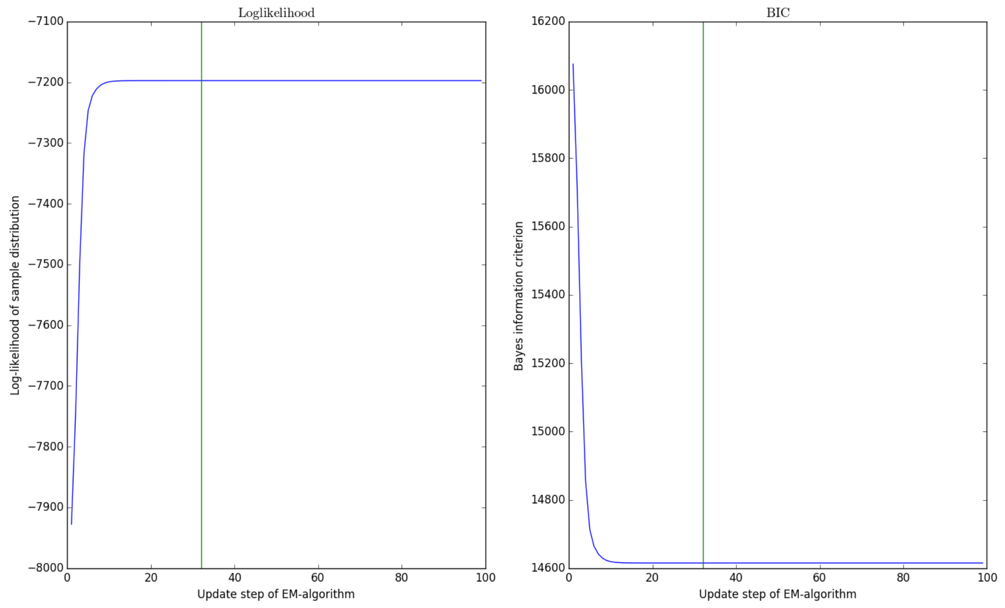


Figure 4: Depicted are the loglikelihood (left) and (due to its popularity for simple model selection) the Bayes information criterion (BIC, right). Obviously, we have a quick, close to exponential convergence. The EM-algorithm converges at step 32, marked in green, with an change in loglikelihood smaller than 10^{-8}.

3.3 Fitting Gaussian mixture models with EM

The $K = 2$-case has already been described above (see figure 4).

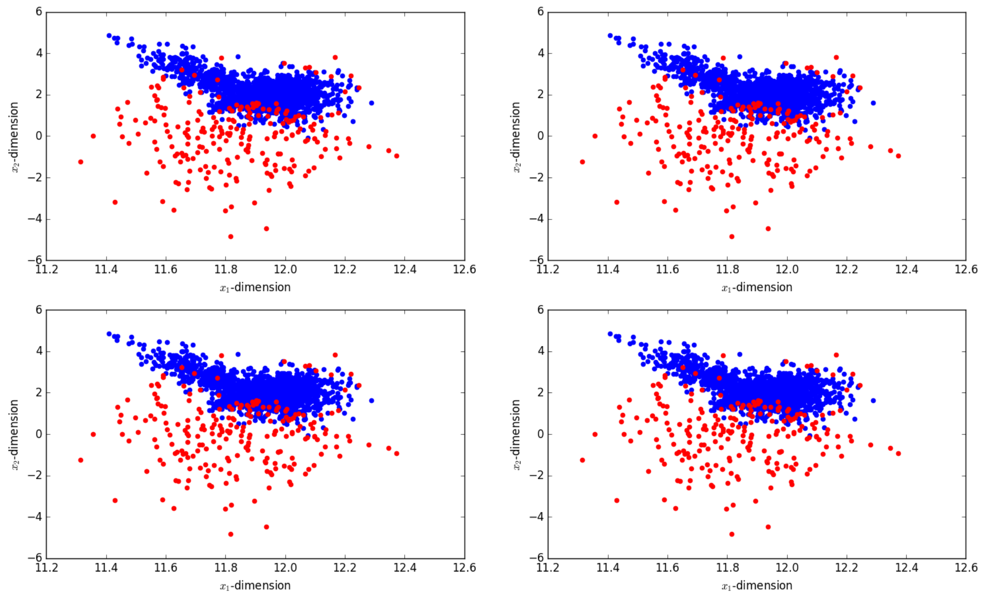


Figure 5: The four plots show classifications from different random starting points. The classifications are similar or identical but convergence is reached unequally fast.

If we look at the clustering from different starting points, we can not visually differentiate (see figure 5), however, when looking at the BIC-curves, we can see, that the algorithm converges more or less equally quickly. In fact, all the graphs look like figure 4 when not zooming in extremely. The earliest convergence measured in 8 randomised starting points was in 19 steps, the longest took 33 steps. This is marginal, considering there is only a change $< 10^{-3}$ after the 6th step. Of course, this is not yet enough to reliably infer on the distribution.

The correlation matrix of the components averaged over the different RNG initialisations is obtained the following way:

```
# correlation coefficients
corrcoeffs = np.zeros((K, randomisations))
for l in np.arange(randomisations):
    for k in np.arange(K):
        submat = X[ labels2[ :, l ] == k, : ]
        corrcoeffs[ k, l ] = \
            np.corrcoef(submat[ :, 0 ], submat[ :, 1 ])[ 0, 1 ]
corrcoeffs = np.mean(corrcoeffs, axis=1)
print(corrcoeffs)
```

This gives:

$$R^{(K=2)} = \begin{pmatrix} -0.38041957 & -0.06159985 \end{pmatrix} \tag{1}$$

We can see that there is a medaite correlation between dimensions x_1 and x_2. We will thus continue and investigate further instances.

3.4 Gaussian mixture models with 3 components

If we increase $K = 3$ or $K = 4$ and otherwise keep the code from section 3.3, we will obtain the following clustering (examples):

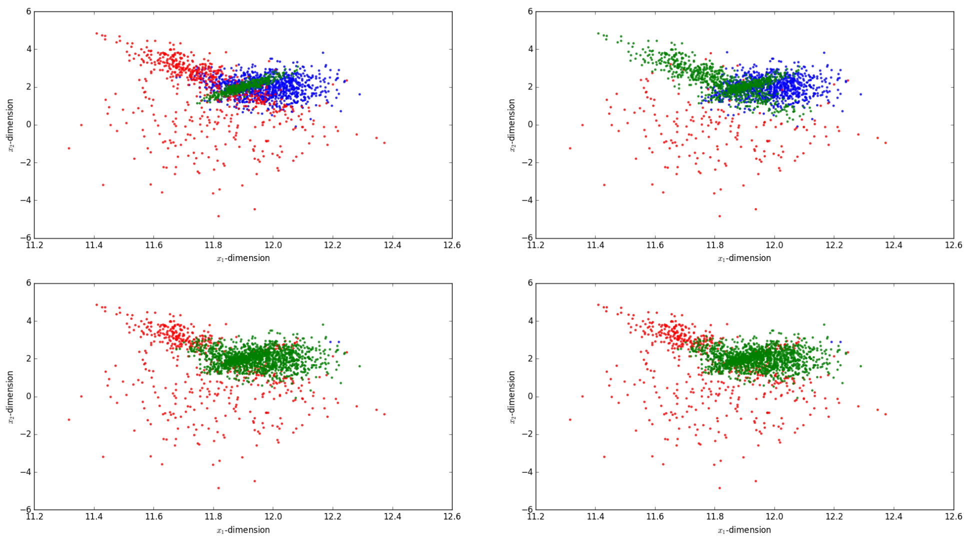


Figure 6: From left to right, top to bottom, the best, second-best, third-best and worst classification with $K = 3$ components measured by BIC, colors corresponding to curves in 8.

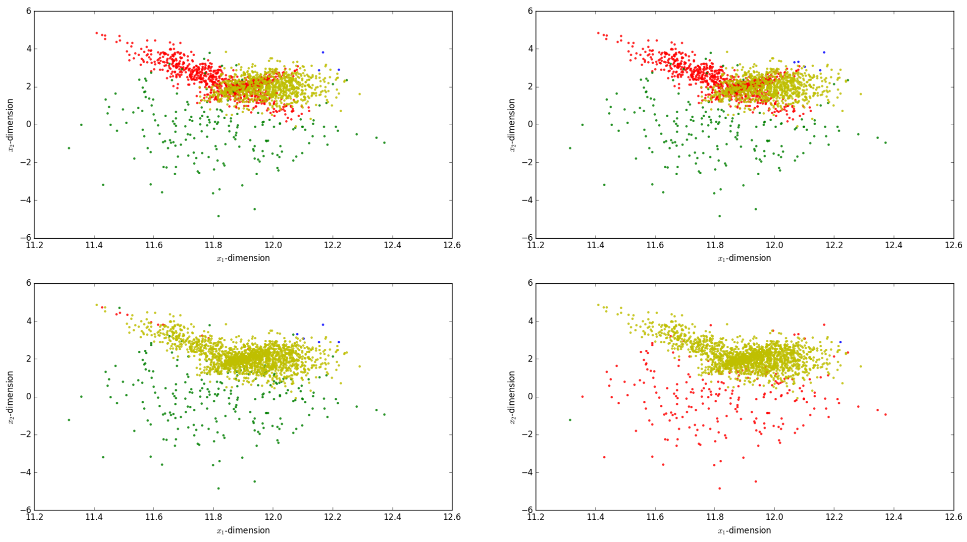


Figure 7: From left to right, top to bottom, the best, second-best, third-best and worst classification with $K = 4$ components measured by BIC, colors corresponding to curves in 9.

For the mean across eight initialisations, we get the following correlations between the first two components:

$$R^{(K=3)} = \begin{pmatrix} -0.4350281 & -0.29754575 & -0.0639132 \end{pmatrix} \tag{2}$$

and

$$R^{(K=4)} = \begin{pmatrix} 0.30244767 & -0.59228445 & -0.20562716 & -0.03762572 \end{pmatrix} \tag{3}$$

We can now see that the first two variables in the first components of the $K = 3$ and $K = 4$ solutions show intermediate correlations and the second component in the $K = 4$ solution has the strong correlation between x_1 and x_2, but the solution has a significantly higher BIC than the $K = 3$ solution:

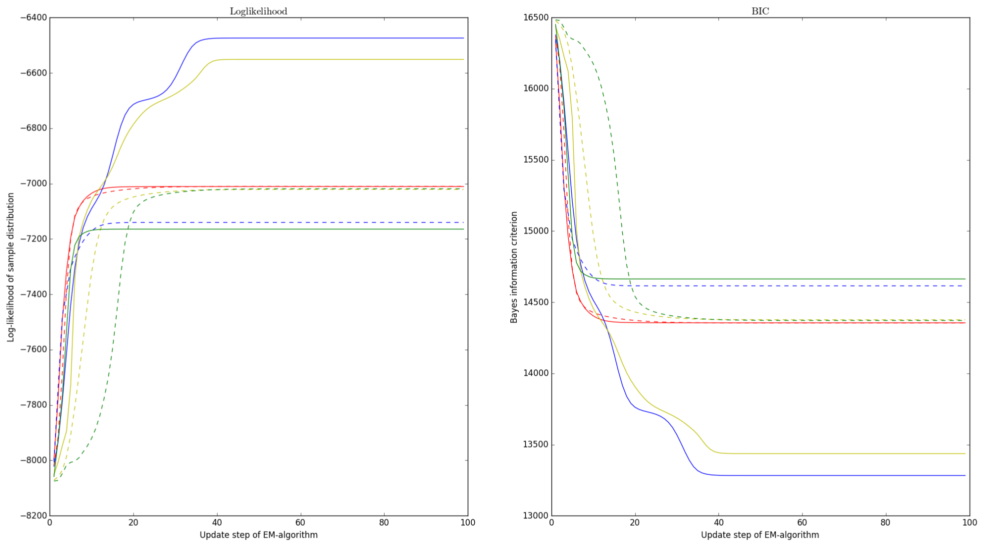


Figure 8: Criteria for 8 random starting points for means and covariance matrices for $K = 3$ clusters. The upper cluster of curves in the BIC plot apparently falls into a local minimum. The blue and yellow curves avoid this one and one additional minimum around $\mathrm{BIC} = 13700$
. The classification scatter plots for the blue, yellow, red and green curves are shown in figure 6.

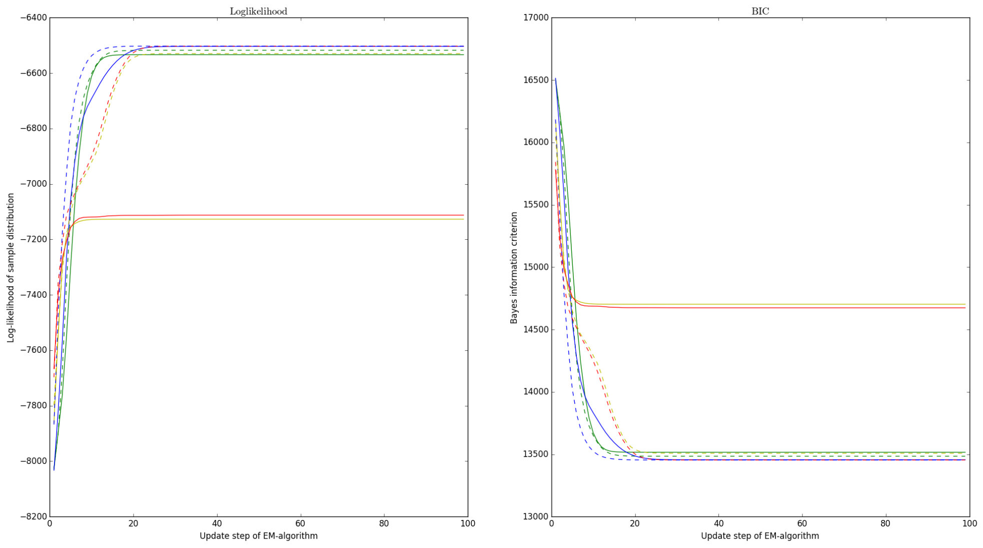


Figure 9: Criteria for 8 random starting points for means and covariance matrices for $K = 3$ clusters. There are again local minima, which are avoided by some instances, but the BIC for the optimal solution is higher than the one of case blue in figure 8, so the 3-Cluster solution is to be preferred.

Our favourite candidates so far are the blue and yellow instances from the $K = 3$ approach. The specific correlations for these cases are

$$R_{12}^{\text{blue}} = \begin{pmatrix}-0.07132327 & -0.40488436 & 0.91446507\end{pmatrix} \tag{4}$$

and

$$R_{12}^{\text{yellow}} = \left(0.12459622, -0.12769156, -0.72187397\right) \tag{5}$$

This fits nicely, as we can then say that the best solution is blue with the third component having the high correlation between variables x_1 and x_2 and thus containing the samples with substance X. The number of points mapped to this cluster is `np.sum(labels3 == 2)` = 406, so 1 in 5 is indeed appropriate! Cluster 1 has 974 cases (which we assume to be okay) and cluster to contains 620 cases, which are undecided.

3.5 Prediction for new samples

Now, let's do something dangerous, because you should never infer from the statistics on the single case...

```
# Here are our possible culprits:
newsamples = np.array([ [ 11.85, 2.2, 0.5, 4.0 ],
                        [ 11.95, 3.1, 0.0, 1.0 ],
                        [ 12.00, 2.5, 0.0, 2.0 ],
                        [ 12.00, 3.0, 1.0, 6.3 ] ])
alpha = gmm2.predict_proba(newsamples)
```

We get:

$$\alpha = \begin{pmatrix} \mathbf{.25357808e-01} & 7.43299250e-02 & 3.12267174e-04 \\ 2.67197658e-07 & \mathbf{9.99999715e-01} & 1.75890986e-08 \\ 8.94732884e-05 & 7.57923921e-03 & \mathbf{9.92331288e-01} \\ \mathbf{9.76578326e-01} & 2.34216745e-02 & 5.81372639e-13 \end{pmatrix} \tag{6}$$

From the boldfaced numbers, we deduce that sample C is the sample generated after consumption of substance X and sample B has been tampered with.

```
"""
This is a default script that should be adapted to the respective purpose.
"""

import numpy as np
from scipy import linalg
import os
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numbers
import matplotlib as mpl
import matplotlib.cm as cmx
import mixture_models
from itertools import combinations

cm = mpl.colors.ListedColormap('YlGnBu')
seashore = cm = plt.get_cmap('YlGnBu')
scalarMap = cmx.ScalarMappable(cmap=seashore)
plt.clf()

if __name__ == '__main__':
    # let us first load the data:
    os.chdir('../Data/')
    X = np.loadtxt('a011_mixdata.txt')
    os.chdir('../Code/')
    N, D = X.shape
    # n = number of datapoints
    # D = number of features

    ex31 = False
    ex32 = False
    ex33 = False
    ex34 = False
    ex35 = True

    if ex31:
        """
        Exercise 3.1
        """
        fig = plt.figure()
        ax = fig.add_subplot(121, projection='3d')
        ax.scatter(X[ :, 0 ], X[ :, 1 ], X[ :, 2 ], c='c', marker='o',  s=6, alpha=
        ax.set_xlabel('1st variable')
        ax.set_ylabel('2nd variable')
        ax.set_zlabel('3rd variable')
        plt.title(r'$\mathrm{Point\ cloud\ in\ dims\ 1-3}$')
        ax2 = fig.add_subplot(122, projection='3d')
        ax2.scatter(X[ :, 0 ], X[ :, 2 ], X[ :, 3 ], c='c', marker='o',  s=6, alpha
        ax2.set_xlabel('1st variable')
        ax2.set_ylabel('2th variable')
        ax2.set_zlabel('4th variable')
        plt.title(r'$\mathrm{Point\ cloud\ in\ dims\ 1,\ 2,\ 4}$')

        fig31 = plt.figure()
        ax1 = fig31.add_subplot(221)
        n1, bins1, patches1 = plt.hist(X[ :, 0 ], 25, normed=1, alpha=0.75)
        plt.xlabel('frequency (normed)')
        plt.ylabel('value')
        plt.title(r'$\mathrm{Histogram\ in\ dim\ 1}$')
        ax2 = fig31.add_subplot(222)
        n2, bins2, patches2 = plt.hist(X[ :, 1 ], 25, normed=1, alpha=0.75)
        plt.xlabel('frequency (normed)')
        plt.ylabel('value')
        plt.title(r'$\mathrm{Histogram\ in\ dim\ 2}$')
        ax3 = fig31.add_subplot(223)
        n3, bins3, patches3 = plt.hist(X[ :, 2 ], 25, normed=1, alpha=0.75)
        plt.xlabel('frequency (normed)')
        plt.ylabel('value')
        plt.title(r'$\mathrm{Histogram\ in\ dim\ 3}$')
        ax4 = fig31.add_subplot(224)
        n4, bins4, patches4 = plt.hist(X[ :, 3 ], 25, normed=1, alpha=0.75)
        plt.xlabel('frequency (normed)')
        plt.ylabel('value')
        plt.title(r'$\mathrm{Histogram\ in\ dim\ 4}$')
        plt.show()

    if ex32:
        """
        Exercise 3.2
        """
        # set up some stuff for the gaussian mixture models
        n_iterations = 100
        K = 2
        np.random.seed(1)
        # initialisation
        init_means = np.repeat(np.mean(X, axis=0), K, axis=0).reshape(K, D)
        init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
        init_weights = np.ones(K, dtype=float) / K
        init_covars = np.zeros((K, D, D))
        for k in np.arange(K):
            # the initialisation cannot lead to singularity.
            Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
            init_covars[ k, :, : ] = np.diag(Sigma_k)
        init_covars = init_covars[ :, :, : ]

        # Fit a Gaussian mixture with EM using five components
```

```
loglikelihoods = np.zeros(n_iterations)
criterions = np.zeros(n_iterations)
convergence_print = False
for i in np.arange(1, n_iterations):
    gmm = mixture_models.MixtureModel(
            n_components=K,
            means_init=init_means,
            weights_init=init_weights,
            covars_init=init_covars,
            random_state=np.random.seed(1),
            n_iter=i)
    gmm = gmm.fit(X)
    if gmm.converged_ and not convergence_print:
        print('converged at step {0}'.format(i))
        convergence_print = True
    loglikelihoods[ i ] = np.sum(gmm.score_samples(X)[ 0 ])
    criterions[ i ] = gmm.bic(X)

# The final data labels:
labels = gmm.score_samples(X)[ 1 ].argmax(axis=1)

# let's plot the change in the loglikelihood over iterations
fig321 = plt.figure()
ax1 = fig321.add_subplot(121)
ax1.plot(np.arange(1, n_iterations), loglikelihoods[ 1: ])
plt.axvline(x=32, color='g')
ax1.set_xlabel('Update step of EM-algorithm')
ax1.set_ylabel('Log-likelihood of sample distribution')
plt.title(r'$\mathrm{Loglikelihood}$')
ax2 = fig321.add_subplot(122)
ax2.plot(np.arange(1, n_iterations), criterions[ 1: ])
plt.axvline(x=32, color='g')
ax2.set_xlabel('Update step of EM-algorithm')
ax2.set_ylabel('Bayes information criterion')
plt.title(r'$\mathrm{BIC}$')

# Now plot the requested colour-coded first two variables
fig322 = plt.figure()
ax3 = fig322.add_subplot(111)
set0 = X[ labels == 0, : ]
set1 = X[ labels == 1, : ]
ax3.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax3.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax3.set_xlabel(r'$x_1$-dimension')
ax3.set_ylabel(r'$x_2$-dimension')
plt.title(r'$\mathrm{BIC}$')
plt.show()
```

```
if ex33:
    """
    Exercise 3.3
    """
    # The above version was a test run. Now some different initialisations:
    randomisations = 8
    n_iterations = 50
    loglikelihoods2 = np.zeros((n_iterations, randomisations))
    criterions2 = np.zeros((n_iterations, randomisations))
    labels2 = np.zeros((X.shape[ 0 ], randomisations))
    K = 2

    for j in np.arange(randomisations):
        np.random.seed(j)

        # initialisation
        init_means = np.repeat(np.mean(X, axis=0), K, axis=0).reshape(K, D)
        init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
        init_weights = np.ones(K, dtype=float) / K
        init_covars = np.zeros((K, D, D))
        for k in np.arange(K):
            # the initialisation cannot lead to singularity.
            Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
            init_covars[ k, :, : ] = np.diag(Sigma_k)
        init_covars = init_covars[ :, :, : ]

        # Fit a Gaussian mixture with EM using five components
        convergence_print = False
        for i in np.arange(0, n_iterations):
            # the data get quite big so we overwrite it every time
            gmm2 = mixture_models.MixtureModel(
                    n_components=K, n_iter=i,
                    means_init=init_means, weights_init=init_weights,
                    covars_init=init_covars,
                    random_state=np.random.seed(randomisations)
            )
            gmm2 = gmm2.fit(X)
            if gmm2.converged_ and not convergence_print:
                print('converged at step {0}'.format(i))
                convergence_print = True
            loglikelihoods2[ i, : ] = np.sum(gmm2.score_samples(X)[ 0 ])
            criterions2[ i, : ] = gmm2.bic(X)
        if not gmm2.converged_:
            print('no convergence in trial {0}'.format(j))
        # The final data labels:
        labels2[ :, j ] = gmm2.score_samples(X)[ 1 ].argmax(axis=1)

    # let's plot the change in the loglikelihood over iterations
```

```
fig331 = plt.figure()
ax1 = fig331.add_subplot(121)
ax1.plot(np.arange(1, n_iterations), loglikelihoods2[ 1:, 1 ], 'r',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 2 ], 'g',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 3 ], 'b',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 4 ], 'y',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 5 ], 'r--',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 6 ], 'g--',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 7 ], 'b--',
         np.arange(1, n_iterations), loglikelihoods2[ 1:, 0 ], 'y--'
         )
ax1.set_xlabel('Update step of EM-algorithm')
ax1.set_ylabel('Log-likelihood of sample distribution')
plt.title(r'$\mathrm{Loglikelihood}$')
ax2 = fig331.add_subplot(122)
ax2.plot(np.arange(1, n_iterations), criterions2[ 1:, 1 ], 'r',
         np.arange(1, n_iterations), criterions2[ 1:, 2 ], 'g',
         np.arange(1, n_iterations), criterions2[ 1:, 3 ], 'b',
         np.arange(1, n_iterations), criterions2[ 1:, 4 ], 'y',
         np.arange(1, n_iterations), criterions2[ 1:, 5 ], 'r--',
         np.arange(1, n_iterations), criterions2[ 1:, 6 ], 'g--',
         np.arange(1, n_iterations), criterions2[ 1:, 7 ], 'b--',
         np.arange(1, n_iterations), criterions2[ 1:, 0 ], 'y--'
         )
ax2.set_xlabel('Update step of EM-algorithm')
ax2.set_ylabel('Bayes information criterion')
plt.title(r'$\mathrm{BIC}$')

# scatter plots
fig332 = plt.figure()
ax0 = fig332.add_subplot(221)
set0 = X[ labels2[ :, 0 ] == 0, : ]
set1 = X[ labels2[ :, 0 ] == 1, : ]
ax0.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax0.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax0.set_xlabel(r'$x_1$-dimension')
ax0.set_ylabel(r'$x_2$-dimension')
ax1 = fig332.add_subplot(222)
set0 = X[ labels2[ :, 2 ] == 0, : ]
set1 = X[ labels2[ :, 2 ] == 1, : ]
ax1.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax1.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax1.set_xlabel(r'$x_1$-dimension')
ax1.set_ylabel(r'$x_2$-dimension')
ax2 = fig332.add_subplot(223)
set0 = X[ labels2[ :, 4 ] == 0, : ]
set1 = X[ labels2[ :, 4 ] == 1, : ]
ax2.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
```

```
    ax2.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
    ax2.set_xlabel(r'$x_1$-dimension')
    ax2.set_ylabel(r'$x_2$-dimension')
    ax3 = fig332.add_subplot(224)
    set0 = X[ labels2[ :, 6 ] == 0, : ]
    set1 = X[ labels2[ :, 6 ] == 1, : ]
    ax3.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
    ax3.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
    ax3.set_xlabel(r'$x_1$-dimension')
    ax3.set_ylabel(r'$x_2$-dimension')

    # correlation coefficients
    corrcoeffs2 = np.zeros((K, randomisations))
    for l in np.arange(randomisations):
        for k in np.arange(K):
            submat = X[ labels2[ :, l ] == k, : ]
            corrcoeffs2[ k, l ] = \
                np.corrcoef(submat[ :, 0 ], submat[ :, 1 ])[ 0, 1 ]
    corrcoeffs_mean2 = np.nanmean(corrcoeffs2, axis=1)
    print(corrcoeffs_mean2)

if ex34:
    """
    Exercise 3.4
    """
    # We move from 2 to 3 Gaussian components
    randomisations = 8
    n_iterations = 100
    loglikelihoods3 = np.zeros((n_iterations, randomisations))
    criterions3 = np.zeros((n_iterations, randomisations))
    labels3 = np.zeros((X.shape[ 0 ], randomisations))
    K = 3
    for j in np.arange(randomisations):
        np.random.seed(j)

        # initialisation
        init_means = np.repeat(np.mean(X, axis=0), K, axis=0).reshape(K, D)
        init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
        init_weights = np.ones(K, dtype=float) / K
        init_covars = np.zeros((K, D, D))
        for k in np.arange(K):
            # the initialisation cannot lead to singularity.
            Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
            init_covars[ k, :, : ] = np.diag(Sigma_k)
        init_covars = init_covars[ :, :, : ]

        # Fit a Gaussian mixture with EM using five components
        convergence_print = False
        for i in np.arange(0, n_iterations):
            # the data get quite big so we overwrite it every time
            gmm3 = mixture_models.MixtureModel(
                    n_components=K, n_iter=i,
                    means_init=init_means, weights_init=init_weights,
                    covars_init=init_covars,
                    random_state=np.random.seed(randomisations)
            )
            gmm3 = gmm3.fit(X)
            if gmm3.converged_ and not convergence_print:
                print('converged at step {0}'.format(i))
                convergence_print = True
            loglikelihoods3[ i, j ] = np.sum(gmm3.score_samples(X)[ 0 ])
            criterions3[ i, j ] = gmm3.bic(X)
        if not gmm3.converged_:
            print('no convergence in trial {0}'.format(j))
        # The final data labels:
        labels3[ :, j ] = gmm3.score_samples(X)[ 1 ].argmax(axis=1)

    # correlation coefficients
    corrcoeffs3 = np.zeros((K, randomisations))
    for l in np.arange(randomisations):
        for k in np.arange(K):
            submat = X[ labels3[ :, l ] == k, : ]
            corrcoeffs3[ k, l ] = \
                np.corrcoef(submat[ :, 0 ], submat[ :, 1 ])[ 0, 1 ]
    corrcoeffs_mean3 = np.nanmean(corrcoeffs3, axis=1)
    print(corrcoeffs_mean3)

    # plotting
    # scatter plots
    fig342 = plt.figure()
    ax0 = fig342.add_subplot(221)
    idx = 3
    set0 = X[ labels3[ :, idx ] == 0, : ]
    set1 = X[ labels3[ :, idx ] == 1, : ]
    set2 = X[ labels3[ :, idx ] == 2, : ]
    ax0.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
    ax0.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
    ax0.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
    ax0.set_xlabel(r'$x_1$-dimension')
    ax0.set_ylabel(r'$x_2$-dimension')
    ax1 = fig342.add_subplot(222)
    idx = 4
    set0 = X[ labels3[ :, idx ] == 0, : ]
    set1 = X[ labels3[ :, idx ] == 1, : ]
    set2 = X[ labels3[ :, idx ] == 2, : ]
    ax1.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
```

```
ax1.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax1.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax1.set_xlabel(r'$x_1$-dimension')
ax1.set_ylabel(r'$x_2$-dimension')
ax2 = fig342.add_subplot(223)
idx = 1
set0 = X[ labels3[ :, idx ] == 0, : ]
set1 = X[ labels3[ :, idx ] == 1, : ]
set2 = X[ labels3[ :, idx ] == 2, : ]
ax2.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax2.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax2.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax2.set_xlabel(r'$x_1$-dimension')
ax2.set_ylabel(r'$x_2$-dimension')
ax3 = fig342.add_subplot(224)
idc = 0
set0 = X[ labels3[ :, idx ] == 0, : ]
set1 = X[ labels3[ :, idx ] == 1, : ]
set2 = X[ labels3[ :, idx ] == 2, : ]
ax3.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax3.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax3.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax3.set_xlabel(r'$x_1$-dimension')
ax3.set_ylabel(r'$x_2$-dimension')

fig341 = plt.figure()
ax1 = fig341.add_subplot(121)
ax1.plot(np.arange(1, n_iterations), loglikelihoods3[ 1:, 1 ], 'r',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 2 ], 'y--',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 3 ], 'b',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 4 ], 'y',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 5 ], 'r--',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 6 ], 'g--',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 7 ], 'b--',
         np.arange(1, n_iterations), loglikelihoods3[ 1:, 0 ], 'g'
         )
ax1.set_xlabel('Update step of EM-algorithm')
ax1.set_ylabel('Log-likelihood of sample distribution')
plt.title(r'$\mathrm{Loglikelihood}$')
ax2 = fig341.add_subplot(122)
ax2.plot(np.arange(1, n_iterations), criterions3[ 1:, 1 ], 'r',
         np.arange(1, n_iterations), criterions3[ 1:, 2 ], 'y--',
         np.arange(1, n_iterations), criterions3[ 1:, 3 ], 'b',
         np.arange(1, n_iterations), criterions3[ 1:, 4 ], 'y',
         np.arange(1, n_iterations), criterions3[ 1:, 5 ], 'r--',
         np.arange(1, n_iterations), criterions3[ 1:, 6 ], 'g--',
         np.arange(1, n_iterations), criterions3[ 1:, 7 ], 'b--',
         np.arange(1, n_iterations), criterions3[ 1:, 0 ], 'g'
```

```
        )
ax2.set_xlabel('Update step of EM-algorithm')
ax2.set_ylabel('Bayes information criterion')
plt.title(r'$\mathrm{BIC}$')

# We move from 3 to 4 Gaussian components
randomisations = 8
n_iterations = 100
loglikelihoods4 = np.zeros((n_iterations, randomisations))
criterions4 = np.zeros((n_iterations, randomisations))
labels4 = np.zeros((X.shape[ 0 ], randomisations))
K = 4
for j in np.arange(randomisations):
    np.random.seed(j)

    # initialisation
    init_means = np.repeat(np.mean(X, axis=0), K, axis=0).reshape(K, D)
    init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
    init_weights = np.ones(K, dtype=float) / K
    init_covars = np.zeros((K, D, D))
    for k in np.arange(K):
        # the initialisation cannot lead to singularity.
        Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
        init_covars[ k, :, : ] = np.diag(Sigma_k)
    init_covars = init_covars[ :, :, : ]

    # Fit a Gaussian mixture with EM using five components
    convergence_print = False
    for i in np.arange(0, n_iterations):
        # the data get quite big so we overwrite it every time
        gmm4 = mixture_models.MixtureModel(
                n_components=K, n_iter=i,
                means_init=init_means, weights_init=init_weights,
                covars_init=init_covars,
                random_state=np.random.seed(randomisations)
        )
        gmm4 = gmm4.fit(X)
        if gmm4.converged_ and not convergence_print:
            print('converged at step {0}'.format(i))
            convergence_print = True
        loglikelihoods4[ i, j ] = np.sum(gmm4.score_samples(X)[ 0 ])
        criterions4[ i, j ] = gmm4.bic(X)
    if not gmm4.converged_:
        print('no convergence in trial {0}'.format(j))
    # The final data labels:
    labels4[ :, j ] = gmm4.score_samples(X)[ 1 ].argmax(axis=1)

# correlation coefficients
```

```
corrcoeffs4 = np.zeros((K, randomisations))
for l in np.arange(randomisations):
    for k in np.arange(K):
        submat = X[ labels4[ :, l ] == k, : ]
        corrcoeffs4[ k, l ] = \
            np.corrcoef(submat[ :, 0 ], submat[ :, 1 ])[ 0, 1 ]
corrcoeffs_mean4 = np.nanmean(corrcoeffs4, axis=1)
print(corrcoeffs_mean4)

# plotting
# scatter plots
fig342 = plt.figure()
ax0 = fig342.add_subplot(221)
idx = 3
set0 = X[ labels4[ :, idx ] == 0, : ]
set1 = X[ labels4[ :, idx ] == 1, : ]
set2 = X[ labels4[ :, idx ] == 2, : ]
set3 = X[ labels4[ :, idx ] == 3, : ]
ax0.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax0.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax0.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax0.scatter(set3[ :, 0 ], set3[ :, 1 ], color='y', s=6, alpha=0.75)
ax0.set_xlabel(r'$x_1$-dimension')
ax0.set_ylabel(r'$x_2$-dimension')
ax1 = fig342.add_subplot(222)
idx = 2
set0 = X[ labels4[ :, idx ] == 0, : ]
set1 = X[ labels4[ :, idx ] == 1, : ]
set2 = X[ labels4[ :, idx ] == 2, : ]
set3 = X[ labels4[ :, idx ] == 3, : ]
ax1.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax1.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax1.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax1.scatter(set3[ :, 0 ], set3[ :, 1 ], color='y', s=6, alpha=0.75)
ax1.set_xlabel(r'$x_1$-dimension')
ax1.set_ylabel(r'$x_2$-dimension')
ax2 = fig342.add_subplot(223)
idx = 6
set0 = X[ labels4[ :, idx ] == 0, : ]
set1 = X[ labels4[ :, idx ] == 1, : ]
set2 = X[ labels4[ :, idx ] == 2, : ]
set3 = X[ labels4[ :, idx ] == 3, : ]
ax2.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
ax2.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
ax2.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
ax2.scatter(set3[ :, 0 ], set3[ :, 1 ], color='y', s=6, alpha=0.75)
ax2.set_xlabel(r'$x_1$-dimension')
ax2.set_ylabel(r'$x_2$-dimension')
```

```
    ax3 = fig342.add_subplot(224)
    idx = 4
    set0 = X[ labels4[ :, idx ] == 0, : ]
    set1 = X[ labels4[ :, idx ] == 1, : ]
    set2 = X[ labels4[ :, idx ] == 2, : ]
    set3 = X[ labels4[ :, idx ] == 3, : ]
    ax3.scatter(set0[ :, 0 ], set0[ :, 1 ], color='b', s=6, alpha=0.75)
    ax3.scatter(set1[ :, 0 ], set1[ :, 1 ], color='r', s=6, alpha=0.75)
    ax3.scatter(set2[ :, 0 ], set2[ :, 1 ], color='g', s=6, alpha=0.75)
    ax3.scatter(set3[ :, 0 ], set3[ :, 1 ], color='y', s=6, alpha=0.75)
    ax3.set_xlabel(r'$x_1$-dimension')
    ax3.set_ylabel(r'$x_2$-dimension')

    fig341 = plt.figure()
    ax1 = fig341.add_subplot(121)
    ax1.plot(np.arange(1, n_iterations), loglikelihoods4[ 1:, 1 ], 'g--',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 2 ], 'g',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 3 ], 'b',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 4 ], 'y',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 5 ],
             'r--',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 6 ],
             'r',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 7 ],
             'b--',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 0 ],
             'y--'
             )
    ax1.set_xlabel('Update step of EM-algorithm')
    ax1.set_ylabel('Log-likelihood of sample distribution')
    plt.title(r'$\mathrm{Loglikelihood}$')
    ax2 = fig341.add_subplot(122)
    ax2.plot(np.arange(1, n_iterations), criterions4[ 1:, 1 ], 'g--',
             np.arange(1, n_iterations), criterions4[ 1:, 2 ], 'g',
             np.arange(1, n_iterations), criterions4[ 1:, 3 ], 'b',
             np.arange(1, n_iterations), criterions4[ 1:, 4 ], 'y',
             np.arange(1, n_iterations), criterions4[ 1:, 5 ], 'r--',
             np.arange(1, n_iterations), criterions4[ 1:, 6 ], 'r',
             np.arange(1, n_iterations), criterions4[ 1:, 7 ], 'b--',
             np.arange(1, n_iterations), criterions4[ 1:, 0 ], 'y--'
             )
    ax2.set_xlabel('Update step of EM-algorithm')
    ax2.set_ylabel('Bayes information criterion')
    plt.title(r'$\mathrm{BIC}$')
    plt.show()

if ex35:
    # initialisation
```

```
random_state = np.random.seed(3)
n_iterations = 100
loglikelihoods3 = np.zeros(n_iterations)
criterions3 = np.zeros(n_iterations)
labels3 = np.zeros(X.shape[ 0 ])
K = 3
init_means = np.repeat(np.mean(X, axis=0), K, axis=0).reshape(K, D)
init_means += np.random.random_sample((K, D)) * 2.0 - 1.0
init_weights = np.ones(K, dtype=float) / K
init_covars = np.zeros((K, D, D))
for k in np.arange(K):
    # the initialisation cannot lead to singularity.
    Sigma_k = np.random.random_sample(D) * 4.0 + 2.0
    init_covars[ k, :, : ] = np.diag(Sigma_k)
init_covars = init_covars[ :, :, : ]

# optimal solution
convergence_print = False
for i in np.arange(0, n_iterations):
    # the data get quite big so we overwrite it every time
    gmm3 = mixture_models.MixtureModel(
            n_components=K, n_iter=i,
            means_init=init_means, weights_init=init_weights,
            covars_init=init_covars,
            random_state=random_state
    )
    gmm3 = gmm3.fit(X)
    if gmm3.converged_ and not convergence_print:
        print('converged at step {0}'.format(i))
        convergence_print = True
    loglikelihoods3[ i ] = np.sum(gmm3.score_samples(X)[ 0 ])
    criterions3[ i ] = gmm3.bic(X)
if not gmm3.converged_:
    print('no convergence in trial {0}'.format(j))
# The final data labels:
labels3[ : ] = gmm3.score_samples(X)[ 1 ].argmax(axis=1)

"""
Exercise 3.5
"""
# Here are our possible culprits:
newsamples = np.array([ [ 11.85, 2.2, 0.5, 4.0 ],
                        [ 11.95, 3.1, 0.0, 1.0 ],
                        [ 12.00, 2.5, 0.0, 2.0 ],
                        [ 12.00, 3.0, 1.0, 6.3 ] ])
alpha = gmm3.predict_proba(newsamples)
    # a priori assumption from two-cluster solution:
    # subject d has taken the substance
```

```
            # subject c has tampered with their values
```

The Mixture model class

```
"""
Gaussian/Bernoulli Mixture Models.
"""

import numpy as np
from scipy import linalg
from sklearn.externals.six.moves import zip
from itertools import product

EPS = np.finfo(float).eps

class MixtureModel:
    def __init__(self,
                 means_init,
                 weights_init,
                 covars_init=None,
                 n_components=1,
                 random_state=None,
                 errtol=1e-8,
                 min_covar=1e-8,
                 n_iter=1,
                 distrib='Gaussian'):
        self.distrib = distrib
        self.n_components = n_components
        self.errtol = errtol
        self.min_covar = min_covar
        self.random_state = random_state
        self.n_iter = n_iter
        self.is_fitted = False
        self.converged_ = False
        self.means_ = means_init
        self.weights_ = weights_init
        self.covars_ = covars_init

    def score_samples(self, X):
        """
        This includes computing the responsibilities, so the E-Step
        """
        X = np.asarray(X)
        if X.ndim == 1:
            X = X[ :, np.newaxis ]
        if X.size == 0:
            return np.array([ ]), np.empty((0, self.n_components))
        if X.shape[ 1 ] != self.means_.shape[1]:
            raise ValueError('The shape of X is not compatible with self')

        if self.distrib == 'Gaussian':
            lpr = _log_multivariate_normal_density(
                    X, self.means_, self.covars_) + np.log(self.weights_)
        elif self.distrib == 'Bernoulli':
            lpr = _log_bernoulli_density(
                    X, self.means_) + np.log(self.weights_)
        else:
            raise ValueError('Did not find a distribution to score samples on')
        logprob = np.log(np.sum(np.exp(lpr), axis=1))
        # (9.23) / (9.56)
        responsibilities = np.exp(lpr - logprob[:, np.newaxis])
        return logprob, responsibilities

    def predict_proba(self, X):
        """public output of the responsibilities"""
        logprob, responsibilities = self.score_samples(X)
        return responsibilities

    def _fit(self, X, y=None, do_prediction=False):
        # initialization step
        X = np.asarray(X, dtype=np.float64)
        if X.shape[ 0 ] < self.n_components:
            raise ValueError(
                'GMM estimation with %s components, but got only %s samples' %
                (self.n_components, X.shape[ 0 ]))

        max_log_prob = -np.infty

        # EM algorithms
        current_log_likelihood = None
        self.converged_ = False

        for i in np.arange(self.n_iter):
            prev_log_likelihood = current_log_likelihood
            # Expectation step
            log_likelihoods, responsibilities = self.score_samples(X)
            current_log_likelihood = log_likelihoods.mean()

            # Check for convergence.
            if prev_log_likelihood is not None:
                change = abs(current_log_likelihood - prev_log_likelihood)
                if change < self.errtol:
                    self.converged_ = True
                    break

            # Maximization step
            self._do_mstep(X, responsibilities, self.min_covar)

        # if the results are better, keep it
        if self.n_iter:
            if current_log_likelihood > max_log_prob:
                if self.distrib == 'Gaussian':
                    best_params = {'weights': self.weights_,
                                   'means': self.means_,
                                   'covars': self.covars_}
                elif self.distrib == 'Bernoulli':
                    best_params = {'weights': self.weights_,
                                   'means':   self.means_}
        if self.n_iter:
            if self.distrib == 'Gaussian':
                self.covars_ = best_params['covars']
            self.means_ = best_params['means']
            self.weights_ = best_params['weights']
        else:
            responsibilities = np.zeros((X.shape[ 0 ], self.n_components))
        self.is_fitted = True
        return responsibilities

    def fit(self, X, y=None):
        """ The public version to call the fit and get responsibilities"""
        self._fit(X, y)
        return self

    def _do_mstep(self, X, responsibilities, min_covar=0):
        """ Perform the Mstep of the EM algorithm and return the class weights
        """
        weights = responsibilities.sum(axis=0)
        weighted_X_sum = np.dot(responsibilities.T, X)
        inverse_weights = 1.0 / (weights[ :, np.newaxis ] + 10 * EPS)

        if self.distrib == 'Gaussian':
            # (9.26)
            self.weights_ = (weights / (weights.sum() + 10 * EPS) + EPS)
            # (9.24)
            self.means_ = weighted_X_sum * inverse_weights
            # (9.25)
            self.covars_ = _covar_mstep(self, X, responsibilities, weighted_X_sum,
                    inverse_weights, min_covar)
        elif self.distrib == 'Bernoulli':
            # (9.60)
            self.weights_ = weights / weights.sum()
            # (9.59)
            self.means_ = weighted_X_sum * inverse_weights
        return weights
    def _n_parameters(self):
        """Return the number of free parameters in the model."""
        ndim = self.means_.shape[1]
        cov_params = self.n_components * ndim * (ndim + 1) / 2.
        mean_params = ndim * self.n_components
        return int(cov_params + mean_params + self.n_components - 1)

    def bic(self, X):
        return (-2 * self.score_samples(X)[ 0 ].sum() +
                self._n_parameters() * np.log(X.shape[0]))

#########################################################################
# some helper routines
#########################################################################


def _log_multivariate_normal_density(X, means, covars, min_covar=1.e-7):
    """Log probability for covariance matrices."""
    n_samples, n_dim = X.shape
    K = len(means)
    log_prob = np.empty((n_samples, K))
    for c, (mu, cv) in enumerate(zip(means, covars)):
        try:
            cv_chol = linalg.cholesky(cv, lower=True)
        except linalg.LinAlgError:
            # The model is most probably stuck in a component with too
            # few observations, we need to reinitialize this components
            try:
                cv_chol = linalg.cholesky(cv + min_covar * np.eye(n_dim),
                                          lower=True)
            except linalg.LinAlgError:
                raise ValueError("'covars' must be symmetric, "
                                 "positive-definite")
        # find the precision via determinant formula and cholesky decomposition
        cv_log_det = 2 * np.sum(np.log(np.diagonal(cv_chol)))
        cv_sol = linalg.solve_triangular(cv_chol, (X - mu).T, lower=True).T
        log_prob[:, c] = - .5 * (np.sum(cv_sol ** 2, axis=1) +
                                 n_dim * np.log(2 * np.pi) + cv_log_det)
    return log_prob


def _log_bernoulli_density(X, means):
    """Log probability for covariance matrices."""
    n_samples, n_dim = X.shape
    K = len(means)
    log_prob = np.zeros((n_samples, K))
    for n, k in product(np.arange(n_samples), np.arange(K)):
        # need log (p(x_n | mu_k)), that is, log( 9.48 )
        # add small constants, because taking the log still leads to errors
        log_prob[ n, k ] = np.sum(np.log(means[ k ] + 10 * EPS) *
            X[ n, : ] + np.log(np.ones_like(means[ k ]) - means[ k ] + 10 *
            EPS) * (np.ones_like(X[ n, : ]) - X[ n, : ]))
    return log_prob


def _covar_mstep(gmm, X, responsibilities, weighted_X_sum, norm, min_covar):
    """Performing the covariance M step"""
    n_features = X.shape[1]
    cv = np.empty((gmm.n_components, n_features, n_features))
    for c in np.arange(gmm.n_components):
        post = responsibilities[:, c]
        mu = gmm.means_[c]
        diff = X - mu
        with np.errstate(under='ignore'):
            # (9.25)
            avg_cv = np.dot(post * diff.T, diff) / (post.sum() + 10 * EPS)
        cv[c] = avg_cv + min_covar * np.eye(n_features)
    return cv
```

4 Handwritten digit recognition with Bernoulli

4.1 Loading and visualising the data

First, note that loading and displaying data of course works differently in Python:

```
N = 800
D = 28

data = np.fromfile('../Data/a012_images.dat', dtype=np.int8)
data = np.array(data, dtype=int)
data2 = data.reshape(N, D, D)  # just for visualisation
data = np.array(data.reshape(N, D**2), dtype=float)

fig, ax = plt.subplots()
for i in np.arange(data2.shape[ 0 ]):
    ax.imshow(data2[ i, :, : ].T, cmap='gray_r')
    plt.pause(1e-3)
```

This shows all images consecutively. Here are some examples:

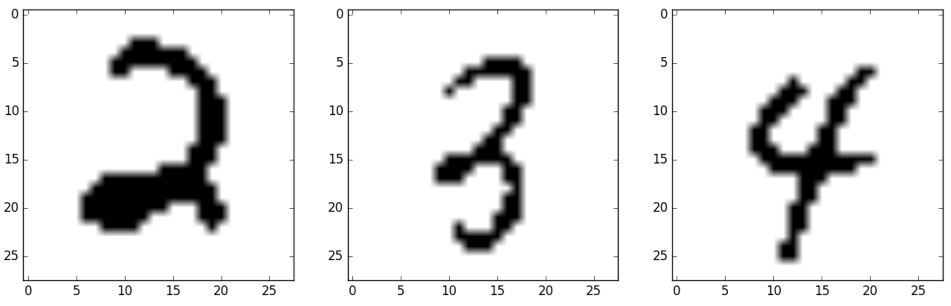


Figure 1: One sample image of each target class to be trained.

Similar classifications with Mixture models, particularly with Bernoulli distributions have been performed successfully, see e.g. "Bernoulli mixture models for binary images" by Juan and Vidal (2004), so yes, it is possible. Let's maximise our performance!

4.2 Expectation maximisation for Bernoulli mixture models

The class code has already been appended to the previous section of the document. It is the same class as for the GMMs, with the class variable `bmm.distrib = 'Bernoulli'` set on initialisation. This then omits a starting value for the covariances, as the responsibilities enter only in terms which can be computed directly (see below). We can initialise an instance analogously to exercise 3 (see section 4.3).

For `_do_mstep` we have replaced formulas (9.24) - (9.26) by (9.59), (9.60), but the implementation stays identical. The loglikelihood can be computed analogously to the sum part of (9.54) (see function `_log_bernoulli_density`). Thus, algorithmically, the BMM is a simplification of the GMM with another density.

Note, that the images are imported in vector format, to keep the old class structure (other than in part 4.1). Note also, that we included an numerically small ϵ (see (9.41), (9.42)) to avoid annihilation of the loglikelihood sum because applying only the logarithm did not work, analogously to the GMM (likely due to numerically different processing than with Matlab).

4.3 Running the algorithm

After 4.2 we can now run the algorithm. Let us have a look at how the class means develop (see figure 2).

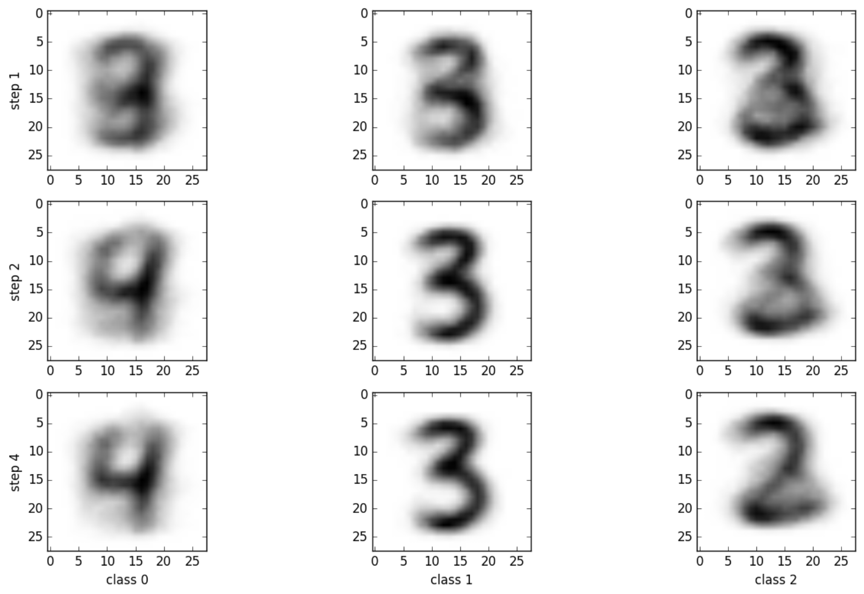


Figure 2: Another example for the rapid progression of EM: after the first steps, hardly any change is visible. At the final step, the images are slightly more smeared.

In the first steps, the means appear to be an overlay of the different numbers. As the weights begin to favour one of the classes, the respective digit becomes more distinct. After some steps, the class-digit is clearly readable. From this point on, the mean appears to loose contrast. This can already be interpreted as overfitting.

When comparing the means to cases with $K = 4$, we see that in the latter case only the 2 is really pronounced (see figure 3), while the other classes contain mixed features from all digits. This is also the case for $K = 2$ (see figure 4), but here 2 and 3 cristallise out and 4 remains missing, while parts of it are still in the left side of the 2.

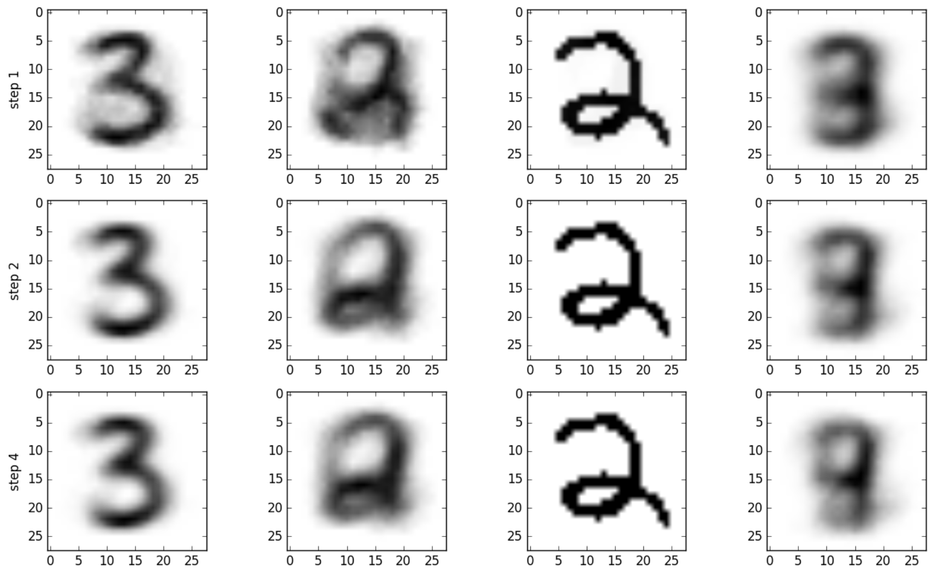


Figure 3: Class means when forcing $K = 4$ components.

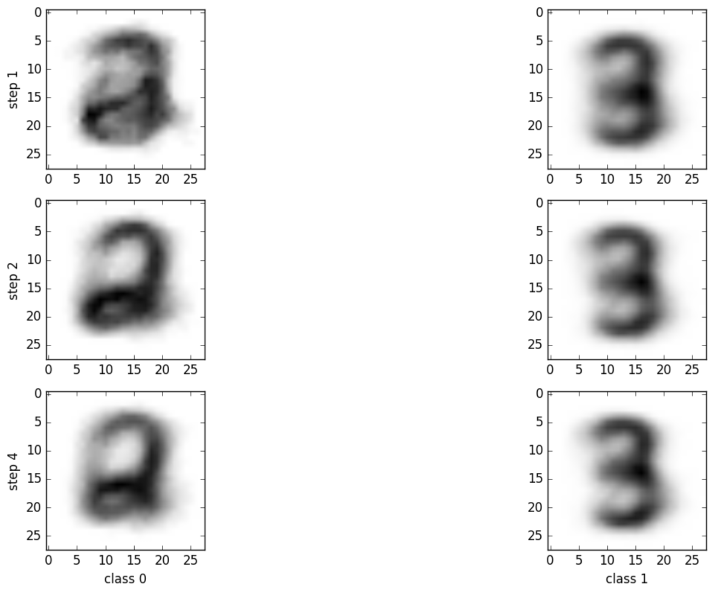


Figure 4: Class means when forcing $K = 4$ components.

4.4 Performance analysis

To compare the true labels with the learned ones, we need to permutate our classes to find the right mapping to the MNIST classes.

```
# now let's compare to the ground truth:
ground_truth = np.fromfile('../Data/a012_labels.dat', dtype=np.int8)
ground_truth = np.array(ground_truth, dtype=int)
# we need to permute the labels, to see how many numbers were
# identified correctly!
permutation_list = [ ]
agreement = np.zeros(int(factorial(len(np.unique(labels)))))
for i, p in enumerate(permutations(np.unique(labels))):
    relabeled = np.zeros_like(labels)
    permutation_list.append(p)
    for j in np.arange(len(np.unique(labels))):
        relabeled[ labels == np.unique(labels)[ j ] ] = p[ j ]
    agreement[ i ] = np.sum(relabeled == ground_truth)
relabeled = np.zeros_like(labels)
for j in np.arange(len(np.unique(labels))):
    relabeled[ labels == np.unique(labels)[ j ] ] = permutation_list[
        np.argmax(agreement) ][ j ]
# how well did we perform?
performance = np.sum(relabeled == ground_truth)
```

It turns out, that for our initial instance we already identified 743 labels correctly, which is a performance of 92.875%.

When changing the number of classes, the performance is obviously reduced. This is reflected in the loglikelihood and the BIC in comparison to the $K = 3$-classification (see figures 5 and 6).

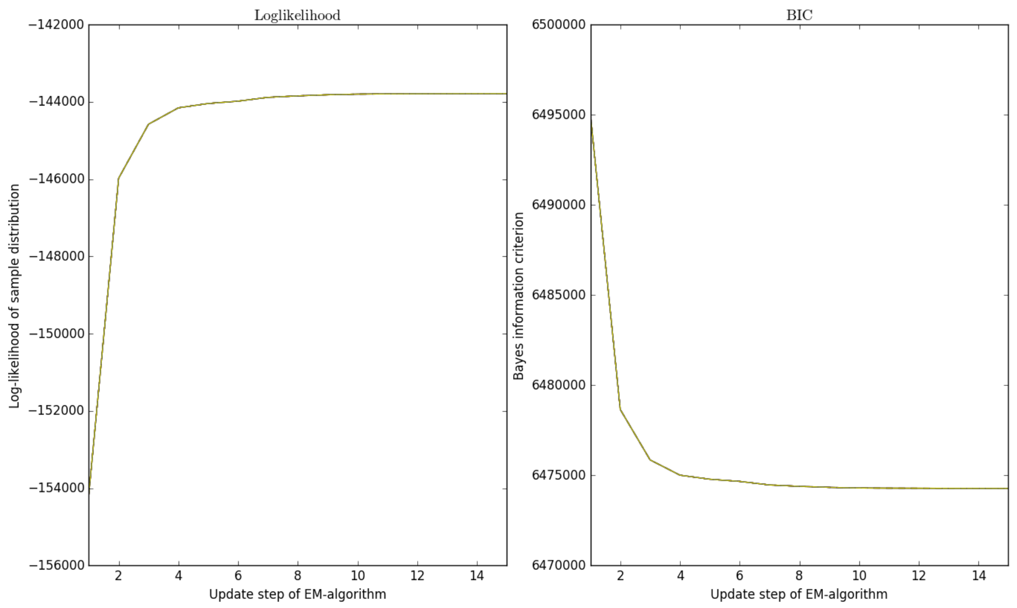


Figure 5: The curves for the different initialisations of the $K = 3$ model. As in exercise 3, the spanned range is so large (due to rapid convergence) that the plots look virtually identical. When initialising the model with the true means, convergence is slower, because the model "cannot decide" which component to weight in favour of which digit.

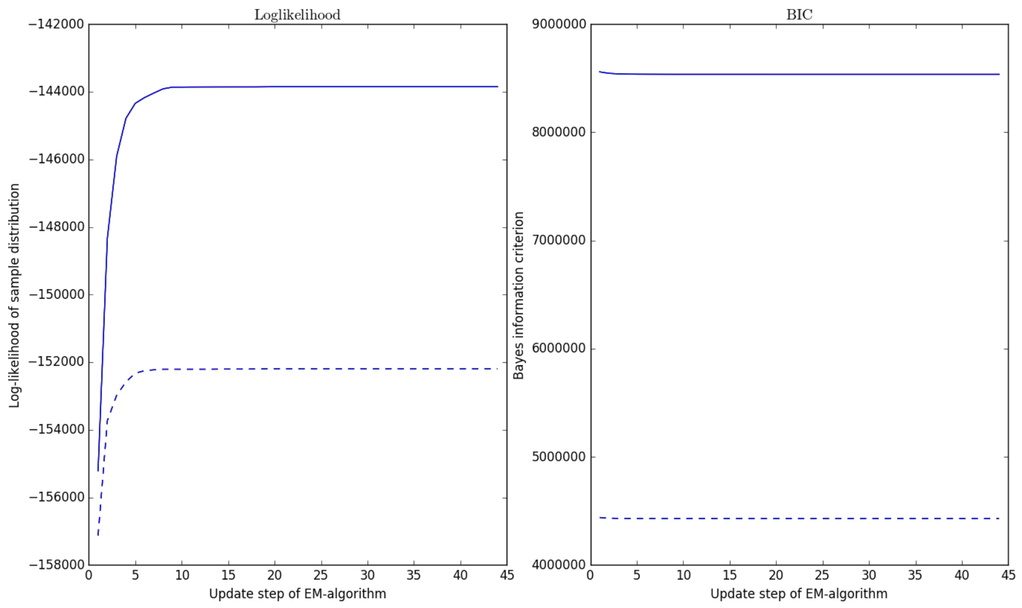


Figure 6: The straight lines represent the $K = 4$, the dashed lines the $K = 2$ model. Obviously, both perform slower and worse in total, than the $K = 3$ solutions.

The differences also become evident when observing the development of class means.

The performance could be improved by splitting the dataset and obtaining a validation set. This would also reduce the overfitting.

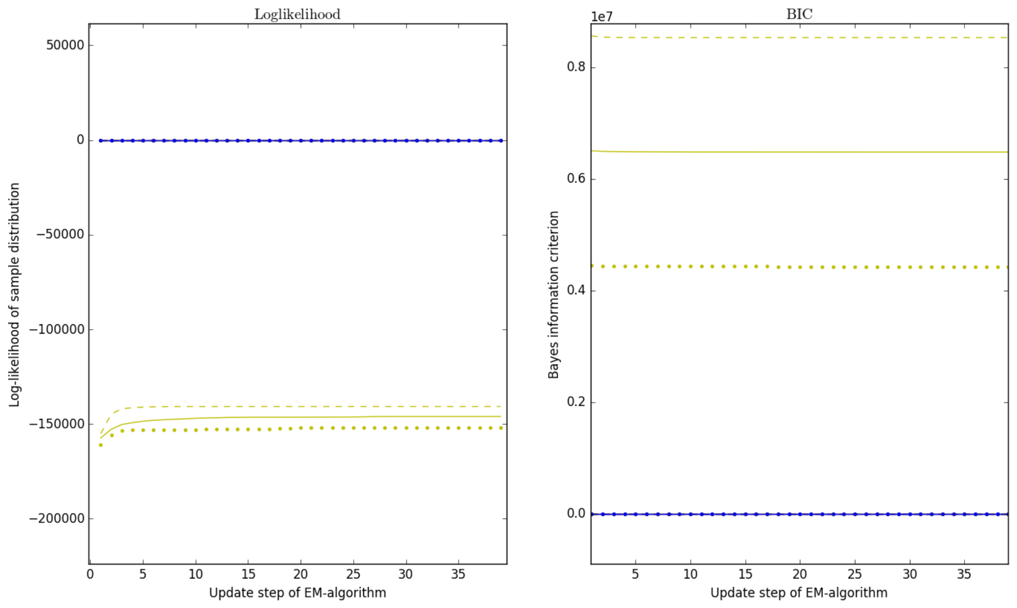


Figure 7: Direct comparison of different component numbers (this looks a bit strange). Straight lines are for 3, dotted for 2 and dashed lines for 4 clusters. The yellow line lies above two other lines, which are generated from different random seeds.

4.5 Self-made samples

This was quite interesting. We started with the following set of handwritten images.

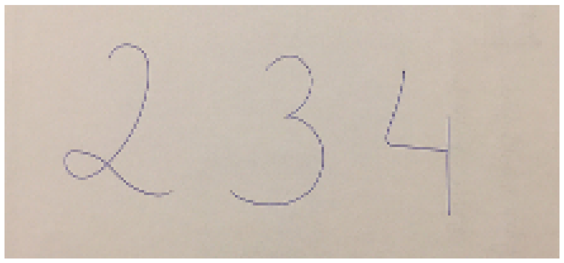
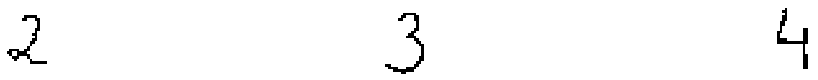


Figure 8: Handwritten samples.

The images had to be separated for individual digits. We then repeatedly applied thresholding and scaled down the image size to obtain 28×28-sized images of grayscales. The result looks like this:



We then import the images and ask to assign probabilities for the mean classes:

```
import scipy.ndimage as nd
D = 28
img1 = nd.imread('../Figures/4.png', flatten=True)/255.
img2 = nd.imread('../Figures/2.png', flatten=True)/255.
img3 = nd.imread('../Figures/3.png', flatten=True)/255.
img1 = np.reshape(1 - np.resize(1 - img1, (28, 28)), D**2)
img2 = np.reshape(1 - np.resize(1 - img2, (28, 28)), D**2)
img3 = np.reshape(1 - np.resize(1 - img3, (28, 28)), D**2)
images = np.vstack((np.vstack((img1, img2)), img3))
alpha = bmm.predict_proba(images)
print(alpha)
```

Considering that there is a permutation in the labelling, the images are assigned the following way:

```
[[             inf              inf              inf]
 [  6.64612757e-01   2.14713483e-38   3.35387243e-01]
 [             inf              inf              inf]]
```

Considering the permutation, the classifier correctly identified the second image as a 2. With the other two images there still seems to be an issue with the log...

5 Code

```
# coding=utf-8
"""
This file contains solutions to Exercise 4 of Assignment 4 of Bert
 Kappen's course "Statistical Machine Learning" 2016/2017.
"""

import numpy as np
from scipy import linalg
import os
import copy
import matplotlib.pyplot as plt
import matplotlib as mpl
import matplotlib.cm as cmx
import mixture_models
from scipy.misc import factorial
from itertools import product
from itertools import permutations

def ground_truth_comparison(labels):
    """Take classification and compare to MNIST labels"""
    # now let's compare to the ground truth:
    ground_truth = np.fromfile('../Data/a012_labels.dat', dtype=np.int8)
    ground_truth = np.array(ground_truth, dtype=int)

    # we need to permute the labels, to see how many numbers were
    # identified correctly!
    permutation_list = [ ]
    agreement = np.zeros(int(factorial(len(np.unique(labels)))))
    for i, p in enumerate(permutations(np.unique(labels))):
        relabeled = np.zeros_like(labels)
        permutation_list.append(p)
        for j in np.arange(len(np.unique(labels))):
            relabeled[ labels == np.unique(labels)[ j ] ] = p[ j ]
        agreement[ i ] = np.sum(relabeled == ground_truth)
    relabeled = np.zeros_like(labels)
    for j in np.arange(len(np.unique(labels))):
        relabeled[ labels == np.unique(labels)[ j ] ] = permutation_list[
            np.argmax(agreement) ][ j ]

    # how well did we perform?
    performance = np.sum(relabeled == ground_truth)
    permutation = permutation_list[ np.argmax(agreement) ]
    return performance, permutation


if __name__ == '__main__':
```

```
ex41 = False  # False just supresses the output, but still loads the data
ex42 = True
ex43 = False
ex442 = False
ex444 = False
ex45 = True
plots1 = False
plots2 = False
plots3 = False
plotnow = False
the_plot = False
"""
Exercise 4.1
"""
# load the image data
N = 800
D = 28
data = np.fromfile('../Data/a012_images.dat', dtype=np.int8)
data = np.array(data, dtype=int)
data2 = data.reshape(N, D, D)  # just for visualisation
data = np.array(data.reshape(N, D**2), dtype=float)
n_iterations = 45

if ex41:
    # fig, ax = plt.subplots()
    # for i in np.arange(data2.shape[ 0 ]):
    #     ax.imshow(data2[ i, :, : ].T, cmap='gray_r')
    #     plt.pause(1e-3)
    fig41 = plt.figure()
    ax1 = fig41.add_subplot(131)
    ax1.imshow(data2[ 0, :, : ].T, cmap='gray_r')
    ax2 = fig41.add_subplot(132)
    ax2.imshow(data2[ 1, :, : ].T, cmap='gray_r')
    ax3 = fig41.add_subplot(133)
    ax3.imshow(data2[ 4, :, : ].T, cmap='gray_r')

"""
Exercise 4.2
"""
# The class for the BMM algorithm is contained in the mixture models
if ex42:

    # K = 3 ---------------------------------------------------------
    # set up some stuff for the gaussian mixture models
    n_iterations = 40
    K = 3
    seed = np.random.seed(3)  # shoud be initialised automatically
    # initialisation
    init_means = np.random.random_sample((K, D**2)) * 0.5 + 0.25
    init_weights = np.ones(K, dtype=float) / K

    # Fit a Gaussian mixture with EM using five components
    loglikelihoods = np.zeros(n_iterations)
    criterions = np.zeros(n_iterations)
    convergence_print = False

    # let's initialise the model for a single run.
    bmm = mixture_models.MixtureModel(n_components=K,
                          means_init=init_means,
                          weights_init=init_weights,
                          n_iter=n_iterations,
                          distrib='Bernoulli')
    bmm = bmm.fit(data)
    loglikelihoods = np.sum(bmm.score_samples(data)[ 0 ])
    criterions = bmm.bic(data)
    labels = bmm.score_samples(data)[ 1 ].argmax(axis=1)
    labels += 2

    performance1, permutation1 = ground_truth_comparison(labels)

"""
Exercise 4.3
"""
if ex43:
    # --------------------------- K = 3 ------------------------------
    K = 3
    n_randomisations = 4
    n_iterations = 40
    for r in np.arange(n_randomisations):
        print('r = {0}'.format(r))
        seed = np.random.RandomState(r)
        init_means = np.random.random_sample((K, D**2)) * 0.5 + 0.25
        init_weights = np.ones(K, dtype=float) / K
        loglikelihoods = np.zeros((n_iterations, n_randomisations))
        criterions = np.zeros((n_iterations, n_randomisations))
        class_means = np.zeros((n_iterations, K, D**2))
        labels = np.zeros((data.shape[ 0 ], n_iterations))
        convergence_print = False
        for i in np.arange(n_iterations):
            print('iteration = {0}'.format(i))
            bmm = mixture_models.MixtureModel(n_components=K,
                                              means_init=init_means,
                                              weights_init=init_weights,
                                              n_iter=i,
                                              distrib='Bernoulli',
                                              random_state=seed)
            bmm = bmm.fit(data)
            # give the class means:
            for k in np.arange(K):
                class_means[ i, k, : ] = bmm.means_[ k, : ]
            loglikelihoods[ i, r ] = np.sum(bmm.score_samples(data)[ 0 ])
            criterions[ i, r ] = bmm.bic(data)
            # convergence?
            if bmm.converged_ and not convergence_print:
                print('converged at step {0}'.format(i))
                convergence_print = True
            labels[ :, r ] = bmm.score_samples(data)[ 1 ].argmax(axis=1)
        labels += 2
                # break
        if not bmm.converged_:
            print('no convergence in trial {0}'.format(r))

"""
Exercise 4.4
"""
# -------------------------- K = 2 ------------------------------
K = 2
n_randomisations = 4
n_iterations = 45
for r in np.arange(n_randomisations):
    print('r = {0}'.format(r))
    seed = np.random.RandomState(r)
    init_means = np.random.random_sample((K, D**2)) * 0.5 + 0.25
    init_weights = np.ones(K, dtype=float) / K
    loglikelihood2 = np.zeros((n_iterations, n_randomisations))
    criterions2 = np.zeros((n_iterations, n_randomisations))
    class_means2 = np.zeros((n_iterations, n_randomisations, K, D**2))
    labels2 = np.zeros((data.shape[ 0 ], n_randomisations))
    convergence_print = False
    for i in np.arange(n_iterations):
        print('iteration = {0}'.format(i))
        bmm2 = mixture_models.MixtureModel(n_components=K,
                                          means_init=init_means,
                                          weights_init=init_weights,
                                          n_iter=i,
                                          distrib='Bernoulli',
                                          random_state=seed)
        bmm2 = bmm2.fit(data)
        # give the class means:
        for k in np.arange(K):
            class_means2[ i, r, k, : ] = bmm2.means_[ k, : ]
        loglikelihood2[ i, r ] = np.sum(bmm2.score_samples(data)[ 0 ])
        criterions2[ i, r ] = bmm2.bic(data)
```

```
            # convergence?
            if bmm2.converged_ and not convergence_print:
                print('converged at step {0}'.format(i))
                convergence_print = True
            labels2[ :, r ] = bmm2.score_samples(data)[ 1 ].argmax(
                    axis=1)
        labels2 += 2
                # break
        if not bmm2.converged_:
            print('no convergence in trial {0}'.format(r))

    # --------------------------- K = 4 ------------------------------
    K = 4
    n_randomisations = 4
    n_iterations = 45
    for r in np.arange(n_randomisations):
        print('r = {0}'.format(r))
        seed = np.random.RandomState(r)
        init_means = np.random.random_sample((K, D**2)) * 0.5 + 0.25
        init_weights = np.ones(K, dtype=float) / K
        loglikelihoods4 = np.zeros((n_iterations, n_randomisations))
        criterions4 = np.zeros((n_iterations, n_randomisations))
        class_means4 = np.zeros((n_iterations, n_randomisations, K, D**2))
        labels4 = np.zeros((data.shape[ 0 ], n_randomisations))
        convergence_print = False
        for i in np.arange(n_iterations):
            print('iteration = {0}'.format(i))
            bmm4 = mixture_models.MixtureModel(n_components=K,
                                              means_init=init_means,
                                              weights_init=init_weights,
                                              n_iter=i,
                                              distrib='Bernoulli',
                                              random_state=seed)
            bmm4 = bmm4.fit(data)
            # give the class means:
            for k in np.arange(K):
                class_means4[ i, r, k, : ] = bmm4.means_[ k, : ]
            loglikelihoods4[ i, r ] = np.sum(bmm4.score_samples(data)[ 0 ])
            criterions4[ i, r ] = bmm4.bic(data)
            # convergence?
            if bmm4.converged_ and not convergence_print:
                print('converged at step {0}'.format(i))
                convergence_print = True
            labels4[ :, r ] = bmm4.score_samples(data)[ 1 ].argmax(
                    axis=1)
                # break
        labels4 += 2
        if not bmm4.converged_:
            print('no convergence in trial {0}'.format(r))

if plots1:
    fig431 = plt.figure()
    ax1 = fig431.add_subplot(331)
    ax1.imshow(class_means[ 1, 0, 0, : ].reshape(D, D).T,
               cmap='gray_r')
    ax1.set_ylabel('step 1', size=12)
    ax2 = fig431.add_subplot(332)
    ax2.imshow(class_means[ 1, 0, 1, : ].reshape(D, D).T,
               cmap='gray_r')
    ax3 = fig431.add_subplot(333)
    ax3.imshow(class_means[ 1, 0, 2, : ].reshape(D, D).T,
               cmap='gray_r')
    ax4 = fig431.add_subplot(334)
    ax4.imshow(class_means[ 2, 0, 0, : ].reshape(D, D).T,
               cmap='gray_r')
    ax4.set_ylabel('step 2', size=12)
    ax5 = fig431.add_subplot(335)
    ax5.imshow(class_means[ 2, 0, 1, : ].reshape(D, D).T,
               cmap='gray_r')
    ax6 = fig431.add_subplot(336)
    ax6.imshow(class_means[ 2, 0, 2, : ].reshape(D, D).T,
               cmap='gray_r')
    ax7 = fig431.add_subplot(337)
    ax7.imshow(class_means[ 4, 0, 0, : ].reshape(D, D).T,
               cmap='gray_r')
    ax7.set_ylabel('step 4', size=12)
    ax7.set_xlabel('class 0', size=12)
    ax8 = fig431.add_subplot(338)
    ax8.imshow(class_means[ 4, 0, 1, : ].reshape(D, D).T,
               cmap='gray_r')
    ax8.set_xlabel('class 1', size=12)
    ax9 = fig431.add_subplot(339)
    ax9.imshow(class_means[ 4, 0, 2, : ].reshape(D, D).T,
               cmap='gray_r')
    ax9.set_xlabel('class 2', size=12)

if plots2:
    fig331 = plt.figure()
    ax1 = fig331.add_subplot(121)
    ax1.plot(np.arange(1, n_iterations), loglikelihoods[ 1:, 0 ], 'b',
             )
    ax1.set_xlabel('Update step of EM-algorithm')
    ax1.set_ylabel('Log-likelihood of sample distribution')
    ax1.set_xlim([ 1, 20 ])
    ax1.set_ylim([ -170000, -160000 ])
    plt.title(r'$\mathrm{Loglikelihood}$')
    ax2 = fig331.add_subplot(122)
    ax2.plot(np.arange(1, n_iterations), criterions[ 1:, 0 ], 'b',
             )
    ax2.set_xlim([ 1, 20 ])
    ax2.set_xlabel('Update step of EM-algorithm')
    ax2.set_ylabel('Bayes information criterion')
    plt.title(r'$\mathrm{BIC}$')

if ex45:
    # import
    import scipy.ndimage as nd
    D = 28
    img1 = nd.imread('../Figures/4.png', flatten=True)/256.
    img2 = nd.imread('../Figures/2.png', flatten=True)/256.
    img3 = nd.imread('../Figures/3.png', flatten=True)/256.
    img1 = np.reshape(np.resize(1 - img1, (28, 28)), D**2)
    img2 = np.reshape(np.resize(1 - img2, (28, 28)), D**2)
    img3 = np.reshape(np.resize(1 - img3, (28, 28)), D**2)
    images = np.vstack((np.vstack((img1, img2)), img3))
    alpha = bmm.predict_proba(images)
    print(alpha)

if plots3:
    fig441 = plt.figure()
    ax1 = fig441.add_subplot(121)
    ax1.plot(np.arange(1, n_iterations), loglikelihoods4[ 1:, 0 ], 'r',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 1 ], 'g',
             np.arange(1, n_iterations), loglikelihoods4[ 1:, 2 ], 'b',
             # np.arange(1, n_iterations), loglikelihoods4[ 1:, 3 ], 'y',
             np.arange(1, n_iterations), loglikelihood2[ 1:, 0 ], 'r--',
             np.arange(1, n_iterations), loglikelihood2[ 1:, 1 ], 'g--',
             np.arange(1, n_iterations), loglikelihood2[ 1:, 2 ], 'b--',
             # np.arange(1, n_iterations), loglikelihood2[ 1:, 3 ], 'y--'
             )
    ax1.set_xlabel('Update step of EM-algorithm')
    ax1.set_ylabel('Log-likelihood of sample distribution')
    plt.title(r'$\mathrm{Loglikelihood}$')
    ax2 = fig441.add_subplot(122)
    ax2.plot(np.arange(1, n_iterations), criterions4[ 1:, 0 ], 'r',
             np.arange(1, n_iterations), criterions4[ 1:, 1 ], 'g',
             np.arange(1, n_iterations), criterions4[ 1:, 2 ], 'b',
             # np.arange(1, n_iterations), criterions4[ 1:, 3 ], 'y',
             np.arange(1, n_iterations), criterions2[ 1:, 0 ], 'r--',
             np.arange(1, n_iterations), criterions2[ 1:, 1 ], 'g--',
             np.arange(1, n_iterations), criterions2[ 1:, 2 ], 'b--',
             # np.arange(1, n_iterations), criterions2[ 1:, 3 ], 'y--'
             )
    ax2.set_xlabel('Update step of EM-algorithm')
```

```
        ax2.set_ylabel('Bayes information criterion')
        plt.title(r'$\mathrm{BIC}$')

    if the_plot:
        n_iterations = 40
        fig442 = plt.figure()
        ax1 = fig442.add_subplot(121)
        ax1.plot(
                np.arange(1, n_iterations), loglikelihoods[ 1:n_iterations, 0 ], 'r
                np.arange(1, n_iterations), loglikelihoods[ 1:n_iterations, 1 ], 'g
                np.arange(1, n_iterations), loglikelihoods[ 1:n_iterations, 2 ], 'b
                np.arange(1, n_iterations), loglikelihoods[ 1:n_iterations, 3 ], 'y
                np.arange(1, n_iterations), loglikelihood2[ 1:n_iterations, 0 ], 'r
                np.arange(1, n_iterations), loglikelihood2[ 1:n_iterations, 1 ], 'g
                np.arange(1, n_iterations), loglikelihood2[ 1:n_iterations, 2 ], 'b
                np.arange(1, n_iterations), loglikelihood2[ 1:n_iterations, 3 ], 'y
                np.arange(1, n_iterations), loglikelihoods4[ 1:n_iterations, 0 ], '
                np.arange(1, n_iterations), loglikelihoods4[ 1:n_iterations, 1 ], '
                np.arange(1, n_iterations), loglikelihoods4[ 1:n_iterations, 2 ], '
                np.arange(1, n_iterations), loglikelihoods4[ 1:n_iterations, 3 ], '
                )
        ax1.set_xlim([1, 15])
        ax1.set_xlabel('Update step of EM-algorithm')
        ax1.set_ylabel('Log-likelihood of sample distribution')
        plt.title(r'$\mathrm{Loglikelihood}$')
        ax2 = fig442.add_subplot(122)
        ax2.plot(
                np.arange(1, n_iterations), criterions[ 1:n_iterations, 0 ], 'r',
                np.arange(1, n_iterations), criterions[ 1:n_iterations, 1 ], 'g',
                np.arange(1, n_iterations), criterions[ 1:n_iterations, 2 ], 'b',
                np.arange(1, n_iterations), criterions[ 1:n_iterations, 3 ], 'y',
                np.arange(1, n_iterations), criterions2[ 1:n_iterations, 0 ], 'r.',
                np.arange(1, n_iterations), criterions2[ 1:n_iterations, 1 ], 'g.',
                np.arange(1, n_iterations), criterions2[ 1:n_iterations, 2 ], 'b.',
                np.arange(1, n_iterations), criterions2[ 1:n_iterations, 3 ], 'y.',
                np.arange(1, n_iterations), criterions4[ 1:n_iterations, 0 ], 'r--'
                np.arange(1, n_iterations), criterions4[ 1:n_iterations, 1 ], 'g--'
                np.arange(1, n_iterations), criterions4[ 1:n_iterations, 2 ], 'b--'
                np.arange(1, n_iterations), criterions4[ 1:n_iterations, 3 ], 'y--'
                )
        ax2.set_xlim([ 1, 15 ])
        ax2.set_xlabel('Update step of EM-algorithm')
        ax2.set_ylabel('Bayes information criterion')
        plt.title(r'$\mathrm{BIC}$')


# ----------------------------------------------------------------------------
    if plotnow:
```

```
fig443 = plt.figure()
ax1 = fig443.add_subplot(341)
ax1.imshow(class_means[ 1, 0, : ].reshape(D, D).T, cmap='gray_r')
ax1.set_ylabel('step 1', size=12)
ax2 = fig443.add_subplot(342)
ax2.imshow(class_means[ 1, 1, : ].reshape(D, D).T, cmap='gray_r')
ax3 = fig443.add_subplot(343)
ax3.imshow(class_means[ 1, 2, : ].reshape(D, D).T, cmap='gray_r')
ax4 = fig443.add_subplot(344)
ax4.imshow(class_means[ 1, 3, : ].reshape(D, D).T, cmap='gray_r')
ax5 = fig443.add_subplot(345)
ax5.imshow(class_means[ 2, 0, : ].reshape(D, D).T, cmap='gray_r')
ax5.set_ylabel('step 2', size=12)
ax6 = fig443.add_subplot(346)
ax6.imshow(class_means[ 2, 1, : ].reshape(D, D).T, cmap='gray_r')
ax7 = fig443.add_subplot(347)
ax7.imshow(class_means[ 2, 2, : ].reshape(D, D).T, cmap='gray_r')
ax8 = fig443.add_subplot(348)
ax8.imshow(class_means[ 2, 3, : ].reshape(D, D).T, cmap='gray_r')
ax9 = fig443.add_subplot(349)
ax9.imshow(class_means[ 4, 0, : ].reshape(D, D).T, cmap='gray_r')
ax9.set_ylabel('step 4', size=12)
ax10 = fig443.add_subplot(3,4,10)
ax10.imshow(class_means[ 4, 1, : ].reshape(D, D).T, cmap='gray_r')
ax11 = fig443.add_subplot(3,4,11)
ax11.imshow(class_means[ 4, 2, : ].reshape(D, D).T, cmap='gray_r')
ax12 = fig443.add_subplot(3,4,12)
ax12.imshow(class_means[ 4, 3, : ].reshape(D, D).T, cmap='gray_r')

fig444 = plt.figure()
ax1 = fig444.add_subplot(321)
ax1.imshow(class_means[ 1, 0, : ].reshape(D, D).T, cmap='gray_r')
ax1.set_ylabel('step 1', size=12)
ax2 = fig444.add_subplot(322)
ax2.imshow(class_means[ 1, 1, : ].reshape(D, D).T, cmap='gray_r')
ax3 = fig444.add_subplot(323)
ax3.imshow(class_means[ 2, 0, : ].reshape(D, D).T, cmap='gray_r')
ax3.set_ylabel('step 2', size=12)
ax4 = fig444.add_subplot(324)
ax4.imshow(class_means[ 2, 1, : ].reshape(D, D).T, cmap='gray_r')
ax5 = fig444.add_subplot(325)
ax5.imshow(class_means[ 4, 0, : ].reshape(D, D).T, cmap='gray_r')
ax5.set_ylabel('step 4', size=12)
ax5.set_xlabel('class 0', size=12)
ax6 = fig444.add_subplot(326)
ax6.imshow(class_means[ 4, 1, : ].reshape(D, D).T, cmap='gray_r')
ax6.set_xlabel('class 1', size=12)
```

```
plt.show()
```