Statistical Machine Learning: Assignment 2

Joris van Vugt, s4279859

November 6, 2016

Exercise 1 – Sequential learning

Part 1 – Obtaining the prior

1. First we compute the precision matrix using `numpy.linalg.inv`:

$$\tilde{\mathbf{\Lambda}} = \tilde{\mathbf{\Sigma}}^{-1} = \left(\begin{array}{cc|cc} 60 & 50 & -48 & 38 \\ 50 & 50 & -50 & 40 \\ \hline -48 & -50 & 52.4 & -41.4 \\ 38 & 40 & -41.4 & 33.4 \end{array}\right) = \left(\begin{array}{c|c} \tilde{\mathbf{\Lambda}}_{aa} & \tilde{\mathbf{\Lambda}}_{ab} \\ \tilde{\mathbf{\Lambda}}_{ba} & \tilde{\mathbf{\Lambda}}_{bb} \end{array}\right)$$

   Now, using equations 2.73 and 2.75 from Bishop, we can compute the conditional covariance:

$$\mathbf{\Sigma}_p = \tilde{\mathbf{\Lambda}}_{aa}^{-1} = \begin{pmatrix} 0.1 & -0.1 \\ -0.1 & 0.12 \end{pmatrix}$$

   and the conditional mean:

$$\begin{aligned} \boldsymbol{\mu}_p &= \boldsymbol{\mu}_a - \tilde{\mathbf{\Lambda}}_{aa}^{-1}\tilde{\mathbf{\Lambda}}_{ab}(\boldsymbol{x}_b - \boldsymbol{\mu}_b) \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.1 & -0.1 \\ -0.1 & 0.12 \end{pmatrix} \begin{pmatrix} -48 & 38 \\ -50 & 40 \end{pmatrix} \left[\begin{pmatrix} 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right] \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.1 & -0.1 \\ -0.1 & 0.12 \end{pmatrix} \begin{pmatrix} -48 & 38 \\ -50 & 40 \end{pmatrix} \begin{pmatrix} -1 \\ -2 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.2 & -0.2 \\ -1.2 & 1 \end{pmatrix} \begin{pmatrix} -1 \\ -2 \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.2 \\ -0.8 \end{pmatrix} \\ &= \begin{pmatrix} 0.8 \\ 0.8 \end{pmatrix} \end{aligned}$$

2.
```python
def generate_pair():
    return np.random.multivariate_normal([0.8, 0.8],
                                         [[0.1, -0.1],
                                          [-0.1, 0.12]])
```

$$\boldsymbol{\mu}_t = \begin{pmatrix} 0.28 \\ 1.18 \end{pmatrix}$$

3. To calculate the probability density of our multivariate Gaussian random variable, we use `scipy.stats.multivariate_normal` and its `pdf` method.

```python
x, y = np.mgrid[-0.25:2.25:.01, -1:2:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x
pos[:, :, 1] = y
mu_p = [0.8, 0.8]
cov_p = [[0.1, -0.1], [-0.1, 0.12]]
z = multivariate_normal(mu_p, cov_p).pdf(pos)

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(x, y, z, cmap=plt.cm.viridis)
# More plot formatting code
```

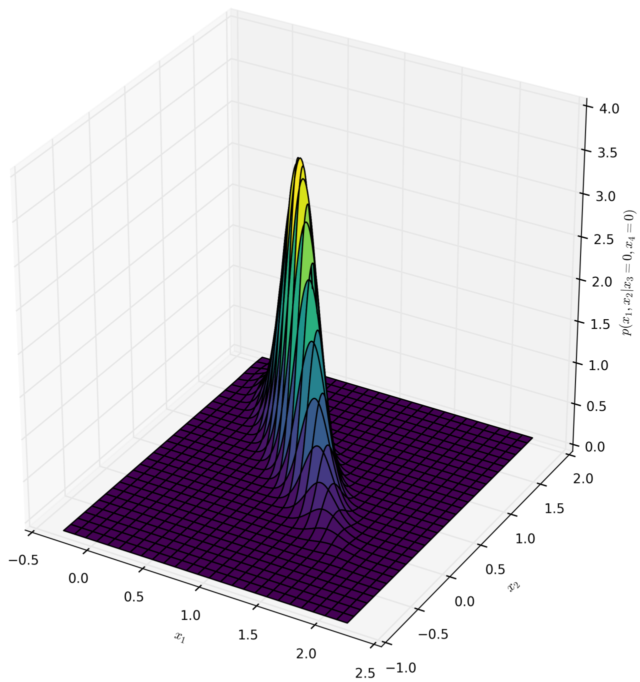

Figure 1: Probability density plot of the multivariate Gaussian (Ex 1.1.3)

Part 2 – Generating the data

1.
```python
N = 1000
data = np.random.multivariate_normal([0.28, 1.18],
                                     [[2.0, 0.8],
                                      [0.8, 4.0]],
                                     N)
np.savetxt('data.txt', data)
```

2. The maximum likelihood estimate of the mean is simply the mean of the observed data:

$$\boldsymbol{\mu}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{x}_n = \begin{pmatrix}0.25\\1.21\end{pmatrix}$$

Computing the maximum likelihood estimate of the covariance is slightly more involved:

$$\boldsymbol{\Sigma}_{\mathrm{ML}} = \frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\boldsymbol{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^T = \begin{pmatrix}2.023 & 0.828\\0.828 & 3.626\end{pmatrix}$$

To compute the unbiased maximum likelihood covariance estimate, we normalize by $N-1$ instead of N:

$$\boldsymbol{\Sigma}_{\mathrm{ML}} = \frac{1}{N-1}\sum_{n=1}^{N}(\boldsymbol{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})(\boldsymbol{x}_n - \boldsymbol{\mu}_{\mathrm{ML}})^T = \begin{pmatrix}2.025 & 0.829\\0.829 & 3.629\end{pmatrix}$$

These results were obtained with the following code:

```python
mu_ml = data.mean(axis=0)
x = data - mu_ml
cov_ml = np.dot(x.T, x) / N
cov_ml_unbiased = np.dot(x.T, x) / (N - 1)
```

Note that the left factor is transposed instead of the right factor in the covariance estimates. This is because our points are row vectors instead of column vectors (i.e., `data` has shape $N \times 2$).

We can compare our estimates to the true statistics:

$$\boldsymbol{\mu}_t = \begin{pmatrix}0.28\\1.18\end{pmatrix} \qquad \boldsymbol{\Sigma}_t = \begin{pmatrix}2.0 & 0.8\\0.8 & 4.0\end{pmatrix}$$

Our estimates are pretty close to their true values. The unbiased estimate of the covariance is not much closer than the biased estimate. Because N is relatively high, the slight change in normalization does not have much effect.

Part 3 – Sequential learning algorithms

1. Using equation 2.126 from Bishop

$$\boldsymbol{\mu}_{\mathrm{ML}}^{(N)} = \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)} + \frac{1}{N}(\boldsymbol{x}_N - \boldsymbol{\mu}_{\mathrm{ML}}^{(N-1)})$$

we can come up with a Python procedure for sequential learning:

```python
def seq_ml(data):
    mus = [np.array([[0], [0]])]
    for i in range(N):
        x_n = data[i].reshape(2, 1)
        mu_n = mus[-1] + (x_n-mus[-1]) / (i + 1)
        mus.append(mu_n)
    return mus
```

The starting value $\boldsymbol{\mu}_{\mathrm{ML}}^{(0)}$ does not matter, but is arbitrarily set to $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$. We divide by $i+1$ instead of just i, because Python uses 0-based indexing.

2. We can use equation 2.113 from Bishop[1]

$$P(\boldsymbol{\mu}|D_{n-1}) = \mathcal{N}(\boldsymbol{\mu}|\boldsymbol{\mu}_{(n-1)}, \boldsymbol{\Sigma}_{(n-1)})$$

and equation 2.114 from Bishop[2]

$$P(\boldsymbol{x}_n|\boldsymbol{\mu}, D_{n-1}) = \mathcal{N}(\boldsymbol{x}_n|\boldsymbol{\mu}, \boldsymbol{\Sigma}_t).$$

Now we can apply equation 2.116 from Bishop to find out the posterior

$$P(\boldsymbol{\mu}|\boldsymbol{x}_n D_{n-1}) = \mathcal{N}(\boldsymbol{\mu}|\boldsymbol{\Sigma}\{\boldsymbol{\Sigma}_t^{-1}\boldsymbol{x}_n + \boldsymbol{\Sigma}_{(n-1)}^{-1}\boldsymbol{\mu}_{(n-1)}\}, \boldsymbol{\Sigma}).$$

Equation 2.117 gives us

$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_{(n-1)}^{-1} + \boldsymbol{\Sigma}_t^{-1})^{-1}.$$

The sequential learning rules for the mean and the covariance are thus

$$\boldsymbol{\mu}_{(n)} = \boldsymbol{\Sigma}_{(n)}\{\boldsymbol{\Sigma}_t^{-1}\boldsymbol{x}_n + \boldsymbol{\Sigma}_{(n-1)}^{-1}\boldsymbol{\mu}_{(n-1)}\}$$
$$\boldsymbol{\Sigma}_{(n)} = (\boldsymbol{\Sigma}_{(n-1)}^{-1} + \boldsymbol{\Sigma}_t^{-1})^{-1}$$

3.

```python
def seq_map(data, mu_p, cov_p, cov_t):
    mus, covs = [mu_p], [cov_p]
    for x in data:
        x_n = x.reshape(2, 1)
        cov_n = np.linalg.inv(np.linalg.inv(covs[-1]) + np.linalg.inv(cov_t))
        mu_n = cov_n.dot(np.linalg.inv(cov_t).dot(x_n) + np.linalg.inv(covs[-1]).dot(mus[-1]))
        mus.append(mu_n)
        covs.append(cov_n)

    return mus, covs
```

[1] With mappings $\boldsymbol{x} = \boldsymbol{\mu}$, $\boldsymbol{\mu} = \boldsymbol{\mu}_{(n-1)}$ and $\boldsymbol{\Lambda}^{-1} = \boldsymbol{\Sigma}_{(n-1)}$

[2] With mappings $\boldsymbol{y} = \boldsymbol{x}_n$, $\boldsymbol{A} = \boldsymbol{I}$, $\boldsymbol{x} = \boldsymbol{\mu}$, $\boldsymbol{b} = 0$ and $\boldsymbol{L}^{-1} = \boldsymbol{\Sigma}_t$

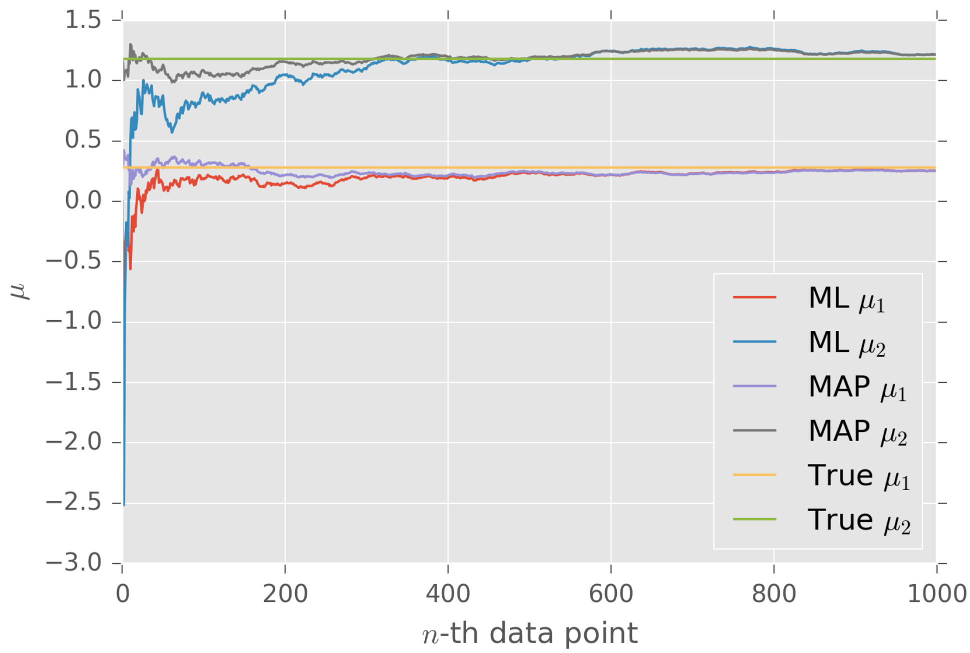

Figure 2: The ML and MAP estimates for both components of μ as a function of the number of data points observed. (Ex 1.3.4)

4. Both the ML and MAP estimates converge to the true values quite quickly. Because a reasonable prior is chosen, the MAP converges a bit faster. After enough data points, the effect of the prior becomes negligable and the ML and MAP estimates are roughly equal.

Exercise 2 – The faulty lighthouse

Part 1 – Constructing the model

1. The photo-detectors are located on the coast and thus only half of the angles can be observed, namely those towards the coast. A full circle corresponds to 2π. We can only observe half a circle, so $p(\theta_k|\alpha, \beta) = \frac{1}{\pi}$ indicates a uniform distribution over all possible angles. We can show that this distribution integrates to 1 and is thus a valid probability distribution

$$\begin{aligned}\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \frac{1}{\pi} &= \frac{x}{\pi}\Big|_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \\ &= \frac{\frac{\pi}{2} + \frac{\pi}{2}}{\pi} = 1.\end{aligned}$$

2. Rewriting equation 7 gives

$$\theta_k = \tan^{-1}(\frac{x_k - \alpha}{\beta}).$$

We also need the Jacobian

$$\begin{aligned}
|\frac{\mathrm{d}\theta}{\mathrm{d}x}| &= \frac{1}{1+(\frac{x_k-\alpha}{\beta})^2} \times \frac{\beta}{\beta^2} \\
&= \frac{\beta}{\beta^2+\beta^2(\frac{x_k-\alpha}{\beta})^2} \\
&= \frac{\beta}{\beta^2+(x_k-\alpha)^2}
\end{aligned}$$

multiplying these gives the expected distribution over x_k

$$\begin{aligned}
p(x_k|\alpha,\beta) &= p(\theta_k)\left|\frac{\mathrm{d}\theta}{\mathrm{d}x}\right| \\
&= \frac{1}{\pi} \times \frac{\beta}{\beta^2+(x_k-\alpha)^2} \\
&= \frac{\beta}{\pi[\beta^2+(x_k-\alpha)^2]}
\end{aligned}$$

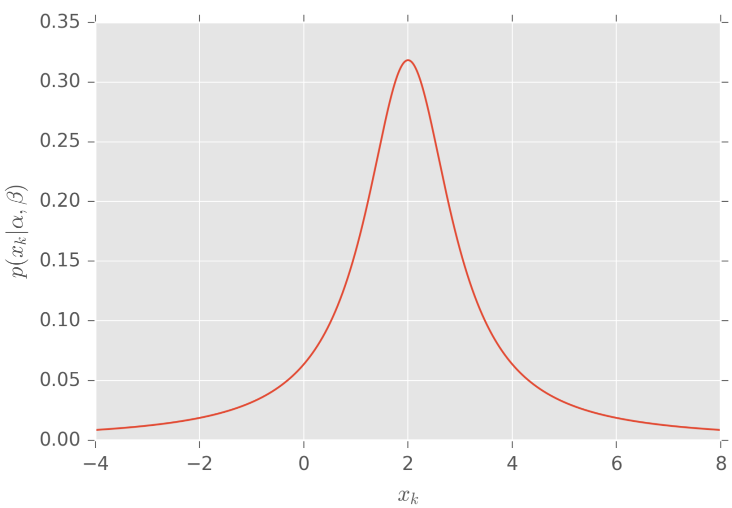

Figure 3: The probability distribution $p(x_k|\alpha=2,\beta=1)$ (Ex 2.2.2)

```
def p_xk(x, alpha, beta):
    return beta / (np.pi * (beta**2 + (x-alpha)**2))

x = np.linspace(-4, 8, num=1000)
probs = p_xk(x, 2, 1)
plt.plot(x, probs)
```

3. According to $p(\alpha|\mathcal{D},\beta) \propto p(\mathcal{D}|\alpha,\beta)p(\alpha|\beta)$, we just need to calculcate $p(\mathcal{D}|\alpha,\beta)$, since $p(\alpha|\beta)$ does not depend on the data. $p(\mathcal{D}|\alpha,\beta)$ is simply the product of the probability of each data point. We can now calculate log of the posterior density

$$
\begin{aligned}
L &= \ln(p(\alpha|\mathcal{D},\beta)) \\
&= \ln(\prod_{k=1}^{N} \frac{\beta}{\pi[\beta^2 + (x_k - \alpha)^2]}) \\
&= \sum_{k=1}^{N} \ln\left(\frac{\beta}{\pi} \times \frac{1}{\beta^2 + (x_k - \alpha)^2}\right) \\
&= N \ln \frac{\beta}{\pi} - \sum_{k=1}^{N} \ln[\beta^2 + (x_k - \alpha)^2] \\
&= \mathit{constant} \ - \sum_{k=1}^{N} \ln[\beta^2 + (x_k - \alpha)^2]
\end{aligned}
$$

and an expression for maximizing this density

$$
\hat{\alpha} = \arg\max_{a} \sum_{k=1}^{N} \ln[\beta^2 + (x_k - \alpha)^2]
$$

Figure 4: Probability density of α (Ex 2.1.4)

4.

```
def p_a(x, alpha, beta):
    return np.product(beta / (np.pi * beta**2 + (x-alpha)**2))

D = np.array([4.8, -2.7, 2.2, 1.1, 0.8, -7.3])
alphas = np.linspace(-5, 5, num=1000)
beta = 1
likelihoods = [p_a(D, alpha, beta) for alpha in alphas]
plt.plot(alphas, likelihoods)
```

The mean of the data is -0.183, but the maximum likelood is $\hat{\alpha} \approx 1.18$. The discrepancy might be the result of the small size of the data set.

Part 2 – Generate the lighthouse data

1.
```
alpha_t = np.random.uniform(0, 10)
beta_t = np.random.uniform(1, 2)
```

   We have $\alpha_t = 6.03$ and $\beta_t = 1.76$.

2. Given an angle, we need to calculate the position it will be detected at

$$\beta \tan(\theta) = x - \alpha$$
$$x = \beta \tan(\theta) + \alpha$$

```
def location(angle, alpha, beta):
    return beta * np.tan(angle) + alpha

N = 200
angles = np.random.uniform(-np.pi/2, np.pi/2, N)
locations = [location(angle, alpha_t, beta_t) for angle in angles]
```

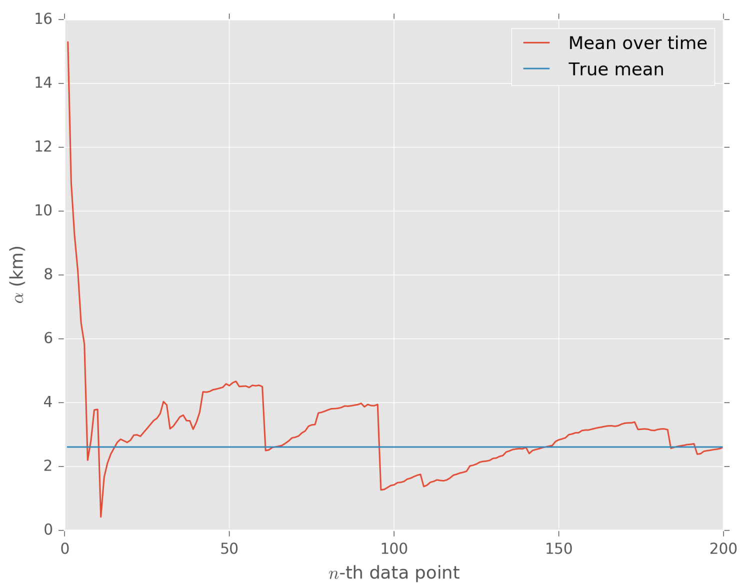

Figure 5: Mean of x as a function of the number of datapoints (Ex 2.2.3)

3. The mean of the data is 2.61. This is not close to the true position of the lighthouse ($\alpha = 6.03$). The mean over time is at the beginning also quite far off, but starts getting closer towards the end. I reckon 150 points is a good lower bound for the necessary number of data points.

Part 3 – Find the lighthouse

1.

$$
\begin{aligned}
p(\mathcal{D}|\alpha,\beta) &= \prod_{k=1}^{N} p(x_k|\alpha,\beta) \\
\ln p(\mathcal{D}|\alpha,\beta) &= \ln \prod_{k=1}^{N} p(x_k|\alpha,\beta) \\
&= \sum_{k=1}^{N} \ln p(x_k|\alpha,\beta) \\
&= \sum_{k=1}^{N} \ln \frac{\beta}{\pi[\beta^2+(x_k-\alpha)^2]} \\
&= N \ln \frac{\beta}{\pi} - \sum_{k=1}^{N} \ln[\beta^2+(x_k-\alpha)^2]
\end{aligned}
$$

This is similar to Exericise 2.1.3

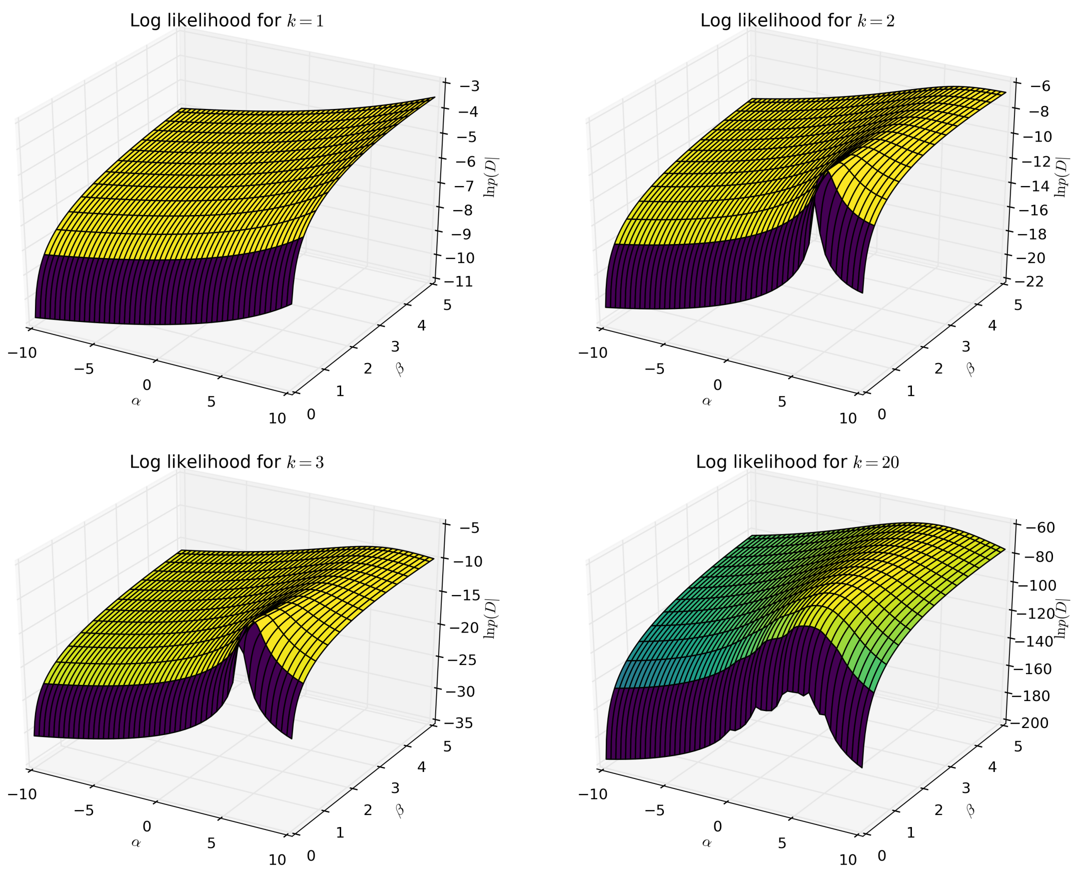

Figure 6: Log likelihood for α and β for different numbers of datapoints (Ex 2.3.2)

2.

```python
ks = [1, 2, 3, 20]
alphas, betas = np.mgrid[-10:10:0.04, 0:5:0.04]
# alphas, betas = np.meshgrid(np.linspace(-10, 10, num=500), np.linspace(0, 5, num=250))
for k in ks:
    x = locations[:k]
    # We only have to calculate the constant once
    likelihood = k * np.log(betas/np.pi)
    for loc in x:
        likelihood -= np.log(betas**2 + (loc - alphas)**2)

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_surface(alphas, betas, likelihood, cmap=plt.cm.viridis)
    # And some more plot formatting code
```

With few datapoints, it is not very likely that de lighthouse is close to the coast. This is because a lighthouse really close to the coast would result in few data points close to that position. As more points come in, this does become more likely. We also see that the likelihood starts increasing around a_t as more points are considered.

The log-likelihood does not suffer from numerical precision errors after multiplication of a lot of small numbers. The expression may also be easier to compute. However, it is not a probability distrubition and can thus not directly be compared with other distributions.

3.

```python
from scipy.optimize import fmin

def log_likelihood(params, locations):
    """
    This function will be minimized.
    This is why it returns -likelihood
    """
    alpha, beta = params
    likelihood = len(locations) * np.log(beta/np.pi)
    for loc in locations:
        likelihood -= np.log(beta**2 + (loc - alpha)**2)
    return -likelihood

def plot_maximize_logl(data):
    alphas, betas = [], []
    x = np.arange(len(data))
    for k in x:
        [alpha, beta] = fmin(log_likelihood, (0, 1), args=(data[:k],))
        alphas.append(alpha)
        betas.append(beta)

    return alphas, betas
```

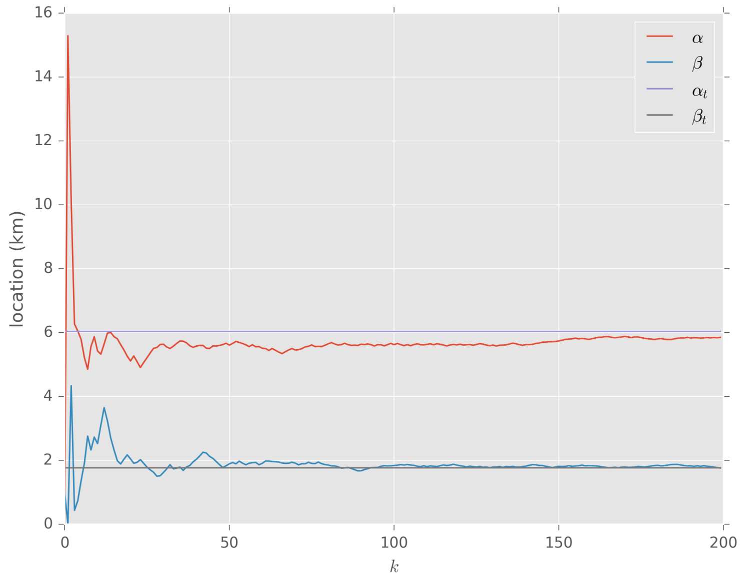

Figure 7: The maximum likelihood estimate of α and β as a function of the number of data points. (Ex 2.4.3)

When considering all 200 data points, the point $\alpha = 5.85$ and $\beta = 1.76$ yield the highest likelihood. This is close to the real position ($\alpha_t = 6.03$ and $\beta_t = 1.76$). The estimates converge to their true values relatively fast (a lot faster than I estimated in 2.2.3)