Compulsory exercise - STK4051

Øystein Høistad Bruce

February 2022

UiO **: Universitetet i Oslo**

Department of Mathematics

Contents

1 Exercise 1 (Lp-regularization)

Simplified regression model:

$$y_i = \beta_i + \epsilon_i \quad i = 1, ..., n \tag{1}$$

where y_i is the data, β_i are the parameters, and $\epsilon_i \sim \mathcal{N}(0, 1^2)$ is an error term. The probability distribution of ϵ_i is:

$$p(\epsilon_i) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\epsilon_i^2/2\right\} \qquad -\infty < \epsilon_i < \infty \tag{2}$$

1.a Maximum likelihood estimator

Derive the maximum likelihood estimator for β_i, i = 1, ..., n

The probability of getting y_i (given current estimates on beta) is the probability that a random sample from the standard normal distribution to be $y_i - \beta_i$. The likelihood can then be written as:

$$L(\theta|y) = \prod_{i=1}^{n} p(y_i - \beta_i)$$

further, we get the following expression for the log-likelihood:

$$l(\theta|y) = \log\left\{L(\theta|y)\right\} = \log\left\{\prod_{i=1}^{n} p(y_i - \beta_i)\right\}$$

$$= \sum_{i=1}^{n} \log\{p(y_i - \beta_i)\}$$

$$= \sum_{i=1}^{n} -\log\{\sqrt{2\pi}\} + \frac{(y_i - \beta_i)^2}{2}$$

Now, we are ready to find the maximum likelihood estimator for β_i, by finding the derivative of the log-likelihood function w.r.t. β_i and set it equal to zero.

$$\frac{\partial l(\theta|y)}{\partial \beta_i} = \frac{2(y_i - \beta_i)}{2} = (y_i - \beta_i) \overset{!}{=} 0$$

The maximum likelihood estimator, will then be the following:

$$\hat{\beta}_i = y_i \qquad i = 1, ..., n \tag{3}$$

1.b Penalized least squares

An alternative estimator is derived using a penalized least squares, by solving the optimization problem:

$$\min_{\beta} \left\{-l(\beta|y) + \frac{\gamma}{p}\|\beta\|_p^p\right\} \tag{4}$$

where l is the log-likelihood, γ is a regularization parameter and

$$\|\beta\|_p^p = \sum_{i=1}^{n} |\beta_i|^p$$

Show that

$$f_{p,\gamma}(\beta_i) = \beta_i + \gamma \cdot \text{sign}(\beta_i)|\beta_i|^{p-1} \tag{5}$$

where $f_{p,y}(\beta_i)$ should satisfy

$$f_{p,y}(\beta_i) = y_i \quad i = 1, ..., n \tag{6}$$

Answer

We will first find the derivative of the expression we want to minimize w.r.t. β_i, then set the expression to 0.

$$\frac{\partial}{\partial \beta_i}\left(-l(\beta|y) + \frac{\gamma}{p}\|\beta\|_p^p\right) = -(y_i - \beta_i) + \frac{\gamma}{p}\, p|\beta_i|^{p-1}\frac{\partial|\beta_i|}{\beta_i}$$

$$= -y_i + \beta_i + \gamma|\beta_i|^{p-1}\frac{\partial|\beta_i|}{\beta_i}$$

$$= -y_i + \beta_i + \gamma|\beta_i|^{p-1}\,\text{sign}(\beta_i) \overset{!}{=} 0$$

Then, we solve for y_i and get the result we were gonna prove:

$$y_i = \beta_i + \gamma|\beta_i|^{p-1}\,\text{sign}(\beta_i) \qquad i = 1, ..., n$$

1.c Plotting the function

Let $\gamma = \{1, 0.2\}$, plot the function $f_{p,\gamma}(\beta)$ for $p = \{1.1, 2, 5, 100\}$ on the interval $[-5, 5]$. Give an interpretation of the result. The code for doing this is attached in the appendix section 6.2.
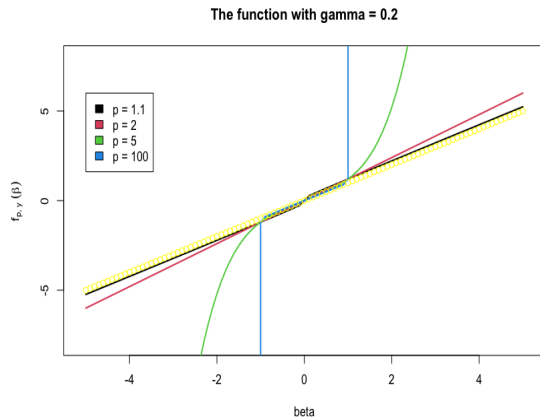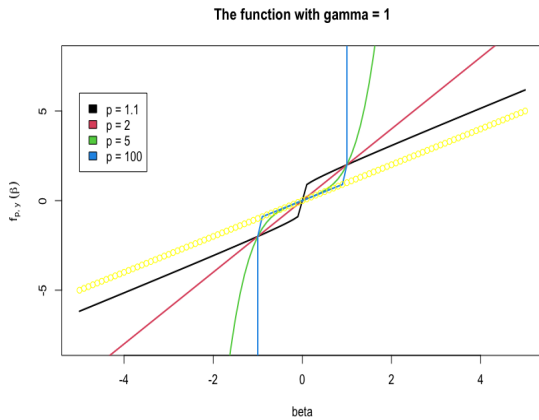

Figure 1

Figure 2

where the yellow line is $f(\beta) = \beta$

Plot the inverse function by flipping the order of the arguments in the plotting function.
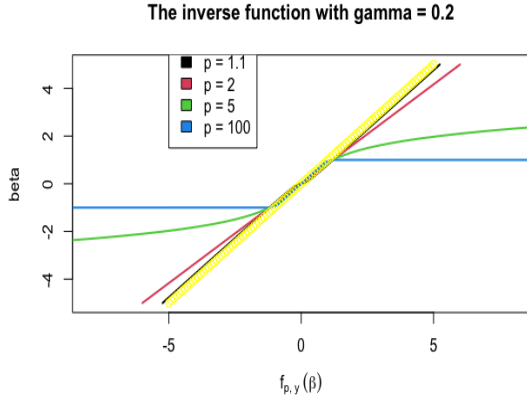
Figure 3
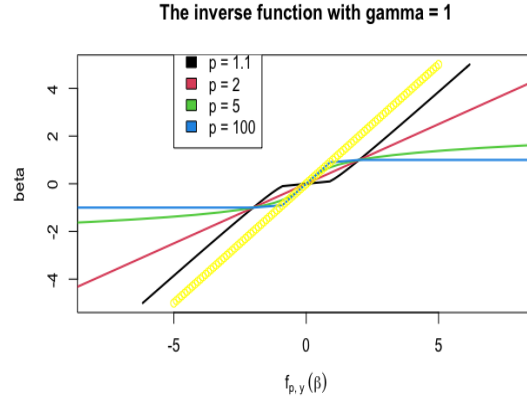
Figure 4

By the figures 1, 2, 3 and 4 we can see that an increasing p value will result in more bounded β value (lower than the actual value y_i) - more or less a penalty on the beta term. The γ term describes how sensitive beta should be to changes in the $y_i's$ it should be to the solutions. A huge change in y_i with a big γ would result in a minimal change for the β.

1.d Bisection method

Implement a function which finds the root of expression (6). The script of doing this is added in the appendix section 6.3. We will find the optimal β values for each $y \in [-5, 5]$ with a model of $\gamma = 1$ and $p = \{1.1, 2, 100\}$. In general, good starting values for the bisection method will be s.t. the function values (of the function we want to find the root of) are one negative and one positive. Then, you are guaranteed a root of the function.

Here are solutions of β given some y values:
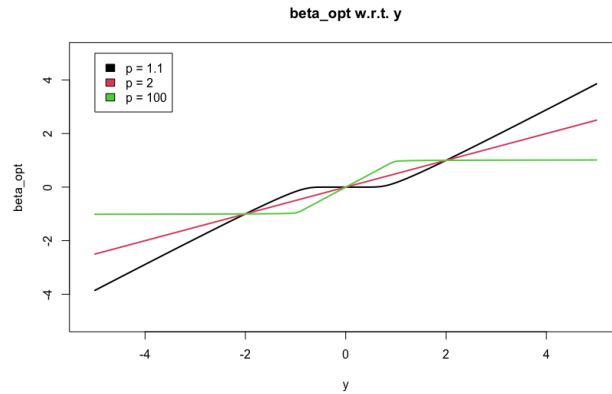
Figure 5: solution of β given y with bisection method

1.e Use data on the results

Now, we are ready for doing the different estimations with actual data. The data we are given are y, and the ground truth values for beta, β_{GT}. The script are attached in the section 6.4. The results are also achieved from it.

Firstly, we will look at the penalized regression. We want to compute $\hat{\beta}_{\gamma,p}$ with $\gamma = 1, p = \{1.1, 2, 100\}$ and get the following residual sum of squares:

| p-values	penalized regression	Alternative method
1.1	273.97	5785.65
2	1256.53	2425.73
100	3786.62	333.37

where the alternative method is another way of achieve an estimater. That is:

$$\hat{\beta}_{1,100}^{\text{Alt}} = y - \hat{\beta}_{1,100}$$

For the MLE estimator, we get the residual sum of squares to be 998.42.

So, in this case the penalized regression method will get less residual sum of squares, and will then be the best estimators of all of the methods. Penalized regression with $p = 1.1$ (given that $\gamma = 1$) is the best option (of the ones we compared), then secondly is the alternative method with $p = 100$.

1.f ADMM algorithm

With the ADMM we want to minimize an expression with some constraint. In our case, we want to minimize the following expression:

$$\min_{\beta} \{-l(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}) + \frac{\gamma}{p}\|\boldsymbol{\beta}\|_p^p\}$$

which can be transferred, to the following problem:

$$\min_{\beta, z} \{-l(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}) + \frac{\gamma}{p}\|\mathbf{z}\|_p^p\}$$
$$\text{subject to} \quad \boldsymbol{\beta} - \mathbf{z} = 0$$

Form it to the format of an ADMM problem (remove the "subject to" part)

$$\min_{\beta, z} \{-l(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}) + \frac{\gamma}{p}\|\mathbf{z}\|_p^p + \lambda^T(\boldsymbol{\beta} - \mathbf{z}) + \frac{\phi}{2}\|\boldsymbol{\beta} - \mathbf{z}\|_2^2\}$$

The algorithm goes as follows:

1. $\boldsymbol{\lambda}^{(0)} = \mathbf{0}, \boldsymbol{z}^{(0)} = \mathbf{0}$

2. Iterate by increasing i, until the defined convergence property

    (a) $\boldsymbol{\beta}^{(i)} = \text{argmin} \left[-l(\boldsymbol{\beta}|\mathbf{y}, \mathbf{X}) + \lambda^{(i-1)T}(\boldsymbol{\beta} - \boldsymbol{z}^{(i-1)}) + \frac{\phi}{2}\|\boldsymbol{\beta} - \boldsymbol{z}^{(i-1)}\|_2^2 \right]$

    (b) $\boldsymbol{z}^{(i)} = \text{argmin} \left[\frac{\gamma}{p}\|\mathbf{z}\|_p^p + \lambda^{(i-1)T}(\boldsymbol{\beta}^{(i)} - \boldsymbol{z}) + \frac{\phi}{2}\|\boldsymbol{\beta}^{(i)} - \boldsymbol{z}\|_2^2 \right]$

    (c) $\boldsymbol{\lambda}^{(i)} = \boldsymbol{\lambda}^{(i-1)} + \phi(\boldsymbol{\beta}^{(i)} - \boldsymbol{z}^{(i)})$

2 Exercise 2 (EM-algorithm)

In this exercise, we will be using the data in the file *sparseDataWithErrors.dat*

For this exercise we will assume that $Y_i \quad i = 1, ..., n$ are independent and identically distributed according to the mixture distribution:

$$f(y_i) = p \cdot \phi(y_i; 0, 1^2) + (1 - p) \cdot \phi(y_i; 0, \tau^2 + 1^2) \tag{7}$$

where $\phi(y_i; \mu, \sigma^2)$ is the normal density with mean μ and variance σ^2. We will now consider estimation of the parameters $\theta = (p, \tau^2)$.

2.a   Expression for the likelihood of θ

The likelihood of θ can be expressed as follows:

$$L(\theta|y) = \prod_{i=1}^{n} f(y_i)$$

$$= \prod_{i=1}^{n} \left[p \cdot \phi(y_i; 0, 1^2) + (1-p) \cdot \phi(y_i; 0, \tau^2 + 1^2) \right]$$

2.b   Complete log-likelihood

Introduce the variable C_i which identifies the model y_i belongs to. Give an expression for the complete log-likelihood using the pairs $(C_i, y_i)_{i=1}^{n}$

We know that we can belong to either one of the two classes $\{0, 1\}$. Let class 0 be the data from the normal distribution with variance 1, and class 1 be the other one.

Starting up with an expression for the complete likelihood:

$$L_{\text{comp}}(\theta|(C, y)) = \prod_{i=1}^{n} \left[\mathbb{1}_{c_i=0} \cdot \phi(y_i; 0, 1^2) + \mathbb{1}_{c_i=1} \cdot \phi(y_i; 0, \tau^2 + 1^2) \right]$$

Then we get the following for the complete log-likelihood:

$$l_{\text{comp}}(\theta|(C, y)) = \log\{L_{\text{comp}}(\theta|(C, y))\}$$

$$= \sum_{i=1}^{n} \left[\mathbb{1}_{c_i=0} \log\{\phi(y_i; 0, 1^2)\} + \mathbb{1}_{c_i=1} \log\{\phi(y_i; 0, \tau^2 + 1^2)\} \right]$$

$$= \sum_{i=1}^{n} \left[\mathbb{1}_{c_i=0} \left(-\log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right) + \mathbb{1}_{c_i=1} \left(-\log(\sqrt{2\pi} \cdot \sqrt{\tau^2 + 1}) - \frac{y_i^2}{2(\tau^2 + 1)} \right) \right]$$

2.c   Expression for $Q(\theta, \theta^{(t)})$

Give an expression for $Q(\theta, \theta^{(t)})$, what is the interpretation of $Q(\theta, \theta^{(t)})$. Derive the estimates for $\theta = (p, \tau^2)$, using $Q(\theta, \theta^{(t)})$.

$$Q(\theta, \theta^{(t)}) = E[l_{\text{comp}}|y, \theta^{(t)}]$$

$$= E\left[\sum_{i=1}^{n} \left(\mathbb{1}_{c_i=0} \left(-\log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right) + \mathbb{1}_{c_i=1} \left(-\log(\sqrt{2\pi} \cdot \sqrt{\tau^2 + 1}) - \frac{y_i^2}{2(\tau^2 + 1)} \right) \right) \right]$$

$$\overset{1}{=} \sum_{i=1}^{n} \left(E[\mathbb{1}_{c_i=0}|y, \theta^{(t)}] \left(-\log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right) + E[\mathbb{1}_{c_i=1}|y, \theta^{(t)}] \left(-\log(\sqrt{2\pi} \cdot \sqrt{\tau^2 + 1}) - \frac{y_i^2}{2(\tau^2 + 1)} \right) \right)$$

$$\overset{2}{=} \sum_{i=1}^{n} \left(p_i \left(-\log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right) + (1 - p_i) \left(-\log(\sqrt{2\pi} \cdot \sqrt{\tau^2 + 1}) - \frac{y_i^2}{2(\tau^2 + 1)} \right) \right)$$

$$= \sum_{i=1}^{n} \left(p_i \left(-\log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right) + (1 - p_i) \left(-\log(\sqrt{2\pi}) - \frac{1}{2}\log(\tau^2 + 1) - \frac{y_i^2}{2(\tau^2 + 1)} \right) \right)$$

Note:

1. Expected value of a sum is the sum of the expected values. We do also have that the only thing that is stochastic in the expectation are the indicator functions.

2. let $p_i = P(C_i = 0|y, \theta^{(t)})$. This will imply that $P(C_i = 1|y, \theta^{(t)}) = 1 - P(C_i = 0|y, \theta^{(t)}) = (1 - p_i)$.

The Q function is taking the expected value of the complete log likelihood given both the observed data and the current estimate of the parameters we want to optimize. By optimizing the Q-function (find the arguments that gives the highest value), we will also find the parameters that will give the expected value of the complete log likelihood as high as possible.

First, we will find the updating algorithm for p

$$p^{(t+1)} = \frac{\sum_{i=1}^n p_i}{\sum_{i=1}^n p_i + \sum_{i=1}^n (1 - p_i)} = \frac{\sum_{i=1}^n p_i}{n}$$

where p_i can be calculated as:

$$
\begin{aligned}
p_i &= P(C_i = 0|y_i, \theta^{(t)}) \\
&=^1 \frac{P(C_i = 0, Y_i = y_i|\theta^{(t)})}{P(Y_i = y_i|\theta^{(t)})} \\
&= \frac{P(Y_i = y_i|C_i = 0, \theta^{(t)})P(C_i = 0|\theta^{(t)})}{P(Y_i = y_i|\theta^{(t)})} \\
&=^2 \frac{\phi(y_i; 0, 1^2) \cdot p^{(t)}}{\phi(y_i; 0, 1^2) \cdot p^{(t)} + \phi(y_i; 0, \tau^{2(t)} + 1) \cdot (1 - p^{(t)})}
\end{aligned}
$$

which will make the fully algorithm for updating p as:

$$p^{(t+1)} = \frac{1}{n} \sum_{i=1}^n \frac{\phi(y_i; 0, 1^2) \cdot p^{(t)}}{\phi(y_i; 0, 1^2) \cdot p^{(t)} + \phi(y_i; 0, \tau^{2(t)} + 1) \cdot (1 - p^{(t)})}$$

Note:

1. Bayes theorem

2. Law of total probability

Deriving the estimate for τ^2 by first taking the derivative of the function Q, with respect to τ^2 - and then find the parameter that set the expression to 0.

$$
\begin{aligned}
\frac{\partial}{\partial(\tau^2)} Q(\theta, \theta^{(t)}) &= \sum_{i=1}^n (1 - p_i) \left(-\frac{1}{2(\tau^2 + 1)} + \frac{y_i^2}{2(\tau^2 + 1)^2} \right) \\
&= \sum_{i=1}^n (1 - p_i) \left(\frac{y_i^2}{2(\tau^2 + 1)^2} - \frac{1}{2(\tau^2 + 1)} \right) \stackrel{!}{=} 0
\end{aligned}
$$

$$\frac{\partial}{\partial(\tau^2)}Q(\theta, \theta^{(t)}) = 0 \implies \sum_{i=1}^{n}(1 - p_i)\left(\frac{y_i^2}{2\,(\tau^2 + 1)^2} - \frac{1}{2(\tau^2 + 1)}\right) = 0$$

$$\implies^1 \sum_{i=1}^{n}(1 - p_i)\left(y_i^2 - (\tau^2 + 1)\right) = 0$$

$$\implies \sum_{i=1}^{n}(1 - p_i)y_i^2 = \sum_{i=1}^{n}(1 - p_i)(\tau^2 + 1)$$

$$\implies \sum_{i=1}^{n}(1 - p_i)y_i^2 = (\tau^2 + 1)\sum_{i=1}^{n}(1 - p_i)$$

$$\implies \frac{\sum_{i=1}^{n}(1 - p_i)y_i^2}{\sum_{i=1}^{n}(1 - p_i)} = (\tau^2 + 1)$$

$$\implies \tau^2 = \frac{\sum_{i=1}^{n}(1 - p_i)y_i^2}{\sum_{i=1}^{n}(1 - p_i)} - 1$$

The updating algorithm for τ^2 will then be:

$$\tau^{2(t+1)} = \frac{\sum_{i=1}^{n}(1 - p_i)y_i^2}{\sum_{i=1}^{n}(1 - p_i)} - 1 \tag{8}$$

Note:

1. multiplied both sides with $(\tau^2 + 1)/2$

2.d Implementation of functions

Implement the solution you derived in (c) as a function, and apply it to the data. What is a good initialization

You can see the implementation in the appendix 6.6. A good initialization is to have some $0 < p < 1$, and $\tau^2 \neq 1$ so we do not have two models with same distribution and parameters.

After running the script, we can see that the optimal values converged towards $(p, \tau^2) = (0.964656, 112.156166)$.

2.e Bootstrap estimate (uncertainty)

e) Compute a bootstrap estimate of the uncertainty of the two parameters, using the functions from (d). Sample $B = 1000$ times and display scatter plot of the values

The bootstrap method implementation can be looked up in the appendix 6.7. The bootstrap estimate of the uncertainty are (standard error):

$$SD[p\_simulated] = 0.00623$$
$$SD[\tau^2\_simulated] = 15.6746$$
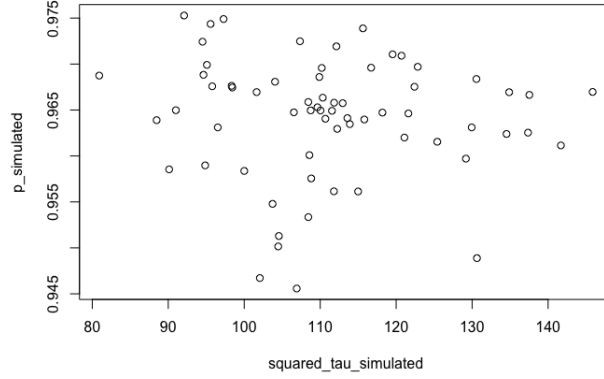
The scatter plot of the values:

Figure 6: scatter plot of the values achieved from the bootstrapping

2.f   Observed information matrix

Compute the observed information matrix. How can you use the observed information matrix to give an uncertainty estimate for θ? Compare the result to (e)

The observed information matrix can be found by:

$$
J_Y(\theta) = - \begin{pmatrix} \frac{\partial^2 Q(\theta)}{\partial p^2} & \frac{\partial^2 Q(\theta)}{\partial p \partial (\tau^2)} \\[2mm] \frac{\partial^2 Q(\theta)}{\partial (\tau^2) \partial p} & \frac{\partial^2 Q(\theta)}{\partial (\tau^2)^2} \end{pmatrix}
$$

Now, we need to have the expression in the EM-algorithm which will include p:

$$
l(\theta) = \sum_{i=1}^n \mathbb{1}_{C_i=0}[\log(p) + \log(\phi(y_i; 0, 1))] + \mathbb{1}_{C_i=1}[\log(1-p) + \log(\phi(y_i; 0, \tau^2+1))]
$$

Then we will get a kind of similar Q as before:

$$
Q(\theta|\theta^{(t)}) = \sum_{i=1}^n \left(p_i \left[\log(p) - \log(\sqrt{2\pi}) - \frac{y_i^2}{2} \right] + (1-p_i) \left[\log(1-p) - \log(\sqrt{2\pi}) - \frac{1}{2}\log(\tau^2+1) - \frac{y_i^2}{2(\tau^2+1)} \right] \right)
$$

Note: We have that $p_i = P(C_i = 0|y_i, \theta^{(t)})$ as before.

Now, we need to do some partial derivatives to get the information matrix.

$$
\frac{\partial}{\partial(\tau^2)} Q(\theta, \theta^{(t)}) = \sum_{i=1}^n (1-p_i) \left(-\frac{1}{2(\tau^2+1)} + \frac{y_i^2}{2(\tau^2+1)^2} \right)
$$

$$
= \sum_{i=1}^n (1-p_i) \left(\frac{y_i^2}{2(\tau^2+1)^2} - \frac{1}{2(\tau^2+1)} \right)
$$

$$
\frac{\partial^2}{\partial(\tau^2)^2} Q(\theta, \theta^{(t)}) = \sum_{i=1}^n (1-p_i) \left(\frac{1}{2(\tau^2+1)^2} - \frac{y_i^2}{(\tau^2+1)^3} \right)
$$

11

$$\frac{\partial^2}{\partial(\tau^2)\partial p}Q(\theta, \theta^{(t)}) = \frac{\partial}{\partial p}\sum_{i=1}^{n}(1-p_i)\left(\frac{1}{2(\tau^2+1)^2} - \frac{y_i^2}{(\tau^2+1)^3}\right)$$

$$= 0$$

$$\frac{\partial}{\partial p}Q(\theta, \theta^{(t)}) = \sum_{i=1}^{n}\left[\frac{p_i}{p} - \frac{1-p_i}{1-p}\right]$$

$$\frac{\partial^2}{\partial p^2}Q(\theta, \theta^{(t)}) = \sum_{i=1}^{n}\left[-\frac{p_i}{p^2} - \frac{1-p_i}{(1-p)^2}\right]$$

So, the final information matrix (before inserting the θ values) will be this diagonal matrix:

$$J_Y(\theta) = -\begin{bmatrix} \sum_{i=1}^{n}\left[-\frac{p_i}{p^2} - \frac{1-p_i}{(1-p)^2}\right] & 0 \\ 0 & \sum_{i=1}^{n}(1-p_i)\left(\frac{1}{2(\tau^2+1)^2} - \frac{y_i^2}{(\tau^2+1)^3}\right) \end{bmatrix}$$

You could use the inverse of the observed information matrix to give an uncertainty estimate for θ. The code for computing the observed information matrix and the estimated covariance matrix (the inverse) will be located at the appendix 6.8.

The estimated covariance matrix looks like this:

$$\text{Cov}(\hat{\theta}) = \begin{bmatrix} 724.5557 & 0 \\ 0 & 3.4 \cdot 10^{-5} \end{bmatrix}$$

So, the standard error for τ^2, p are $26.91757, 0.005839$ (respectively). We can see kind of similar result for the p value, but some. change for the τ^2 value.

2.g   Grid computation of likelihood

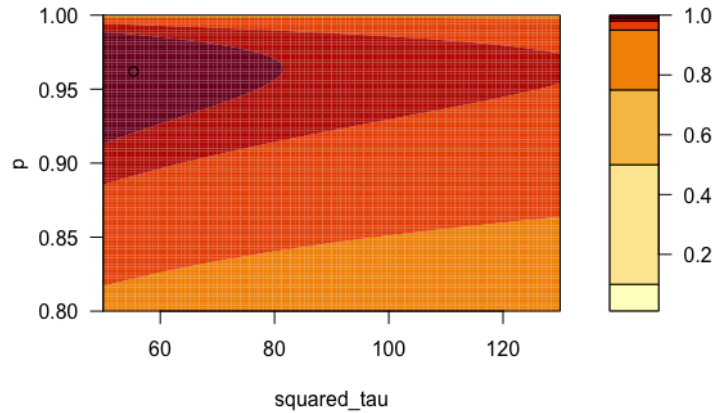The following shows ac contour plot of the likelihood from problem **2a**.

Figure 7: Contour plot of the likelihood

The contour plot shows us where the maximum likelihood estimator (in this region/ grid of values) will approximatly be. It's also easy to interpret that the variation in τ^2 is quite large, while the variation in the p value (probability of corresponding to class 0) is rather small - which was the same insight we got in the two previous subtasks.

3   Exercise 3

Assuming that the data Y_i are the same as in (1) and that y_i, follows mixture distribution as in (7).

3.a   Prove some expressions

Use a Bayesian interpretation of β_i and argue that

$$P(\beta_i = 0|C_i = 0) = 1 \quad \text{and} \quad f(\beta_i|C_i = 1) = \phi(\beta_i; 0, \tau^2)$$

Answer: We have that

$$
\begin{aligned}
Y_i &= \beta_i + \mathcal{E}_i \\
&\implies \quad Y_i|(C_i = 0) = \beta_i|(C_i = 0) + \mathcal{E}_i|(C_i = 0) \\
&\implies^1 \quad Y_i|(C_i = 0) = \beta_i|(C_i = 0) + \mathcal{E}_i
\end{aligned}
$$

Note:

1. $\mathcal{E}_i$ is independent of the class

where we have that the left side of the equation are normal distributed with mean 0 and standard deviation 1. The $\mathcal{E}_i$'s have the same probability density. Then, by assuming that β_i and $\mathcal{E}_i$ are independent, it follows that $\beta_i \sim \mathcal{N}(0,0)$ - have a density of the dirac delta function. This will imply that $P(\beta_i = 0|C_i = 0) = 1$, same as saying that $\beta_i = 0$ (constant), if we are looking at class 0.

Equivalently for the second case:

$$
\begin{aligned}
Y_i &= \beta_i + \mathcal{E}_i \\
&\implies \quad Y_i|(C_i = 1) = \beta_i|(C_i = 1) + \mathcal{E}_i|(C_i = 1) \\
&\implies^1 \quad Y_i|(C_i = 1) = \beta_i|(C_i = 1) + \mathcal{E}_i
\end{aligned}
$$

here the left hand side will be normal distributed with mean 0 and variance $\tau^2 + 1$. $\mathcal{E}_i$ will have the same distribution as always (that is the normal distribution with mean 0 and standard deviation 1). We want to find the distribution of $\beta_i|(C_i = 1)$. We need to have the addition rule of normal distributed variables in mind (as in the previous case). So again, by assuming that $\mathcal{E}_i$ and $\beta_i|(C_i = 1)$ are independent, the only distribution that satisfy this equation will be $\beta_i|(C_i = 1) \sim \mathcal{N}(0, \tau^2)$.

Also, give an expression for $P(\beta_i = 0|y_i = y)$

$$P(\beta_i = 0 | y_i = y) =^1 P(\beta_i = 0 | C_i = 0, y_i = y)P(C_i = 0 | y_i = y)$$
$$+ P(\beta_i = 0 | C_i = 1, y_i = y)P(C_i = 1 | y_i = y)$$
$$=^2 \frac{P(\beta_i = 0, y_i = y | C_i = 0)}{P(y_i = y | C_i = 0)} \frac{P(y_i = y | C_i = 0)P(C_i = 0)}{P(y_i = y)}$$
$$+ \frac{P(\beta_i = 0, y_i = y | C_i = 1)}{P(y_i = y | C_i = 1)} \frac{P(y_i = y | C_i = 1)P(C_i = 1)}{P(y_i = y)}$$
$$= P(\beta_i = 0, y_i = y | C_i = 0)\frac{P(C_i = 0)}{P(y_i = y)}$$
$$+ P(\beta_i = 0, y_i = y | C_i = 1)\frac{P(C_i = 1)}{P(y_i = y)}$$
$$=^3 P(\beta_i = 0, \varepsilon_i = y | C_i = 0)\frac{p}{P(y_i = y)}$$
$$+ P(\beta_i = 0, \varepsilon_i = y | C_i = 1)\frac{(1-p)}{P(y_i = y)}$$
$$=^4 P(\beta_i = 0 | C_i = 0)P(\varepsilon_i = y | C_i = 0)\frac{p}{P(y_i = y)}$$
$$+ P(\beta_i = 0 | C_i = 1)P(\varepsilon_i = y | C_i = 1)\frac{(1-p)}{P(y_i = y)}$$
$$=^5 \phi(y; 0, 1)\frac{p}{P(y_i = y)} + \phi(y; 0, \tau^2)\phi(y; 0, 1)\frac{(1-p)}{P(y_i = y)}$$
$$=^6 \frac{\phi(y; 0, 1)}{P(y_i = y | C_i = 0)P(C_i = 0) + P(y_i = y | C_i = 1)P(C_i = 1)}(p + (1-p)\phi(y; 0, \tau^2))$$
$$= \frac{\phi(y; 0, 1)}{\phi(y; 0, 1)p + \phi(y; 0, \tau^2 + 1)(1-p)}(p + (1-p)\phi(y; 0, \tau^2))$$

Note:

1. Rule of total probability

2. Bayes' rule on all expressions

3. assuming that the data are on the form $y_i = \beta_i + \varepsilon_i$

4. independence of β_i and ε_i

5. adding the known probabilities

6. Rule of total probability

3.b Conditional expectation

Show that:

$$E[\beta_i | y_i = y] \stackrel{!}{=} P(C_i = 1 | y_i = y)E[\beta_i | y_i = y, C_i = 0]$$

Answer:

$$E[\beta_i|y_i = y] = \int_{-\infty}^{\infty} \beta_i[f(\beta_i|y_i = y)]\,d\beta_i$$

$$=^1 \int_{-\infty}^{\infty} \beta_i[f(\beta_i|y_i = y, C_i = 0)P(C_i = 0|y_i = y) + f(\beta_i|y_i = y, C_i = 1)P(C_i = 1|y_i = y)]\,d\beta_i$$

$$= \int_{-\infty}^{\infty} \beta_i[f(\beta_i|y_i = y, C_i = 0)P(C_i = 0|y_i = y)]\,d\beta_i + \int_{-\infty}^{\infty} \beta_i[f(\beta_i|y_i = y, C_i = 1)P(C_i = 1|y_i = y)]\,d\beta_i$$

$$= \left[P(C_i = 0|y_i = y) \int_{-\infty}^{\infty} \beta_i \mathbb{1}_{\beta_i=0}\,d\beta_i \right] + P(C_i = 1|y_i = y)E[\beta_i|y_i = y, C_i = 0]$$

$$=^2 0 + P(C_i = 1|y_i = y)E[\beta_i|y_i = y, C_i = 1]$$

$$= P(C_i = 1|y_i = y)E[\beta_i|y_i = y, C_i = 1]$$

Note:

1. Total probability

2. Integration over 1 point (of value 1), is 0.

Plotting the conditional expectation expression (code: 6.11)
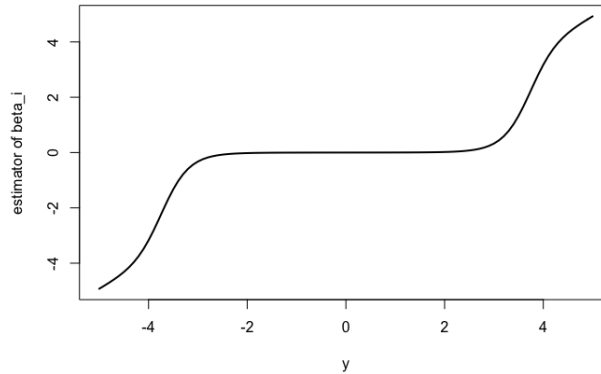
Figure 8: The estimated β for some y values

We can see that the estimator of the *beta*-value are kind of similar to one of the lines in the figure 18 in **exercise 1d** (1.d). It is kind of similar to a p value of $p = 1.1$, since the best beta according to that estimation will be 0 for several values of y around zero. By increasing the γ value from its current value (1), we can see that the optimal beta value will be more similar to the case we are in now with the figure 8.

3.c   Find estimator using a data set

The implementation is done in the appendix section 6.12. We are getting the following output of the code:

residuals
[1] 953.4367

The result isn't too bad, but not the best estimator for β if we compare the result with 1.e.

4 Exercise 4 (Combinatorial optimization)

We will try to minimize the total time spent on delivering a product to 20 different cities.

4.a Simulated annealing

The implementation of the simulated annealing algorithm can be seen in the appendix section 6.14. The results from the code were these figures:

Figure 9: the distance of the current solution over time
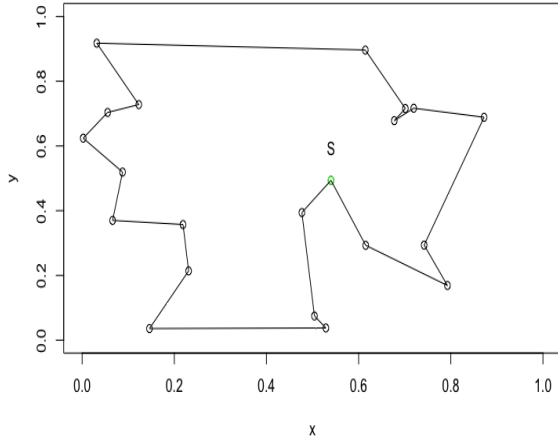
Figure 10: The best route in current simulation (distance: 4.045892)

Figure 11: The all time best route (distance: 3.866708)

I used this neighborhood in the search for a optimum:

$$\mathcal{N}(\theta) = \{\theta^* \mid \exists\, k, j \text{ s.t. } \theta_k^* = \theta_j,\ \theta_j^* = \theta_k \quad \text{and} \quad \theta_i^* = \theta_i\ \forall\, i \neq k, j\}$$

which is an mathematical expression for switching two of the entries in the sequence. The algorithm can reach each state with this neighborhood property. Given a state we want to reach, we can start by switching the first entry to the first entry in the state we want to reach (by switching

the elements in our current sequence), then we will go one with this procedure until we reach the end of the sequence.

The cooling schedule I will be using is:

$$\alpha(t) = 100/(t+1)$$

where t in this context will be the iteration of the algorithm.

4.b   TABU search

Implementation of the algorithm will be found in the appendix (6.15).

TABU search and simulated annealing are able to get to worse states, while the combinatorial optimization algorithm steepest descent will be monotonic since it will always improve (or stand at its current guess). Here are the results from the script, that shows us that this algorithm will go (very often) to worse solutions for finding a better optimum (after more iterations).

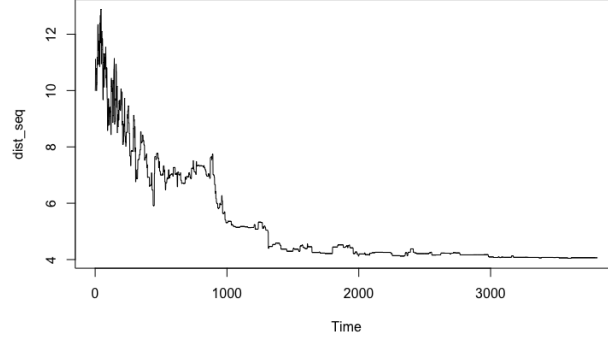The results from the script are as follows:

Figure 12: the distance of the current solution over time

Figure 13: The best route in current simulation (distance: 4.164201)

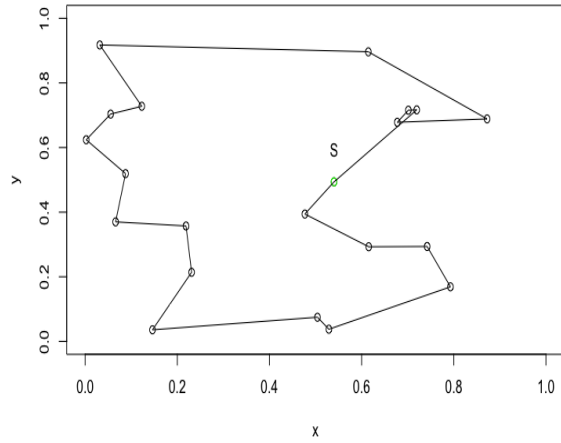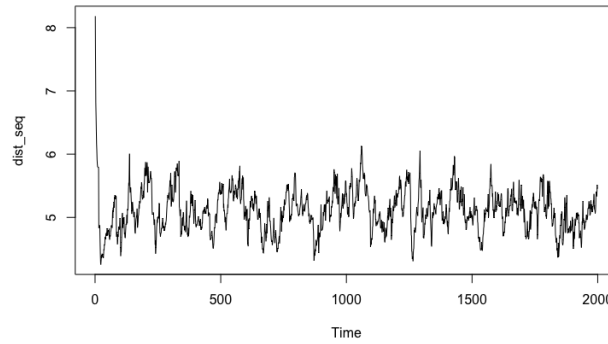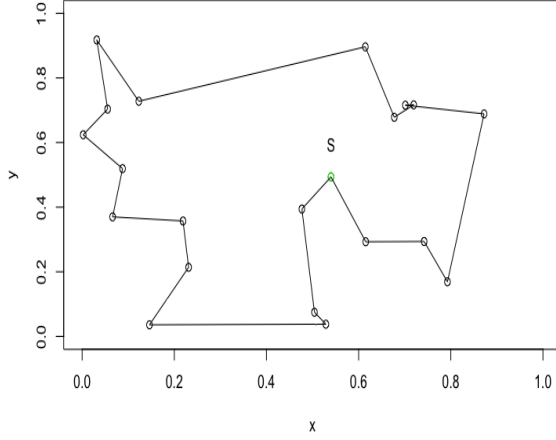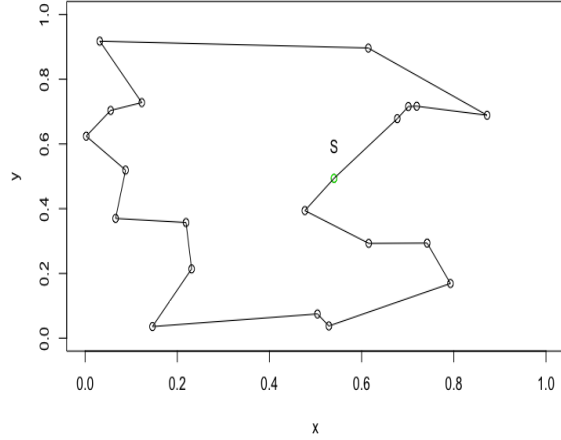**Figure 14:** The all time best route (distance: 3.770697)

4.c   Interpretation to the boss

Manager: "Your current solution, will it be the optimal one?"

Øystein: "Nah, don't think so. It's like $20! = 2.43 * 10^{18}$ solutions to this problem, so I will not guarantee that there will not be a better solution to the optimal route"

Manager: "I knew it, I shouldn't hire you!

Øystein: "Ey boss, I can assure you that the solution I have proposed will be near the optimal solution tho. I have run the problem several times with different initial guesses, and different combinatorial optimization algorithm. My confidence on this result are based on how frequently it has been converged to, and if it converged to something else - it has always been a worse result. It's easy to see that the route is efficient with respect to the map, by reasoning."

Manager: "Oh, sorry for doubting you. You will get a raise soon."

Øystein: "I'm just doing my job sir"

4.d   Another lorry to help with the product transporting

There are many solutions to this problem, but I have decided to go with the following algorithm. In each iteration, there are three possibilities (with equal probability to happen):

1. Lorry 1 switches two destinations

2. Lorry 2 switches two destinations

3. Switching between the lorries (one destination)

Let $(\theta^{(1)}, \theta^{(2)})$ be the routes for the lorries. With the neighbourhoods below (9, 10 and 11) we can't communicate with all states. Instead of changing the neighbourhoods for the states to communicate, we will rather introduce some 0's in the routes $(\theta^{(1)}, \theta^{(2)})$. These 0's can either be placed randomly or strategically (in between every number, for instance). We want to add 0's such that

the length of the route is equal to the actual number of cities (20 in this case).

The initial guess can for instance be random amount of destination (in random order), where every city are included in exactly one of the routes. Then add the 0's as described above.

Note: the 0's will be skipped when first driving a route (invisible for the drivers), but will be helpful for us making the routes. These 0's may be confusing at first sight, but it will be very helpful.

Let's introduce three neighbourhoods which all will be used in the final neighbourhood expression for the problem:

$$
\mathcal{N}_1(\theta^{(1)}, \theta^{(2)}) = \{\theta^{(1)*}, \theta^{(2)*} \mid \exists\, k, j \text{ s.t. } \theta^{(1)*}_k = \theta^{(1)}_j,\ \theta^{(1)*}_j = \theta^{(1)}_k \quad \text{and} \quad \theta^{(1)*}_i = \theta^{(1)}_i \,\forall\, i \neq k, j
$$
$$
\theta^{(2)*}_i = \theta^{(2)}_i \,\forall\, i\} \tag{9}
$$

$$
\mathcal{N}_2(\theta^{(1)}, \theta^{(2)}) = \{\theta^{(1)*}, \theta^{(2)*} \mid \exists\, k, j \text{ s.t. } \theta^{(2)*}_k = \theta^{(2)}_j,\ \theta^{(2)*}_j = \theta^{(2)}_k \quad \text{and} \quad \theta^{(2)*}_i = \theta^{(2)}_i \,\forall\, i \neq k, j
$$
$$
\theta^{(1)*}_i = \theta^{(1)}_i \,\forall\, i\} \tag{10}
$$

$$
\mathcal{N}_3(\theta^{(1)}, \theta^{(2)}) = \{\theta^{(1)*}, \theta^{(2)*} \mid \exists\, k, j \text{ s.t. } \theta^{(1)*}_k = \theta^{(2)}_j,\ \theta^{(2)*}_j = \theta^{(1)}_k \quad \text{and} \quad \theta^{(2)*}_i = \theta^{(2)}_i \,\forall\, i \neq j
$$
$$
\theta^{(1)*}_i = \theta^{(1)}_i \,\forall\, i \neq k\} \tag{11}
$$

our neighbourhood will the union of 9, 10 and 11:

$$
\mathcal{N}(\theta^{(1)}, \theta^{(2)}) = \mathcal{N}_1(\theta^{(1)}, \theta^{(2)}) \cup \mathcal{N}_2(\theta^{(1)}, \theta^{(2)}) \cup \mathcal{N}_3(\theta^{(1)}, \theta^{(2)}) \tag{12}
$$

Communication: The algorithm will then be able to reach to all possible states. This will be the case, since you can follow the following steps:

Given routes/ states: $R = (R_1, R_2)$, where R_i is the route for the i-th lorry.

1. First, use $\mathcal{N}_3$ to switch in all cities in $\theta^{(1)}$ that are included in R_1 (with either 0's or cities included in R_2).

2. Second, use $\mathcal{N}_3$ to switch out all cities in $\theta^{(1)}$ that are included in R_2 (switch them with 0's)

3. Then use both $\mathcal{N}_1$ and $\mathcal{N}_2$ to move all the 0's at the end of the sequence.

4. Last step, rearrange $(\theta^{(1)}, \theta^{(2)})$ s.t. they are equal to R. This can be done, as explained in exercise 4a 4.

We want to minimize the maximum distance that the lorries are driving. In other words, minimize the time for both of them to come back to the starting point given that all the cities have got their products. The implementation of the algorithm is located in the appendix 6.16. The script will get us the following results:
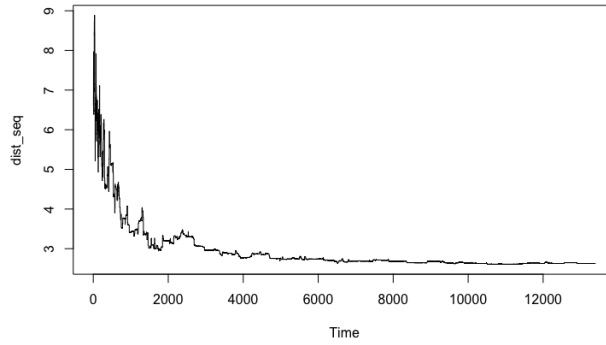
Figure 15: the distribution time of the current solution over iteration in the simulated annealing algorithm
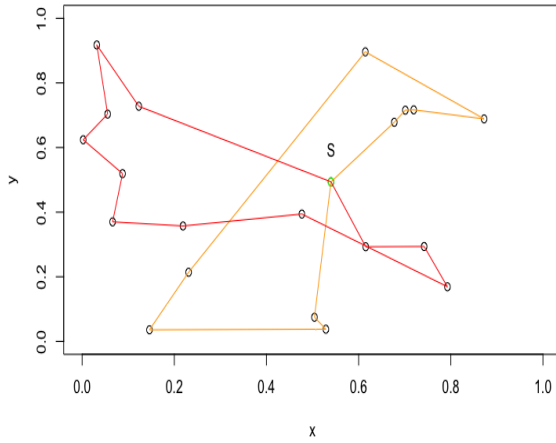
Figure 16: The best route in current simulation (distance: 2.606809)
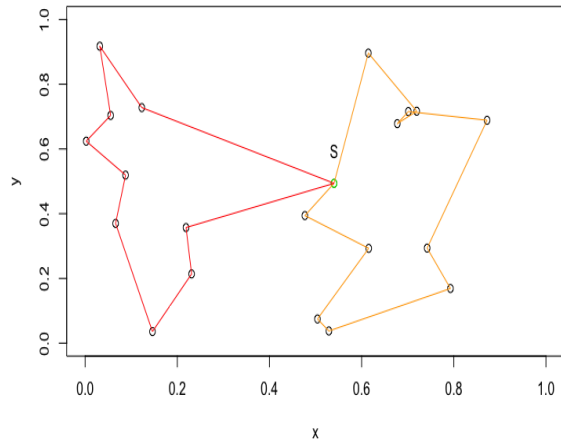
Figure 17: The all time best route (distance: 2.319307)

The company should consider buying a new lorry, since the time usage of delivering the products will reduce a lot (but they need to find if it is necessary to deliver the products faster, or it would be a total waste).

5 Exercise 5 (Stochastic gradient decent, SGD)

We are going through an example of a simple neural network, with one hidden layer of length 50, and will be using the RELU activation function, which is:

$$\sigma(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{13}$$

where the derivative is defined as:

$$\sigma'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{14}$$

,
We have further the architecture of the neural network:

$$f(x) = \sum_{j=1}^{50} \beta_j \sigma(\alpha_j x + \alpha_{0,j}) + \beta_0$$

where α_j and $\alpha_{0,j}$ are the j-th input weight and the j-th bias for the hidden layer. β_j is the output weight for node j and β_0 are the output bias.

We are going to optimize (minimize) according to the sum of squared errors:

$$\text{sse} = \sum_{i=1}^{N} (y_i - \hat{f}(x_i))^2 \tag{15}$$

where y_i are the target (ground truth) value, and $\hat{f}(x_i)$ are the prediction of datapoint i.

5.a   Adapt the gradients of the cost function (15)

The gradients of interest:

$$\frac{\partial \, \text{sse}(\theta)}{\partial \beta_0} = -2 \sum_{i=1}^{N} (y_i - f(x_i))$$

Then for $j = 1, 2, ..., 50$

$$\frac{\partial \, \text{sse}(\theta)}{\partial \beta_j} = -2 \sum_{i=1}^{N} (y_i - f(x_i)) \, \sigma(\alpha_j x_i + \alpha_{0,j})$$

$$\frac{\partial \, \text{sse}(\theta)}{\partial \alpha_j} = -2 \sum_{i=1}^{N} (y_i - f(x_i)) \, \beta_j \sigma'(\alpha_j x_i + \alpha_{0,j}) \, x_i$$

$$\frac{\partial \, \text{sse}(\theta)}{\partial \alpha_{0,j}} = -2 \sum_{i=1}^{N} (y_i - f(x_i)) \, \beta_j \sigma'(\alpha_j x_i + \alpha_{0,j})$$

So, the number of trainable parameters we will have in our current architecture is 151.

5.b   Implement the SGD algorithm

The implementation can be seen in the appendix section at 6.18. I have tested some learning rates, and found a parameter that didn't diverge, but also "converged" kind of fast. The implementation of the SGD algorithm includes a learning-rate scheduler for approach closer to a (local) minima - where I multiply the previous learning rate with a constant (0.8) for each epoch.

Here is the error vs. epoch by running the SGD-algorithm.
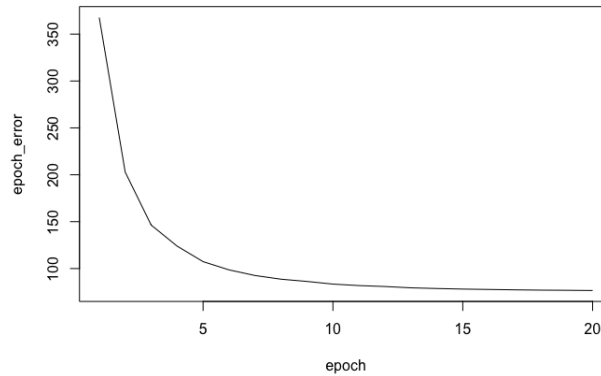
Figure 18: SSE for the data after each epoch

5.c Discuss some choices

I initialized the all the trainable parameters (weights and biases) w.r.t. random standard normal distributed variables. I did this because it is (sometime) used as an intializer for parameters in neural networks and it is easy. Those parameters are random, and will often make the predictions approach different (local) minima's for every time the network is trained.

I chose the learning rate to start with a high enough value for the algorithm to find the right track fast, and apply a scheduler for each epoch (multiplying the learning rate with a number between $(0, 1)$).

I chose to follow the batch choosing as the SGD note used. So, there will be no guarantee that all the data are used in the SGD-algorithm.

6 Appendix

If other exercises uses functions from other exercises, the source - function will be used in the beginning of the script (to make the scripts available for the current task).

6.1 Exercise 1

Here you can see some functions used in the **exercise 1**. The bisection method is a method for finding a root of an expression.

```r
 1   # The function in the task
 2 - f = function(beta, gamma, p) {
 3     beta + gamma * sign(beta) * abs(beta) ^ (p-1)
 4 - }
 5
 6   # Method for finding root of an expression
 7 - bisection_method= function(y, gamma, p, beta1, beta2, eps = 10^-10, max_iter = 1000) {
 8
 9     # We want to find beta, s.t. f(beta) = y
10     # f_bi - the function we want to use bisection method on
11 -   f_bi = function(beta, y.=y, gamma.=gamma, p.=p){
12       return(f(beta, gamma, p) - y)
13 -   }
14
15     # If one of the beta-values give 0, then we have the solution
16 -   if (f_bi(beta1) * f_bi(beta2) == 0) {
17 -     if (f_bi(beta1) == 0){
18         beta1
19 -     } else {
20         beta2
21 -     }
22 -   }
23
24
25     # If they have same sign, we are not guaranteed an
26 -   if (f_bi(beta1) * f_bi(beta2) > 0){
27       stop('The initial guesses have the same sign of the function (not acceptable)')
28 -   }
29
30     converged = FALSE
31     iter = 0
32 -   while (!converged && iter < max_iter) {
33       beta_mid = (beta1 + beta2) /2
34
35       # Checking if beta1, or beta2 should be updated, if opposite sign => keep them
36 -     if (f_bi(beta_mid) * f_bi(beta1) < 0){
37         beta2 = beta_mid
38 -     } else if (f_bi(beta_mid) * f_bi(beta2) < 0) {
39         beta1 = beta_mid
40 -     } else {
41         return(beta_mid)
42 -     }
43
44
45       # updating criterion of convergence
46       iter = iter + 1
47       converged = abs(f_bi(beta_mid)) < eps
48
49 -   }
50
51     return(beta_mid)
52 - }
```

6.2 Exercise 1c

This script will for two $'s$, plotting the function f for different p-values. We will also plot the inverse function by flipping the order of the arguments.

```
 1  source('exercise_1_functions.R')
 2
 3  beta = seq(-5, 5, length = 101)
 4  gamma = 1
 5  ps = c(1.1, 2, 5, 100)
 6
 7  # Plotting all the results for gamma = 1
 8  first = TRUE
 9  count = 1
10  for(p in ps){
11    if(first){
12      plot(beta, f(beta, gamma, p), type='l', col = count, ylim = c(-8, 8), xlim = c(-5, 5),
13          lwd = 2, main = 'The function with gamma = 1', ylab = expression("f"['p, y'] ~ (beta)))
14      first = FALSE
15    } else {
16      lines(beta, f(beta, gamma, p), col = count, lwd = 2)
17    }
18
19    count = count + 1
20  }
21
22  legend(-5, 6,legend = c('p = 1.1', 'p = 2', 'p = 5', 'p = 100'), fill = 1:4)
23  points(beta, beta, col = 'yellow')
24
25  gamma = 0.2
26
27  # Plotting all the results for gamma = 0.2
28  first = TRUE
29  count = 1
30  for(p in ps){
31    if(first){
32      plot(beta, f(beta, gamma, p), type='l', col = count, ylim = c(-8, 8), xlim = c(-5.1, 5.1),
33          lwd = 2, main = 'The function with gamma = 0.2', ylab = expression("f"['p, y'] ~ (beta)))
34      first = FALSE
35    } else {
36      lines(beta, f(beta, gamma, p), col = count, lwd = 2)
37    }
38
39    count = count + 1
40  }
41
42
43  legend(-5, 6,legend = c('p = 1.1', 'p = 2', 'p = 5', 'p = 100'), fill = 1:4)
44  points(beta, beta, col = 'yellow')
45
46
```

6.3   Exercise 1d

This script calls the functions in 6.1 to be able to use bisection method to find the solution of $beta_opt$ given y.

```r
source('exercise_1_functions.R')

# check the f_bi - function
# plot(beta, f_bi(beta, y, gamma, p), type = 'l')

# test bisection method
gamma = 1
ps = c(1.1, 2, 100)
y = seq(-5, 5, length = 101)

beta_init1 = -5
beta_init2 = 4.9

first = TRUE
count = 1
for(p in ps){
  val = c()
  for (y_i in y){
    val = c(val, bisection_method(y_i, gamma, p, beta_init1, beta_init2))
  }
  print(val)

  if(first){
    plot(y, val, type='l', col = count, ylim = c(-5, 5), xlim = c(-5.1, 5.1),
         lwd = 2, main = 'beta_opt w.r.t. y', xlab = 'y', ylab = 'beta_opt')
    first = FALSE
  } else {
    lines(y, val, col = count, lwd = 2)
  }

  count = count + 1
}

legend(-5, 5 ,legend = c('p = 1.1', 'p = 2', 'p = 100'), fill = 1:3)
```

6.4 Exercise 1e

This script calls the functions in 6.1 to be able to use bisection method to find the solution of $beta_opt$ given y. We are using a data set, and comparing the different methods (derived earlier in the task). And some alternative way of achieving the $\beta's$ that were introduced in this task.

```r
1   # Reading in the data
2   source('exercise_1_functions.R')
3   sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
4   colnames(sparseData) = c('betaGT', 'y')
5
6   # Collecting the data from the dataframe
7   y = sparseData$y
8   betaGT = sparseData$betaGT
9
10  # We want to estimate beta_optimal for penalized regression
11  gamma = 1
12  ps = c(1.1, 2, 100)
13  init_beta1 = -100
14  init_beta2 = 101
15  residuals = c()
16  beta_penelized = list()
17
18  # Finding the residuals for different p-values
19  for (p in ps){
20      estimation = c()
21      for (y_i in y){
22          estimation = c(estimation, bisection_method(y_i, gamma, p, init_beta1, init_beta2))
23      }
24
25      # comparing the estimation vs  the ground truth
26      beta_penelized = c(beta_penelized, list(estimation))
27      residuals = c(residuals, sum((estimation-betaGT)^2))
28  }
29
30  residuals = data.frame(ps, residuals)
31
32
33  # Comparison to the MLE estimator, just equal to y
34  beta_MLE = y
35  residual_MLE = sum((beta_MLE - betaGT)^2)
36
37  # Estimate beta by using the residuals
38  y_list3 = rep(list(y), 3)
39
40  beta_alternative = list()
41  residuals_alternative = rep(NA, 3)
42  for (i in 1:3) {
43      beta_alternative = c(beta_alternative, list(unlist(y_list3[i]) - unlist(beta_penelized[i])))
44      residuals_alternative[i] = sum((unlist(beta_alternative[i]) - betaGT))^2
45  }
46
47  residuals_alternative_table = data.frame(ps, residual_alternative)
48
49  # First case
50  residuals
51  # Second case
52  residual_MLE
53  # Third case
54  residuals_alternative
55
```

Results (terminal output):

```
> # First case
> residuals
      ps  residuals
1    1.1   273.9674
2    2.0  1256.5316
3  100.0  3786.6181

> # Second case
> residual_MLE
[1]  998.4152

> # Third case
> residuals_alternative
[1] 5785.6463 2425.7279   333.3705
```

where the last numbers are against p-values introduced (respectively).

6.5 Exercise 2

Here you can see some functions used in **exercise 2**. Among those are p_i - which is described in the solution, both updating functions for (p, τ^2), and the EM-algorithm.

```
1  p_i = function(y_i, squared_tau, p) {
2     numerator = dnorm(y_i, mean = 0, sd = 1) * p
3     denominator = numerator + dnorm(y_i, mean = 0, sd = squared_tau + 1) * (1 - p)
4
5     numerator/denominator
6  }
7
8  p_update = function(y, squared_tau_cur, p_cur) {
9     sum = 0
10    for (y_i in y){
11       sum = sum + p_i(y_i, squared_tau_cur, p_cur)
12    }
13
14    sum/length(y)
15  }
16
17  squared_tau_update = function(y, squared_tau_cur, p_cur) {
18     numerator = 0
19     denominator = 0
20     for(y_i in y){
21        numerator = numerator + (1 - p_i(y_i, squared_tau_cur, p_cur))*y_i^2
22        denominator = denominator + (1 - p_i(y_i, squared_tau_cur, p_cur))
23     }
24
25     numerator/denominator - 1
26  }
27
28  EM_algorithm = function(y, p_1, squared_tau_1, printing = FALSE, eps = 10^-10, max_iter = 1000){
29
30     iter = 1
31     converged = FALSE
32     while(!(converged) && iter < max_iter){
33        squared_tau_2 = squared_tau_update(y, squared_tau_1, p_1)
34        p_2 = p_update(y, squared_tau_1, p_1)
35
36        # check if both of the parameters have converged
37        if(abs(p_2 - p_1) + abs(squared_tau_2 - squared_tau_1) < eps) {
38           converged = TRUE
39        }
40
41        if(printing){
42           save = sprintf('iter: %2.0f, %f %f', iter, p_1, squared_tau_1)
43           print(save)
44        }
45
46        # updating the parameters
47        squared_tau_1 = squared_tau_2
48        p_1 = p_2
49
50        iter = iter + 1
51     }
52
53     c(p_2, squared_tau_2)
54  }
55
```

6.6 Exercise 2d

Implementing the solution of the Q function, and use the EM algorithm on the data.

```r
source('exercise_2_functions.R')
sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
colnames(sparseData) = c('betaGT', 'y')

# Collecting the data from the dataframe
y = sparseData$y
betaGT = sparseData$betaGT

# need to take an initialization away from class 0
squared_tau_init = 2
p_init = 0.5

# running the EM-algorithm, and collecting the results
values = EM_algorithm(y, p_init, squared_tau_init, printing = TRUE)
p_opt = values[1]
squared_tau_opt = values[2]
```

Results:

```
iter:   1,  0.500000  2.000000
iter:  17,  0.964656  112.156166
```

where the values correspond to (p, τ^2) respectively.

6.7 Exercise 2e

The code for getting the uncertainty with bootstrapping

```r
1   source('exercise_2_functions.R')
2   sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
3   colnames(sparseData) = c('betaGT', 'y')
4
5   # Collecting the data from the dataframe
6   y = sparseData$y
7   betaGT = sparseData$betaGT
8
9   # need to take an initialization away from class 0
10  squared_tau_init = mean(y)
11  p_init = 0.5
12
13  values = EM_algorithm(y, p_init, squared_tau_init)
14  p_opt = values[1]
15  squared_tau_opt = values[2]
16
17  # running bootstrap method for calculating the uncertainty of the parameters
18  B = 1000
19  n = length(y)
20  p_simulated = rep(NA, B)
21  squared_tau_simulated = rep(NA, B)
22 ▾ for(b in 1:B) {
23    y_sample = sample(y, n, replace = T)
24    values = EM_algorithm(y_sample, p_init, squared_tau_init)
25    p_simulated[b] = values[1]
26    squared_tau_simulated[b] = values[2]
27    print(b)
28
29 ▴ }
30
31  plot(squared_tau_simulated, p_simulated)
32
33
34  print('Bias:')
35  show(mean(p_simulated) - p_opt)
36  show(mean(squared_tau_simulated) - squared_tau_opt)
37
38  print('Standard error:')
39  show(sd(p_simulated))
40  show(sd(squared_tau_simulated))
41
42
```

6.8   Exercise 2f

Computing the information matrix, so i can find the estimated covariance matrix by taking the inverse.

```
1   source('exercise_1_functions.R')
2   source('exercise_2_functions.R')
3
4   sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
5   colnames(sparseData) = c('betaGT', 'y')
6
7   # Collecting the data from the dataframe
8   y = sparseData$y
9   betaGT = sparseData$betaGT
10
11  # the optmial values from the EM-algorithm
12  p = 0.964656
13  squared_tau = 112.156166
14
15 ▾ complete_information_matrix = function(y, squared_tau, p){
16    information = matrix(1:4, nrow = 2, ncol = 2)
17    information[1, 2] = 0
18    information[2, 1] = 0
19
20    one_one = sum((1-p_i(y, squared_tau, p)) *
21                     ((1/(2*(squared_tau + 1 )^2)) -
22                     (y^2/((squared_tau + 1)^3)))
23                   )
24    two_two = sum(-1/(p^2) * p_i(y, squared_tau, p) -
25                     1/((1-p)^2) * (1 - p_i(y, squared_tau, p)))
26
27    information[1, 1] = - one_one
28    information[2, 2] = - two_two
29    return(information)
30 ▴ }
31
32  # The inverse of complete_information -> estimate of covariance matrix
33  # of the estimated parameters
34
35 ▾ estimated_covariance_matrix = function(y, squared_tau, p){
36    return(solve(complete_information_matrix(y, squared_tau, p)))
37 ▴ }
38
39  covariance_matrix = estimated_covariance_matrix(y, squared_tau, p)
40
41
```

6.9   Exercise 2g

The following script will create a contour plot of the likelihood function presented in task 2a (2.a).

```r
1   source('exercise_1_functions.R')
2   sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
3   colnames(sparseData) = c('betaGT', 'y')
4
5   # Collecting the data from the dataframe
6   y = sparseData$y
7   betaGT = sparseData$betaGT
8
9   # the likelihood in 2a
10  loglikelihood = function(y, p, squared_tau) {
11    fy = p*dnorm(y, mean = 0, sd = 1) + (1-p)*dnorm(y, mean = 0, sd = (squared_tau + 1))
12    log_fy = log(fy)
13
14    return(sum(log_fy))
15  }
16
17  p = seq(0.8, 1, length = 101)
18  squared_tau = seq(50, 130, length = 101)
19
20  z = matrix(1:(101*101), length(squared_tau), length(p))
21  for(i in 1:length(squared_tau)) {
22    for(j in 1:length(p)){
23      z[i,j] = loglikelihood(y, p[j], squared_tau[i])
24    }
25  }
26
27  # need a transformation to [0, 1] - where 1 is the best value - max(z)
28  z = max(z)/z
29
30  levels_ = c(0.01, 0.1, 0.5, 0.75, 0.95, 0.98, 0.99, 1)
31  contour(squared_tau, p, z, lwd = 2,
32          ylab = 'p', xlab = 'squared_tau', levels = levels_)
33  filled.contour(squared_tau, p, z, ylab = 'p',
34                 xlab = 'squared_tau', levels = levels_) #nlevels = 50)
35
36  # Mark the ML estimator in the plot
37  idx = which(z == max(z), arr.ind = TRUE)
38  points(squared_tau[idx[1]], p[idx[2]])
```

6.10   Exercise 3

Functions that will be used in the task.

```
1   # The expectation of the expression (using the hint)
2 ▾ E = function(y, squared_tau){
3      return(y * squared_tau/(squared_tau + 1))
4 ▴ }
5
6   # The probability part is (look up def of p_i in overleaf):
7 ▾ prob_C_1 = function(y, squared_tau, p) {
8      return(1 - p_i(y, squared_tau, p))
9 ▴ }
10
11  # estimator
12 ▾ beta_estimator = function(y, squared_tau, p){
13      return(prob_C_1(y, squared_tau, p) * E(y, squared_tau))
14 ▴ }
```

6.11   Exercise 3b

```
1   source('exercise_2_functions.R')
2   source('exercise3_functions.R')
3
4   p = 0.9
5   squared_tau = 80
6
7   y = seq(-5, 5, length = 101)
8
9   # plotting the expression of the task
10  plot(y, beta_estimator(y, squared_tau, p), type = 'l', lwd = 2, ylab = 'estimator of beta_i')
11
12  |
```

6.12   Exercise 3c

```
1   source('exercise_2_functions.R')
2   source('exercise3_functions.R')
3
4   # Reading in the data
5   sparseData = read.table("data/sparseDataWithErrors.ascii", header=F)
6   colnames(sparseData) = c('betaGT', 'y')
7
8   # Collecting the data from the dataframe
9   y = sparseData$y
10  betaGT = sparseData$betaGT
11
12  p = 0.9
13  squared_tau = 80
14
15  # find the sum of squares residuals
16  beta_estimates = beta_estimator(y, squared_tau, p)
17
18  residuals = sum((beta_estimates-y)^2)
19  |
20
```

6.13 Exercise 4

Here you can see some functions used in **exercise 4**. The functions you can see on the first page are two functions for calculating the distance of traveling the routes, and a distance of traveling max - which represent the maximum time there will be for two lorries to deliver products. The last function seen at this page is for visualize the travel, so we can interpret if it is somehow correct (that is do not have some terrible choices).

```r
1
2   # start and end with index 1
3 ▾ distance_of_traveling = function(x, y, route) {
4
5     # previous location is the start (which is index 1 in the list x and y)
6     old_loc = 1
7     dist = 0
8 ▾   for(new_loc in route) {
9       dist = dist + sqrt((x[new_loc] - x[old_loc])^2 + (y[new_loc] - y[old_loc])^2)
10      old_loc = new_loc
11 ▴   }
12
13    # going back to start (index 1 in the lists)
14    dist = dist + sqrt((x[1] - x[old_loc])^2 + (y[1] - y[old_loc])^2)
15 ▴ }
16
17 ▾ distance_of_traveling_max = function(x, y, route1, route2) {
18    # removing the zeros in the routes
19    route1 = route1[route1 > 0]
20    route2 = route2[route2 > 0]
21
22    # finding the maximum distance of them (that is time spent on delivering and get back)
23    dist1 = distance_of_traveling(x, y, route1)
24    dist2 = distance_of_traveling(x, y, route2)
25    max(dist1, dist2)
26 ▴ }
27
28  # Function for visualize the optimized route
29 ▾ draw_route_of_traveling = function(x, y, route, wait = 0.3){
30
31    plot(x, y, xlim = c(0, 1), ylim = c(0, 1))
32    points(x[1], y[1], col = 'green')
33    text(x[1], y[1] + 0.1, labels = 'S')
34
35    old_loc = 1
36 ▾   for(new_loc in route) {
37
38      lines(c(x[new_loc], x[old_loc]), c(y[new_loc],  y[old_loc]))
39      old_loc = new_loc
40      Sys.sleep(wait)
41 ▴   }
42
43    lines(c(x[1], x[old_loc]), c(y[1],  y[old_loc]))
44 ▴ }
45
```

The function seen on this page is used by exercise 4d, where we have two lorries that have their own route they need to drive. It will draw the routes for the two lorries

```
46    # Function for visualize the optimized route with 2 lorries
47  draw_route_of_traveling_2 = function(x, y, route1, route2, wait = 0.3){
48      # keeping just the non-zero values.
49      route1 = route1[route1 > 0]
50      route2 = route2[route2 > 0]
51
52      plot(x, y, xlim = c(0, 1), ylim = c(0, 1))
53      points(x[1], y[1], col = 'green')
54      text(x[1], y[1] + 0.1, labels = 'S')
55
56      old_loc = 1
57      for(new_loc in route1) {
58
59        lines(c(x[new_loc], x[old_loc]), c(y[new_loc],  y[old_loc]), col = 'orange')
60        old_loc = new_loc
61        Sys.sleep(wait)
62      }
63
64      lines(c(x[1], x[old_loc]), c(y[1],  y[old_loc]), col = 'orange')
65
66
67      old_loc = 1
68      dist = 0
69      for(new_loc in route2) {
70
71        lines(c(x[new_loc], x[old_loc]), c(y[new_loc],  y[old_loc]), col = 'red')
72        old_loc = new_loc
73        Sys.sleep(wait)
74      }
75
76      lines(c(x[1], x[old_loc]), c(y[1],  y[old_loc]), col = 'red')
77  }
78
```

The first function on this page shows how to initialize the θ solutions in the last case (two lorries). We introduces the zero's as explained in the solution to this exercise. The last function on this page is a simple function for checking whether the first argument are included in the second argument (where the second argument is a list). This is very convenient to use when looking at the TABU algorithm.

```
79   # Every_other_zero - as long as it is possible
80 ▾ every_other_zero = function(n, array){
81
82     new_array = c(rep(0, max(2 * length(array), n)))
83
84 ▾   for (i in 1:length(array)) {
85       new_array[2*i] = array[i]
86 ▴   }
87
88     array_len = length(new_array)
89 ▾   while(array_len > n) {
90 ▾     if (new_array[array_len] == 0 && new_array[array_len-1] == 0) {
91         new_array = new_array[1:(array_len - 1)]
92 ▾     } else {
93         # removing one index with 0
94         remove_index = sample((1:array_len)[new_array == 0], 1)
95         new_array = new_array[1:array_len != remove_index]
96 ▴     }
97       array_len = array_len - 1
98
99 ▴   }
100
101    # ending array shall have length n
102    new_array
103 ▴ }
104
105
106 ▾ list_is_contained = function(list1, seq_list) {
107    # check if list1 (either list or sequence) are inside the seq_list
108
109 ▾   for (list_test in seq_list){
110 ▾     if(sum(unlist(list1) %in% list_test) == 2){
111         return(TRUE)
112 ▴     }
113 ▴   }
114    return(FALSE)
115 ▴ }
```

6.14 Exercise 4a

Implementing the simulated annealing algorithm

```r
1   source('exercise_4_functions.R')
2   optimalTransport = read.table("data/optimalTransport.ascii", header=F)
3   colnames(optimalTransport) = c('x', 'y')
4
5   # x and y are positions (distance is time)
6   x = optimalTransport$x
7   y = optimalTransport$y
8
9
10  # distance 3.8667
11  best_to_now = c(19, 14, 13, 3, 16, 11, 6, 2, 5, 10, 20, 7, 4, 21, 8, 12, 18, 17, 9, 15)
12
13
14  # fox (dist = 4.1695):
15  # best_to_now = c(19, 21, 8, 4, 7, 20, 10, 5, 2, 6, 11, 16, 3, 13, 14, 18, 15, 9, 12, 17)
16
17 - if (FALSE){
18      route.current = best_to_now
19 - } else {
20      route.current = sample(2:length(x), replace = FALSE)
21 - }
22  n = length(route.current)
23
24  # We measure the model in the distance of the route (shorter => better)
25  dist_current = distance_of_traveling(x, y, route.current)
26
27  # Wish to have control of the distance over time
28  dist_seq = c()
29
30  # want to save the best run so far
31  best_route_now = route.current
32  best_dist_now = dist_current
33
34  # Initialization for convergence criterion
35  max_iter = 100000
36  iter = 1
37  converged = FALSE
38  same_value = 0
39  max_same_value = 200
40  last_dist = 0
41
```

```r
42   # simulated annealing
43   while(!(converged) && iter < max_iter)
44     {
45       # the cooling
46       tau = 100/(iter+1)
47
48       # collecting two indexes random, and switch those
49       index_of_switching = sample(1 : n, 2, replace = FALSE)
50
51       value_1 = route.current[index_of_switching[1]]
52       value_2 = route.current[index_of_switching[2]]
53       route.neighboor = route.current
54       route.neighboor[index_of_switching[1]] = value_2
55       route.neighboor[index_of_switching[2]] = value_1
56
57       dist_neighboor = distance_of_traveling(x, y, route.neighboor)
58
59       # The probability of switching the model
60       prob = exp((dist_current - dist_neighboor)/tau)
61
62       # We will switch to newer model if it is better (sometimes if it is worse)
63       u = runif(1)
64       if(u<prob)
65       {
66         route.current = route.neighboor
67         dist_current = dist_neighboor
68
69         if (dist_current < best_dist_now) {
70           best_route_now = route.current
71           best_dist_now = dist_current
72         }
73       }
74
75       # adding the distance to see the improvement of the model
76       dist_seq = c(dist_seq, dist_current)
77
78       # check if we have had an change in the parameters
79       if (last_dist == dist_current){
80         same_value = same_value + 1
81       } else {
82         same_value = 0
83         last_dist = dist_current
84       }
85
86       # If we have same parameters for "max_same_value" rounds, then stop the iterations
87       if (same_value == max_same_value) {
88         converged = TRUE
89       }
90
91       iter = iter + 1
92   }
```

```
 94   plot.ts(dist_seq)
 95   show(distance_of_traveling(x, y, best_to_now))
 96   show(min(dist_seq))
 97
 98   #draw_route_of_traveling(x, y, route.current, wait = 0.3)
 99   draw_route_of_traveling(x, y, best_to_now, wait = 0.3)
100   draw_route_of_traveling(x, y, best_route_now, wait = 0.3)
```

6.15 Exercise 4b

Implementing the TABU-algorithm

```r
 1  source('exercise_4_functions.R')
 2  optimalTransport = read.table("data/optimalTransport.ascii", header=F)
 3  colnames(optimalTransport) = c('x', 'y')
 4
 5  # x and y are positions (distance is time)
 6  x = optimalTransport$x
 7  y = optimalTransport$y
 8
 9  # distance 3.770697
10  best_to_now = c(19, 14, 13, 3, 16, 11, 6, 2, 5, 10, 20, 7, 4, 21, 8, 12, 18, 15, 9, 17)
11
12  # distance 4.169531 fox?
13  # best_to_now3 = c(19, 21, 8, 4, 7, 20, 10, 5, 2, 6, 11, 16, 3, 13, 14, 18, 15, 9, 12, 17)
14
15  # If true start from the best to now solution
16  if (FALSE){
17      route.current = best_to_now
18  } else {
19      route.current = sample(2:length(x), replace = FALSE)
20  }
21  n = length(route.current)
22
23  # We measure the model in the distance of the route (shorter => better)
24  dist_current = distance_of_traveling(x, y, route.current)
25
26  # Wish to have control of the distance over time
27  dist_seq = c()
28
29  # want to save the best run so far
30  best_route_now = route.current
31  best_dist_now = dist_current
32
33  # Initialization for convergence criterion
34  max_iter = 2000
35  iter = 1
36  converged = FALSE
37  same_value = 0
38  max_same_value = 200
39
40  possible_changes = c()
41  for(i in 1: 19){
42      for(j in (i+1): 20) {
43          possible_changes = c(possible_changes, list(c(i, j)))
44      }
45  }
46
47  max_threshold_tabu = 50
48  tabu_list = c(list(1, 5))
49
```

```
50  # tabu algorithm
51  while(!(converged) && iter < max_iter)
52 ▾ {
53     # first go through all non-listed (tabu) - check for the best switch
54     neighboor_best = 100000
55 ▾   for (switch_indexes in possible_changes) {
56       # making copy of the current solution
57       route.test = route.current
58       value1 = route.current[switch_indexes[1]]
59       value2 = route.current[switch_indexes[2]]
60       route.test[switch_indexes[2]] = value1
61       route.test[switch_indexes[1]] = value2
62       dist.test = distance_of_traveling(x, y, route.test)
63
64       # If it is not tabu, we will check if it is the best of the available neighboors so far
65 ▾     if (!(list_is_contained(switch_indexes, tabu_list))) {
66 ▾       if (dist.test < neighboor_best) {
67           best_switch = switch_indexes
68           neighboor_best = dist.test
69 ▴       }
70 ▴     }
71
72       # want to store the best one yet, although it is tabu
73 ▾     if (dist.test < best_dist_now) {
74         best_route_now = route.test
75         best_dist_now = dist.test
76 ▴     }
77 ▴   }
78
79     # switching to the solution to best neighboor, (although it is worse than this state)
80     value1 = route.current[best_switch[1]]
81     value2 = route.current[best_switch[2]]
82     route.current[best_switch[1]] = value2
83     route.current[best_switch[2]] = value1
84     dist_current = distance_of_traveling(x, y, route.current)
85     dist_seq = c(dist_seq, dist_current)
86
87     # updating the tabu_list, oldest tabu out if exceeding the threshold
88     tabu_list = c(tabu_list, list(c(best_switch[1], best_switch[2])
89                    [order(c(best_switch[1], best_switch[2]))]))
90 ▾   if (length(tabu_list) > max_threshold_tabu) {
91       tabu_list = tabu_list[-1]
92 ▴   }
93 ▴ }
```

```
 95   plot.ts(dist_seq)
 96   show(min(dist_seq))
 97   show(distance_of_traveling(x, y, best_route_now))
 98   show(distance_of_traveling(x, y, best_to_now))
 99
100   #draw_route_of_traveling(x, y, route.current, wait = 0.3)
101   draw_route_of_traveling(x, y, best_route_now, wait = 0.3)
102   draw_route_of_traveling(x, y, best_to_now, wait = 0.3)
```

6.16 Exercise 4d

Implementing the simulated annealing algorithm for two lorries.

```r
1   source('exercise_4_functions.R')
2   optimalTransport = read.table("data/optimalTransport.ascii", header=F)
3   colnames(optimalTransport) = c('x', 'y')
4
5   # x and y are positions (distance is time)
6   x = optimalTransport$x
7   y = optimalTransport$y
8
9   # Adding all the cities, and equal amount of 0's to the places list, that
10  # each theta will divide between
11  places = sample(c(sample(2:length(x), replace = FALSE), rep(0, length(x) - 1)))
12  n = 20
13  random_places = sample(c(0, 1), n, replace = TRUE)
14
15
16  # Score: 2.31907
17  best_route1 = c(19, 14, 11, 16, 3, 13, 18, 9, 17, 15, 12)
18  best_route2 = c(21, 8, 4, 7, 20, 10, 6, 2, 5)
19
20  # We are either using the best one yet, or random (shuffle some 0's inside the array)
21  if(FALSE){
22      route1.current = every_other_zero(n, best_route1)
23      route2.current = every_other_zero(n, best_route2)
24  } else {
25      # splitting 50/50 of the places to the routes
26      route1.current = places[c(1:20)]
27      route2.current = places[c(21:40)]
28  }
29
30
31  dist_current = distance_of_traveling_max(x, y, route1.current, route2.current)
32  #draw_route_of_traveling_2(x, y, route1.current, route2.current)
33
34  # want to save the best run so far
35  best_route1_now = route1.current
36  best_route2_now = route2.current
37  best_dist_now = dist_current
38
39  # Wish to have control of the distance over time
40  dist_seq = c()
41
42  # Initialization for convergence criterion
43  max_iter = 100000
44  iter = 1
45  converged = FALSE
46  same_value = 0
47  max_same_value = 500
48  last_dist = 0
```

```
50  # Simulated annealing
51  while(!(converged) && iter < max_iter)
52 ▾ {
53    # the cooling
54    tau = 100/(iter+1)
55
56    # copying the current route
57    route1.neighbor = route1.current
58    route2.neighbor = route2.current
59
60    # Finding two values to switch, in one case
61    index_of_switching = sample(1 : n, 2, replace = FALSE)
62
63    prob = runif(1)
64    # Half the time, switch with its current cities
65 ▾  if (prob > 2/3) {
66      value_1 = route1.current[index_of_switching[1]]
67      value_2 = route1.current[index_of_switching[2]]
68      # Creating the neighborhood for route1
69      route1.neighbor[index_of_switching[1]] = value_2
70      route1.neighbor[index_of_switching[2]] = value_1
71
72 ▾  } else if (prob > 1/3){
73      value_1 = route2.current[index_of_switching[1]]
74      value_2 = route2.current[index_of_switching[2]]
75      # Creating the neighborhood for route2
76      route2.neighbor[index_of_switching[1]] = value_2
77      route2.neighbor[index_of_switching[2]] = value_1
78
79 ▾  } else {
80      # If not they above, then switching between the cars
81      index_of_switching = sample(1 : n, 2, replace = FALSE)
82
83      value_1 = route1.current[index_of_switching[1]]
84      value_2 = route2.current[index_of_switching[2]]
85      # switching the values
86      route1.neighbor[index_of_switching[1]] = value_2
87      route2.neighbor[index_of_switching[2]] = value_1
88 ▴  }
89
90    dist_neighbor = distance_of_traveling_max(x, y, route1.neighbor, route2.neighbor)
91
```

```r
91
92      # The probability of switching the model
93      prob = exp((dist_current - dist_neighbor)/tau)
94
95      # We will switch to newer model if it is better (sometimes if it is worse)
96      u = runif(1)
97      if(u<prob)
98      {
99        # updating the routes
100       route1.current = route1.neighbor
101       route2.current = route2.neighbor
102       dist_current = dist_neighbor
103
104       if (dist_current < best_dist_now) {
105         best_route1_now = route1.current
106         best_route2_now = route2.current
107         best_dist_now = dist_current
108       }
109
110     }
111
112     # adding the distance to see the improvement of the model
113     dist_seq = c(dist_seq, dist_current)
114
115     # check if we have had an change in the parameters
116     if (last_dist == dist_current){
117       same_value = same_value + 1
118     } else {
119       same_value = 0
120       last_dist = dist_current
121     }
122
123     # If we have same parameters for "max_same_value" rounds, then stop the iterations
124     if (same_value == max_same_value) {
125       converged = TRUE
126     }
127
128     iter = iter + 1
129 }
130
131 plot.ts(dist_seq)
132 show(min(dist_seq))
133 show(distance_of_traveling_max(x, y, best_route1, best_route2))
134
135 draw_route_of_traveling_2(x, y, best_route1, best_route2, wait = 0.3)
136 draw_route_of_traveling_2(x, y, best_route1_now, best_route2_now, wait = 0.3)
137
```

6.17   Exercise 5

The following script are functions that will be used in **exercise 5**

```r
 1 ▾ sigmoid = function(x) {
 2       x[x < 0] = 0
 3       return(x)
 4 ▴ }
 5
 6 ▾ derivative_sigmoid = function(x) {
 7       x[x > 0] = 1
 8       x[x < 0] = 0
 9
10       return(x)
11 ▴ }
12
13 ▾ f = function (x, alpha_weights, beta_weights, alpha_bias, beta_bias) {
14       sum_ = rep(0, 50)
15 ▾     for (i in 1:length(beta_weights)) {
16           sum_ = sum_ + beta_weights[i] * sigmoid(alpha_weights[i]*x + alpha_bias[i])
17 ▴     }
18
19       return(sum_ + beta_bias)
20 ▴ }
21
22   sse = function(architecture, x, y, alpha_weights, beta_weights,
23 ▾                 alpha_bias, beta_bias) {
24       predicted_values = architecture(x, alpha_weights, beta_weights, alpha_bias, beta_bias)
25
26       return(sum((y - predicted_values)^2))
27 ▴ }
28
29   sgd_update_all_parameters = function(x, y, alpha_weights, beta_weights,
30 ▾                                         alpha_bias, beta_bias, lambda) {
31       # calculating the gradients
32       alpha_weights.grad = rep(0, 50)
33       beta_weights.grad = rep(0, 50)
34       alpha_bias.grad = rep(0, 50)
35       f_value = f(x, alpha_weights, beta_weights, alpha_bias, beta_bias)
36 ▾     for (j in 1:50) {
37           alpha_weights.grad[j] = 2 * sum((y - f_value)
38                                           * (-1) * (beta_weights[j] *
39                                       derivative_sigmoid(alpha_weights[j]*x + alpha_bias[j])*x))
40           beta_weights.grad[j] = 2 * sum((y - f_value)
41                                           * (-1) * sigmoid(alpha_weights[j]*x + alpha_bias[j]))
42           alpha_bias.grad[j] = 2 * sum((y - f_value)
43                                       * (-1) * (beta_weights[j] *
44                                   derivative_sigmoid(alpha_weights[j]*x + alpha_bias[j])))
45 ▴     }
46       beta_bias.grad = 2 * sum(y - f_value) * (-1)
47
48       # updating the values
49       alpha_weights = alpha_weights - lambda * alpha_weights.grad
50       beta_weights = beta_weights - lambda * beta_weights.grad
51       alpha_bias = alpha_bias - lambda * alpha_bias.grad
52       beta_bias = beta_bias - lambda * beta_bias.grad
53
54       batch_error = sse(f, x, y, alpha_weights, beta_weights,
55                           alpha_bias, beta_bias)
56
57       return(list(batch_error, alpha_weights, beta_weights,
58               alpha_bias, beta_bias))
59 ▴ }
```

6.18 Exercise 5b

Implementing the SGD algorithm for the current architecture. The above functions are used in the script (6.17)

```r
1   source('exercise5_functions.R')
2   sparseData = read.table("data/functionEstimationNN.ascii", header=F)
3   colnames(sparseData) = c('x', 'y', 'fGT')
4
5   x = sparseData$x
6   y = sparseData$y
7   fGT = sparseData$fGT
8   n = length(x)
9
10  learning_rate = 1e-4
11  num_epochs = 20
12
13  alpha_weights.current = rnorm(50)
14  beta_weights.current = rnorm(50)
15  alpha_bias.current = rnorm(50)
16  beta_bias.current = rnorm(1)
17
18  batch_size = 50
19  amount_of_batches = n/ batch_size
20
21  # SGD-algorithm
22  test_error = c()
23- for (epoch in 1:num_epochs) {
24-   for (j in 1:amount_of_batches){
25      batch = sample(1:n, batch_size)
26
27      x_batch = x[batch]
28      y_batch = y[batch]
29
30      output = sgd_update_all_parameters(x_batch, y_batch,
31                          alpha_weights.current, beta_weights.current,
32                          alpha_bias.current, beta_bias.current,
33                          learning_rate)
34
35      alpha_weights.current = unlist(output[2])
36      beta_weights.current = unlist(output[3])
37      alpha_bias.current = unlist(output[4])
38      beta_bias.current = unlist(output[5])
39-   }
40
41    epoch_error = sse(f, x, y, alpha_weights.current, beta_weights.current,
42                      alpha_bias.current, beta_bias.current)
43
44    print(sprintf('epoch: %d | error : %f', epoch, epoch_error))
45    test_error = c(test_error, epoch_error)
46
47    # Learning rate scheduler
48    learning_rate = learning_rate * 0.8
49
50- }
51  plot(test_error, type = 'l', ylab = 'epoch_error', xlab = 'epoch')
52
```

7 Bibliography