Statistical Machine Learning: Assignment 4

Joris van Vugt, s4279859

January 4, 2017

Exercise 1 – Gaussian processes for regression
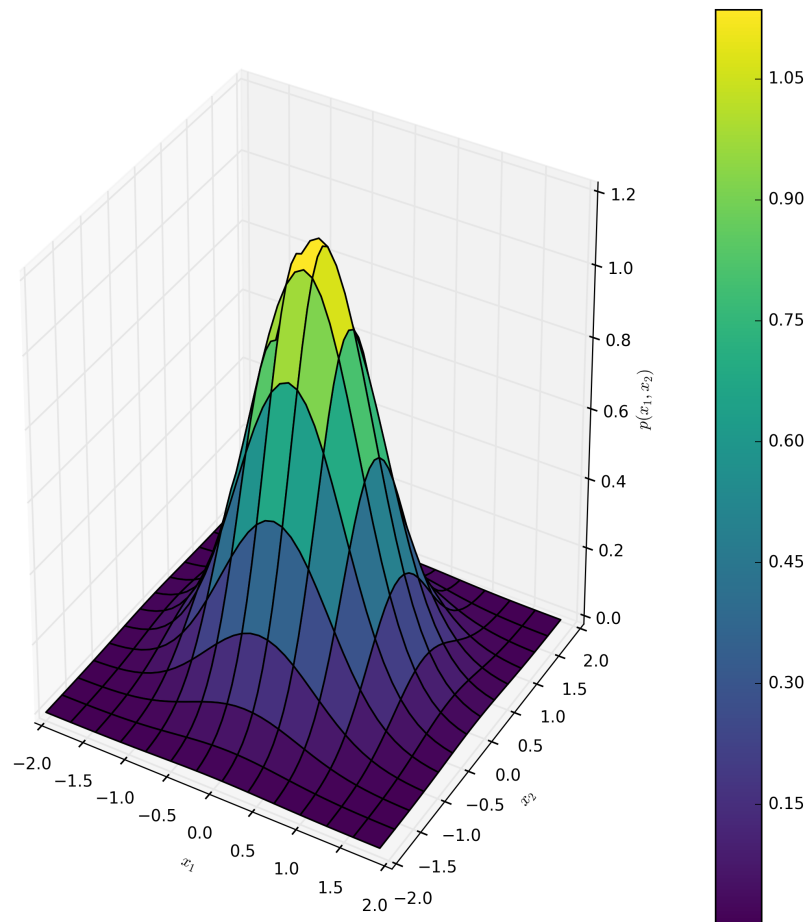
Figure 1: An isotropic 2D gaussian given by $y = 3 \cdot \mathcal{N}(\boldsymbol{x}|\boldsymbol{0}, \frac{2}{5}\boldsymbol{I}_2)$

Exercise 2 – Neural network regression

1. The gaussian shown in Figure 1 is created with the following code

```
X = np.mgrid[-2:2:.1, -2:2:.1]
y = 3 * multivariate_normal(np.zeros((2,)), 0.4*np.eye(2)).pdf(X.T)

fig = plt.figure(figsize=(10, 10), dpi=300)
ax = fig.gca(projection='3d')
col = ax.plot_surface(X[0], X[1], y, cmap=plt.cm.viridis, rstride=3, cstride=3)
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
ax.set_zlabel('$p(x_1, x_2)$')
plt.colorbar(col)
plt.savefig('mvg.png', bbox_inches='tight', dpi=300)
plt.show()
```

2. The following class implements the multi-layer perceptron described in the assignment. It is able to process an entire batch of data simultaneously. However, because of the sum-of-squares error function, the error quickly becomes very large, which is impractical for training.

```python
class MultiLayerPerceptron():
    def __init__(self, D=2, K=1, M=2, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.D, self.K, self.M = D, K, M


        model = {}
        model['W1'] = np.random.uniform(low=-0.5, high=0.5, size=(D, M))
        model['b1'] = np.zeros(M)
        model['W2'] = np.random.uniform(low=-0.5, high=0.5, size=(M, K))
        model['b2'] = np.zeros(K)
        self.model = model

    def forward(self, X):
        """
        X has shape [N x 2]
        Returns the predictions for X [N, 1] and cached variables
        for backprop
        """
        h_in = np.dot(X, self.model['W1']) + self.model['b1']
        h = np.tanh(h_in)
        y = np.dot(h, self.model['W2']) + self.model['b2']
        return y, {'X': X, 'h': h, 'W2': self.model['W2']}

    def backward(self, dout, cache):
        """
        dout is the gradient on the loss function wrt to the predictions
        cache is a dictionary of variables
        """
        grads = {}
        grads['W2'] = np.dot(cache['h'].T, dout)
        grads['b2'] = np.sum(dout, axis=0)
        dh = np.dot(dout, cache['W2'].T)
        dh_in = (1 - cache['h']**2) * dh # backprop through tanh
        grads['W1'] = np.dot(cache['X'].T, dh_in)
        grads['b1'] = np.sum(dh_in, axis=0)
        return grads

    def loss(self, preds, y):
        """
        Computes the sum of squares error
        """
        difference = preds - y.reshape(-1, 1)
        error = 0.5 * np.sum(difference**2)
        return error, difference

    def train(self, X, y, iterations=500, verbose=True):
        for i in range(iterations):
            preds, cache = self.forward(X)
            cost, dpreds = self.loss(preds, y)
            grads = self.backward(dpreds, cache)
            if verbose:
                print('Iteration %d: loss=%.4f' % (i, cost))
            for param in self.model.keys():
                self.model[param] -= grads[param] * self.learning_rate
```
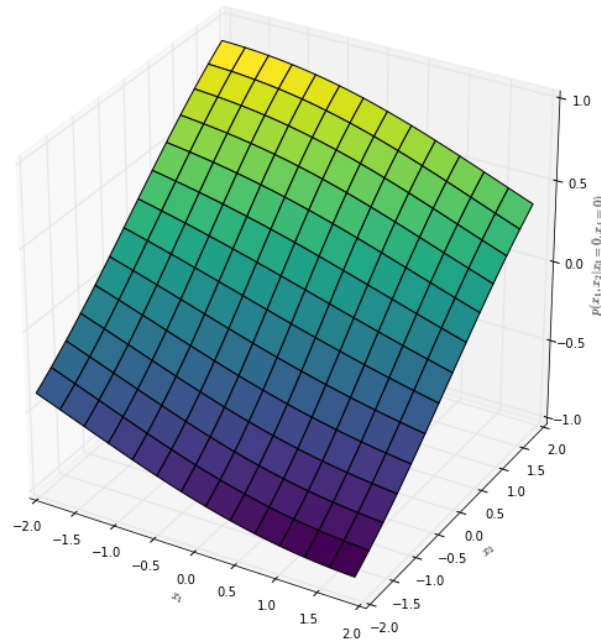
Figure 2: The output of the MLP after random initialization (i.e., no training).

The initial weights yield the "distribution" shown in Figure 2. As to be expected, it doesn't look much like the original distribution yet.

3. After 200 iterations the output indeed starts to resemble a gaussian. However, the loss also shows that the minimal loss was also reached around that point. See Figure 3.

4. Randomly permuting X and Y drastically increases both the speed of convergence and the quality of the final model (see Figure 4. The final loss is 0.54. The reason for this is that the model doesn't overfit as much to specific areas of the gaussian. If the data isn't permuted, the network will see a lot of similar samples sequentially. It will then overfit on this particular area and perform worse on other areas of the gaussian.

Increasing the number of hidden units will further increase the loss (e.g., $M = 40$ yields a loss of 0.07). If the network has more hidden units, it will have more expressive power. The number of weights will also increase, so the weights take longer to converge.

Decreasing the learning rate to 0.01 increases the time it takes to converge. However, there is less noise in the loss over time and the final loss is lower (1.03). With a lower learning rate, gradient descent can make more subtle moves in the loss landscape. This leads to better optima.

4

(a) The output of the network after 200 iterations.   (b) The loss as a function of the number of iterations.
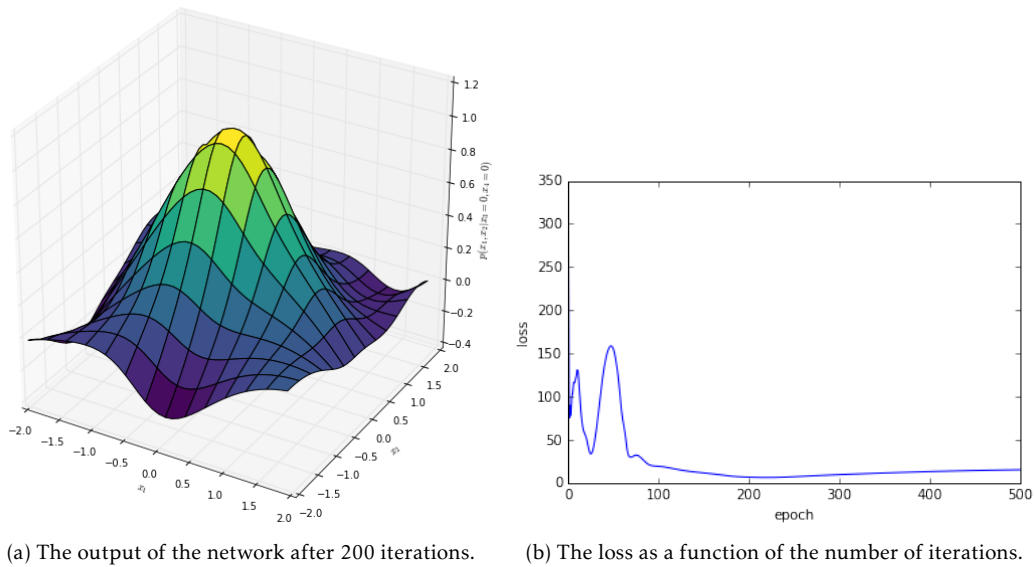
Figure 3: The training process of an MLP with $M = 8$ and $\eta = 0.1$.

Different initial weights only slightly effect the final performce of the network.

5. In Python, a little bit more work is required to plot the desired density (Figure 5).

```python
def plot_density(X, Y):
    fig = plt.figure(figsize=(10, 10), dpi=300)
    ax = fig.gca(projection='3d')
    ax.plot_surface(X[:, 0].reshape(41, 41), X[:, 1].reshape(41, 41), Y.reshape(41, 41),
                    cmap=plt.cm.viridis, rstride=1, cstride=1, linewidth=1)
    plt.xlabel('$x_1$')
    plt.ylabel('$x_2$')
    ax.set_zlabel('$p(x_1, x_2)$')
    ax.view_init(30, 230)
    plt.show()
```

6. The output from the network is quite similar to the original distribution. The major missing feature is the *"peak"* in the original distribution. This peak is probably not smooth enough for the network to represent.

The network may converge faster if the root mean squared error is used instead of simply the sum of squares. The loss will be less noisy, because it is averaged over multiple training examples. To increase the performance, it may be beneficial to use a deeper, but more narrow network. I.e., increase the number of layers, but decrease the number of units per layer. This will increase the expressive power of the network. Lastly, the performance of the network can be increased by using more data. In this case, this can simply be achieved by increasing the number of samples taken from the original distribution.
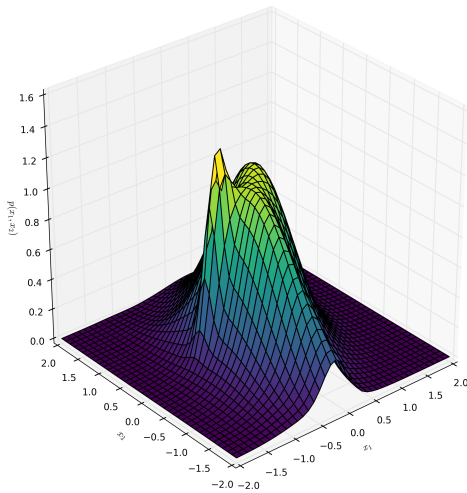
5
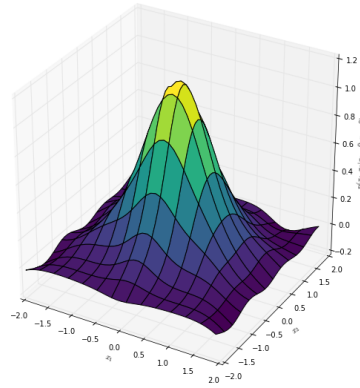
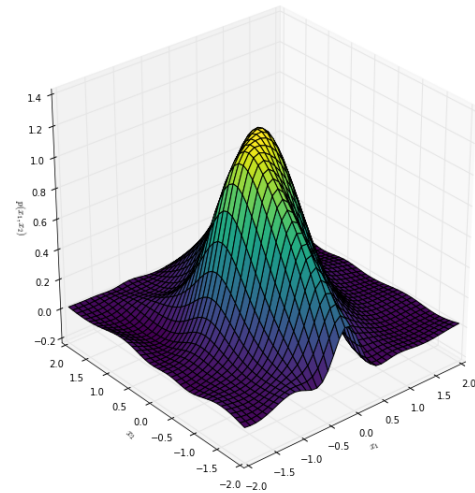(a) The final output of the network.
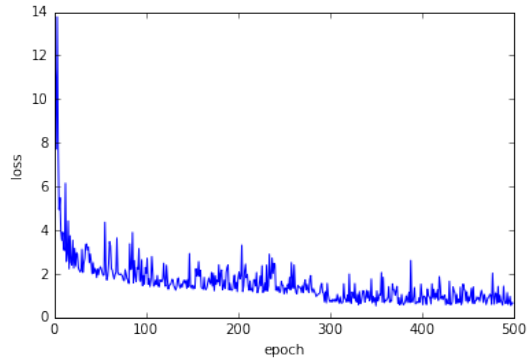
(b) The loss as a function of the number of iterations.

Figure 4: The training process of an MLP with $M = 8$ and $\eta = 0.1$ with the input randomly permuted.

(a) The multi-modal probability density to be modelled in the second part of the second exercise.

(b) The output of the network after 2000 epochs with $M = 40$ and $\eta = 0.01$

Figure 5: The original distribution and the distribution learned by an MLP.

Exercise 3 – EM and doping

1. Using the following code snippet we get the plot shown in Figure 6.

```
X = np.loadtxt('a011_mixdata.txt')
df = pd.DataFrame(X, columns=[1, 2, 3, 4])
sns.pairplot(df)
```

x_1, x_2 and x_3 seem to all be somewhat correlated. x_4 seems mostly uncorrelated with the other variables for the most part. It is hard to spot any meaningful structure with the naked eye. Figure 7 is more promising. It is possible to make out some points that show signs of doping.
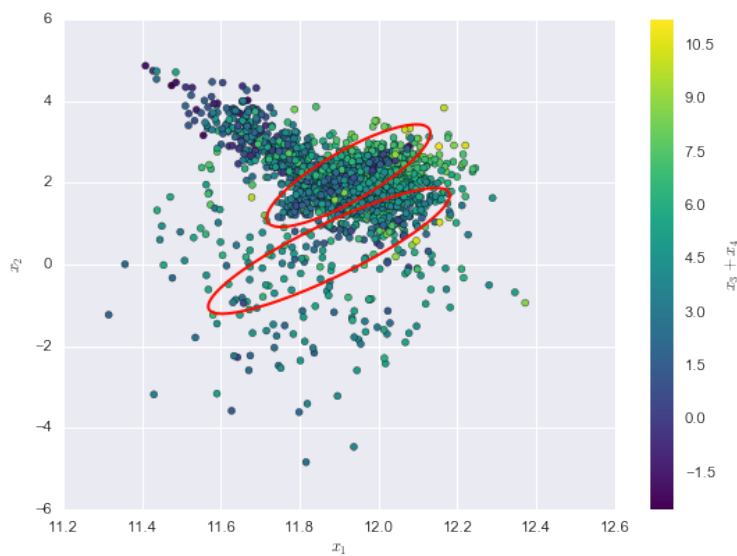

Figure 7: Scatter plot of x_1 and x_2. The points are coloured according to $x_3 + x_4$. The red ellipses indicate points which are positively correlated between x_1 and x_2, and also show some structure in $x_3 + x_4$. These could be possible cases of doping. Other simple combinations of x_3 and x_4 did not show interactions like here.

2. The code closely follows the summarization in Bishop §9.2.2. The E step consists of evaluation the responsibilities

$$\gamma(z_{nk}) = \frac{\pi_k \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}_j, \Sigma_j)}. \tag{1}$$
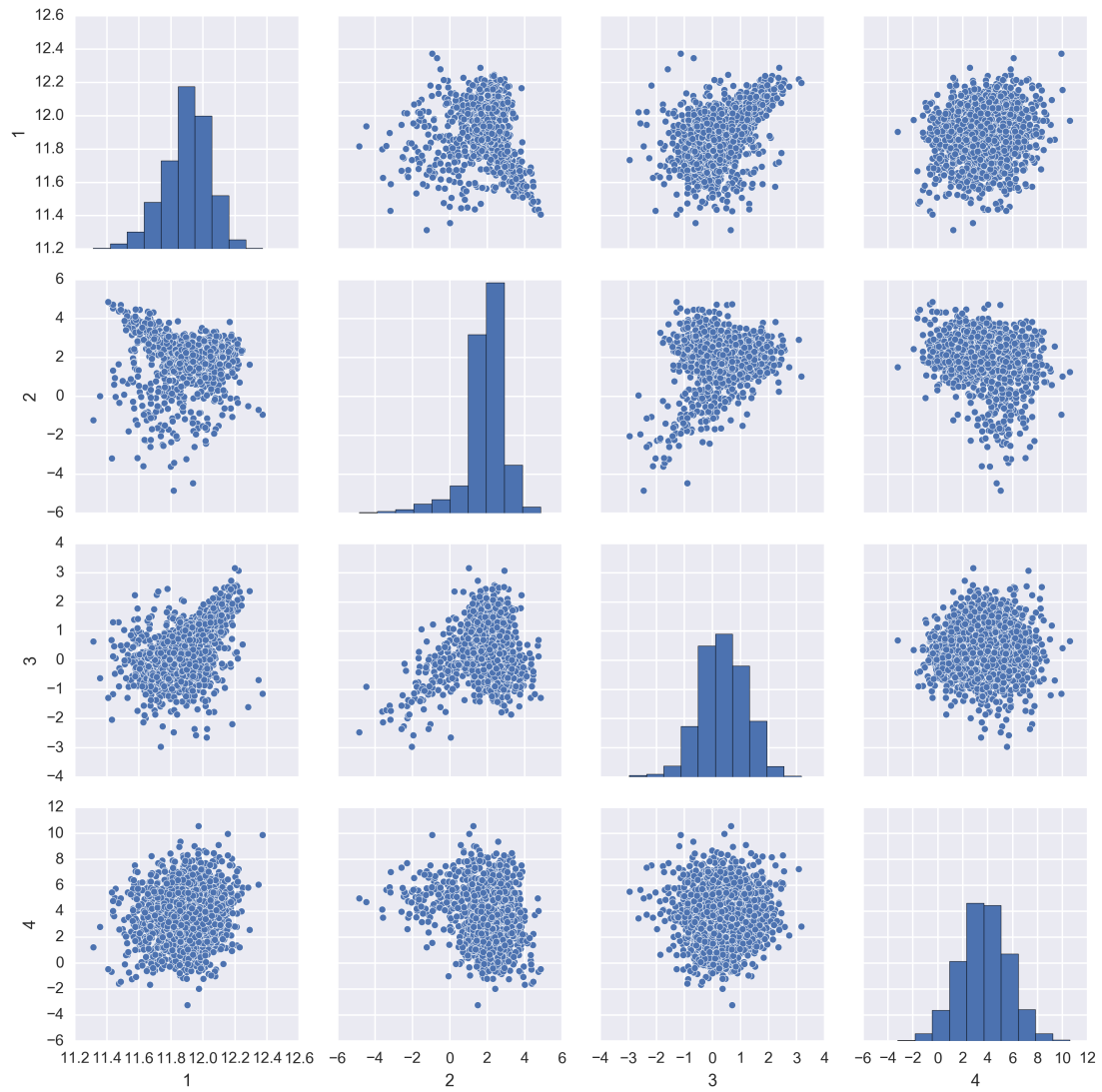
Figure 6: Scatter plots of all combinations of variables. On the diagonal a histogram of that single variable is shown.

In the M step, the parameters are re-estimated

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{2}$$

$$\boldsymbol{\mu}_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) \boldsymbol{x}_n \tag{3}$$

$$\Sigma_k^{\text{new}} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(\boldsymbol{x}_n - \boldsymbol{\mu}_k^{\text{new}})(\boldsymbol{x}_n - \boldsymbol{\mu}_k^{\text{new}})^T \tag{4}$$

$$\pi_k^{\text{new}} = \frac{N_k}{N}. \tag{5}$$

This translates to the following procedure

```python
from numpy.core.umath_tests import matrix_multiply as mm

def EM(X, K=4, max_iter=100, tol=0.01):
    # Initialize variables
    N, D = X.shape
    pis = np.ones(K) / K
    mus = np.random.uniform(-1, 1, size=(4, 4)) + X.mean(axis=0)
    covs = np.ones((K, D, D)) * (4*np.random.rand(D) + 2) * np.eye(4)

    prev_llh = 0
    for i in range(max_iter):
        # E step
        resps = np.zeros((K, N)) # responsibilities
        for k in range(K):
            resps[k] = pis[k] * multivariate_normal(mus[k], covs[k]).pdf(X)
        resps /= resps.sum(axis=0)

        # M step
        Nks = resps.sum(axis=1)[:, np.newaxis]
        mus = np.dot(resps, X) / Nks # Calculate new mus
        for k in range(K):
            diff = X - mus[k]
            sqdiff = mm(diff[:, :, np.newaxis], diff[:, np.newaxis, :])
            covs[k] = (resps[k, :, np.newaxis, np.newaxis] * sqdiff).sum(axis=0)
        covs /= Nks[:, np.newaxis]
        pis = Nks / N

        # Evaluate log likelihood
        llh = 0
        for pi, mu, cov in zip(pis, mus, covs):
            llh += pi*multivariate_normal(mu, cov).pdf(X)
        llh = np.log(llh).sum()
        print('Iteration: %d, likelihood %.4f' % (i, llh))

        if np.abs(llh - prev_llh) < tol:
            # Break if the log-likelihood hasn't improved much
            break
        prev_llh = llh

    return pis, mus, covs
```
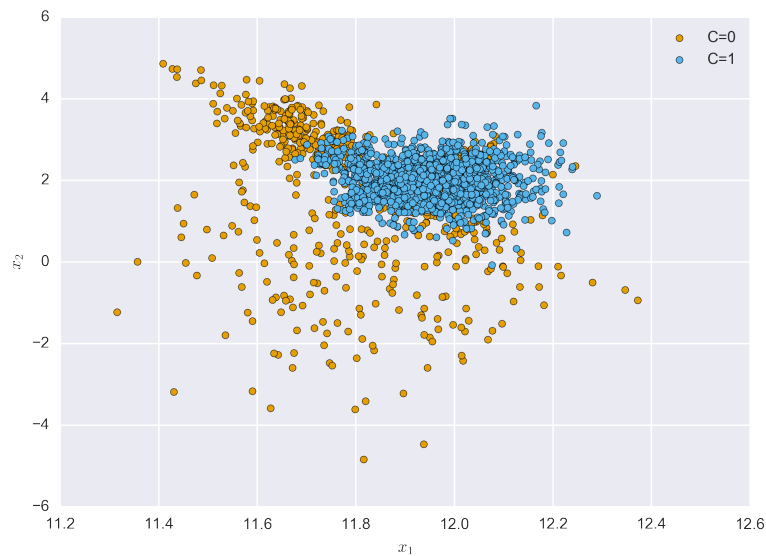
The following method creates a scatter plot of x_1 and x_2. Each point is colored by the index of the weighted gaussian that is most likely.

Figure 8: Most probable components with $K = 2$ after 26 iterations.

```python
def plot_components(X, mus, covs, pis):
    N, K = X.shape[0], mus.shape[0]
    # Evaluate the model
    probs = np.zeros((K, N))
    for k in range(K):
        probs[k] = pis[k] * multivariate_normal(mus[k], covs[k]).pdf(X)
    prediction = np.argmax(probs, axis=0)

    plt.figure()
    colors = ['#E69F00', '#56B4E9', '#F0E442', '#009E73']
    for k, color in zip(range(K), colors):
        plt.scatter(X[prediction==k, 0], X[prediction==k, 1], c=color, label='C=%d'%k)

    plt.legend()
    plt.xlabel('$x_1$')
    plt.xlabel('$x_2$')
    plt.savefig('EM.png', dpi=300, bbox_layout='tight')
    plt.show()
```

3. Using $K = 2$, the algorithm converges after 26 iterations. Running the algorithm with a different random seed resulted in swapping the means and covariances of the two gaussians with minor perturbations. The components shown in 8 show correlations of $\rho_{12} = -0.33$ $\rho_{12} = -0.05$ respectively. No positive correlation is found.

4. Increasing the number of components to $K = 3$, we find the components shown in Figure 9. At this point, the highest correlation is just 0.08, i.e. we still haven't found the cluster we are looking for. With $K = 4$ components, the results are a lot more promising. Cluster 1 displayed in Figure 10 has a correlation of 0.92. 21% of the participants most likely belong to this cluster. This means that the rumour is very accurate (maybe even on the low side).
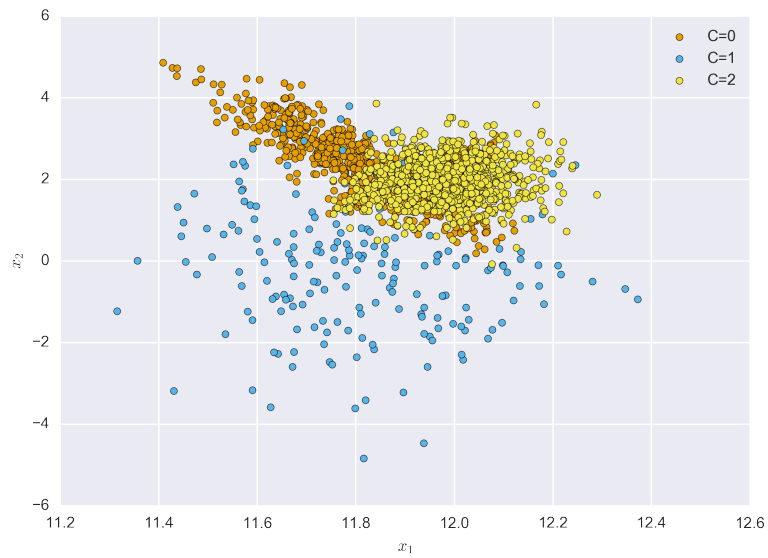
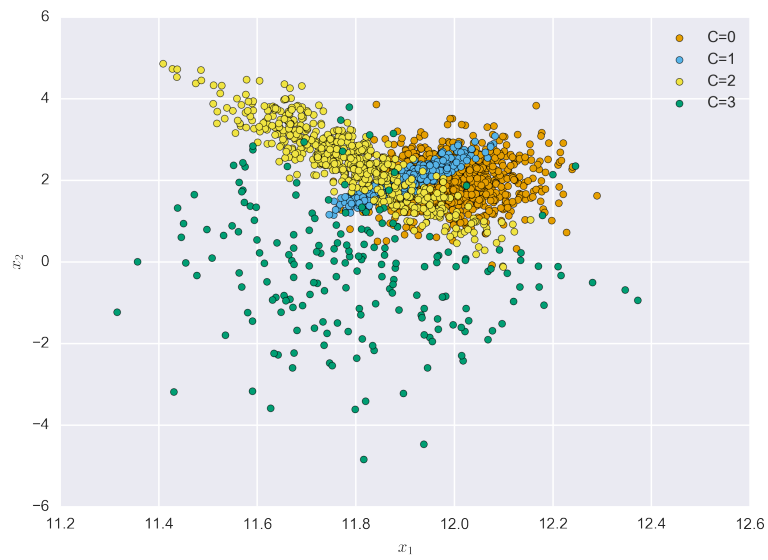Figure 9: Most probable components with $K = 3$ after 22 iterations.

Figure 10: Most probable components with $K = 4$ after 55 iterations.

5. Sample C likely belongs to component 1, which means that they have taken drug X. Finding the fraud is a bit harder: both subjects B and D have a very low likelihood (2.3×10^{-35} and 1.5×10^{-33} respectively). Subject B most likely belongs to component 2, which has a strong negative correlation. Subject D belongs to the 4th component, which is way more sparse. My guess would be that subject B is the fraud, based on the likelihood of the sample.

Exercise 4 – Handwritten digit recognition