Computational Statistics Assignment

Dimitris Zerkelidis

STUDENT ID : 03400049

EMAIL : dimzerkes@gmail.com

Postgraduate program Data Science and Machine Learning

Exercise 1

a) Inverse Transformation Method

In this part of the exercise, we will implement the inverse method to simulate samples from $F(x)$ which is the c.d.f. of $f(x)$. Inverse method works, because we sample u from uniform distribution $U(0,1)$ which can be considered as sampling a probability $\in [0,1]$. After that, we would like to return the value x for which it holds $F(X \leq x) \leq u$. That x equals to $F^{-1}(u)$ and x follows F distribution.

Given the p.d.f. $f(x) = 3x^2$ with $0 \leq x \leq 1$, we will simulate 10000 samples using Inverse Transformation Method. Hence, we will be using the c.d.f. to simulate values. To find the c.d.f. of $f(x)$ we will calculate $F(x) = \int_0^x f(x)dx = x^3$ and $0 \leq x \leq 1$. Now we can do the inverse transformation as below:

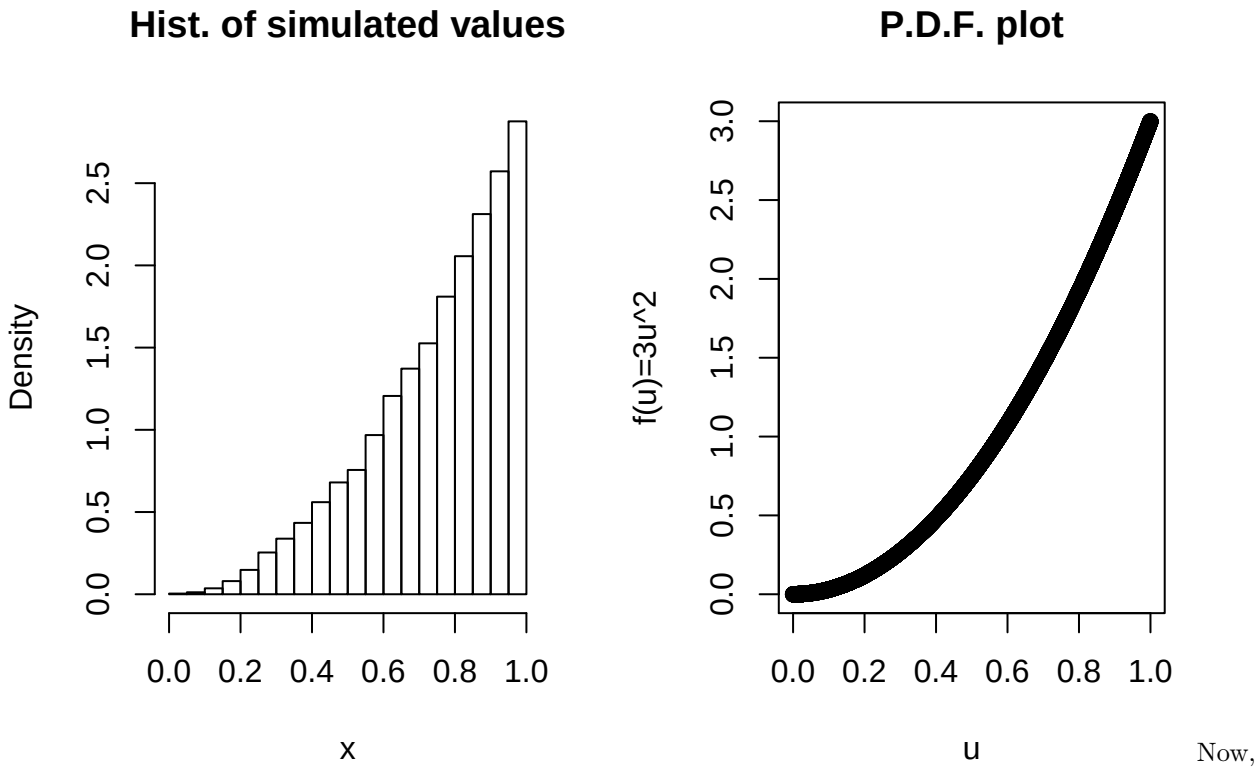
$x = F^{-1}(u)$ so $u = x^3$ <=> $x = u^{1/3}$ or $F^{-1}(u) = u^{1/3}$.

Pseudocode of Inverse Transformation Method

```
u ~ U[0,1]
x = u^{1/3}
then x ~ F(x)
```

The code below is about comparing histogram of simulated values and the P.D.F. plot of $f(x)$. We can see that both plots are very similar. That means that our simulation worked correctly.

```
par(mfrow=c(1,2))
u = runif(10000, min = 0, max = 1)
x = u^(1/3)
f_u = 3*u^2
hist(x,prob=TRUE,main="Hist. of simulated values")
plot(u,f_u,main="P.D.F. plot",ylab="f(u)=3u^2")
```



Now,

we will compare mean and standard deviation results from the simulated values with the theoritical ones from the p.d.f. function. Firstly, we calculate theoritical mean on $0 \leq x \leq 1$ space.

$$E[f(x)] = \int_0^1 f(x)xdx = \int_0^1 x * 3x^2 dx = \int_0^1 3x^3 dx = [\frac{3x^4}{4}]\Big|_1^0 = 0.75$$

After that, we will calculate the theoritical standard deviation.

$$sd = (E[f(x^2)] - E[f(x)]^2)^{1/2} = (\int_0^1 x^2 * 3x^2 dx - E[f(x)]^2)^{1/2} =$$

$$(3\int_0^1 (\frac{x^5}{5})'dx - 0.75^2)^{1/2} = ([3 * \frac{x^5}{5}]\Big|_1^0 - 0.5625)^{1/2} = (\frac{3}{5} - 0.5625)^{1/2} = 0.19364916731$$

The above result proves us that the standard deviation of the simulated values is really close to the theoritical one. The outputs from the simulated values are shown below.

```
cat('Mean of our algorithm',mean(x))
```

```
## Mean of our algorithm 0.7506223
```

```
cat('Sd of our algorithm',sd(x))
```

```
## Sd of our algorithm 0.1935766
```

Clearly, the algorithm's results and the theoritical ones are very similar.

b) Rejection Sampling

In this sub-question we will implement the same simulation as above but with the use of Rejection Sampling. Then we will compare the histogram of the simulated plot with the p.d.f. plot as before. Also we will check how close is the mean and standard deviation of the simulation with the theoritical ones.

For this method, we are called to find a density function $g(y)$, combined with a variable M, that is positive such that it holds for the envelope function which is $G(y) = Mg(y)$, the following:

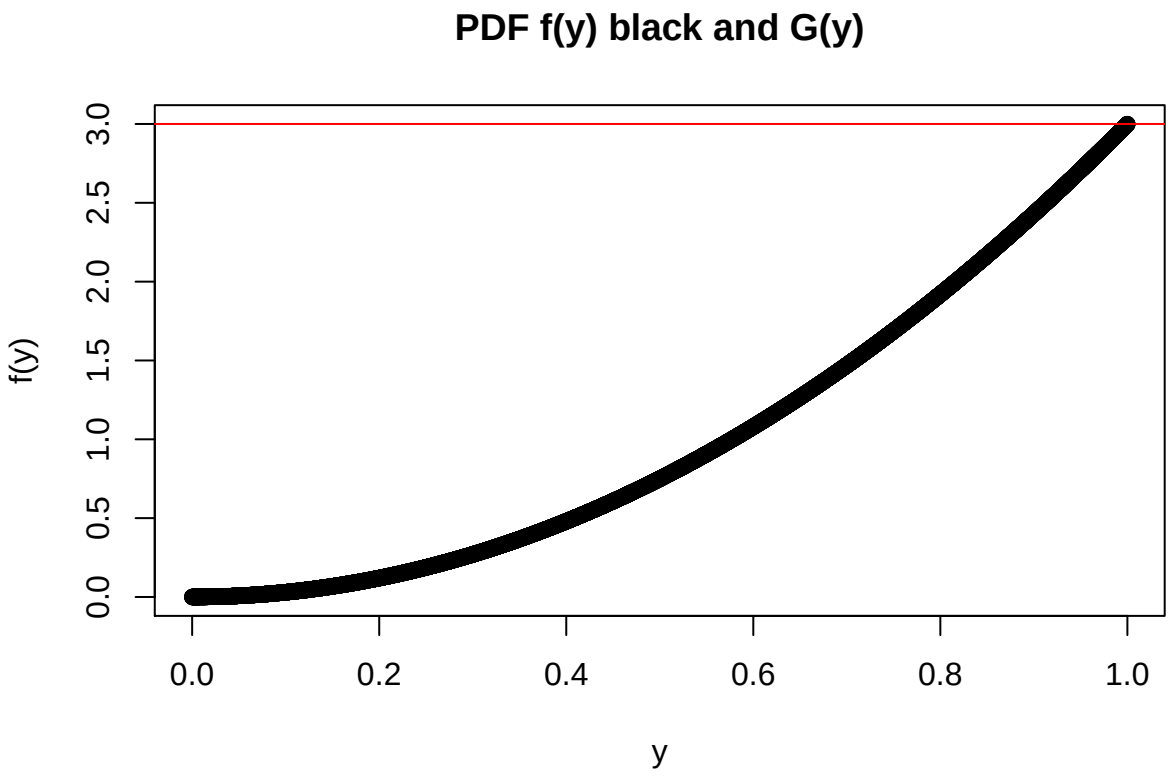
$$f(y) \leq G(y)$$

Firstly we need to choose a p.d.f $g(y)$ that we can sample from. This will be the uniform U[0,1]. So $g(y) = \frac{1}{1-0} = 1$. Then, we need to find the M such as $M = max_y \frac{f(y)}{g(y)} = max_y \frac{3y^2}{1} = 3$, because y is between 0 and 1. So the max value will be for y = 1. Hence, we will get M = 3.

Now that we know our envelope function $G(y) = M * g(y) = 3 * 1 = 3$ that covers f(y) we can find the acceptance probability which equals to $f(y)/G(y) = f(y)/3$.

Below we can see how our envelope function $G(y) = 3$ with $0 \leq y \leq 1$ covers the p.d.f. function that we are interested in. We have to be careful in the choice of the envelope function so that it covers f but also being as much close to f as it can be.

```r
plot(u,f_u,main="PDF f(y) black and G(y)",xlab="y",ylab="f(y)")
abline(h=3,col='red')
```



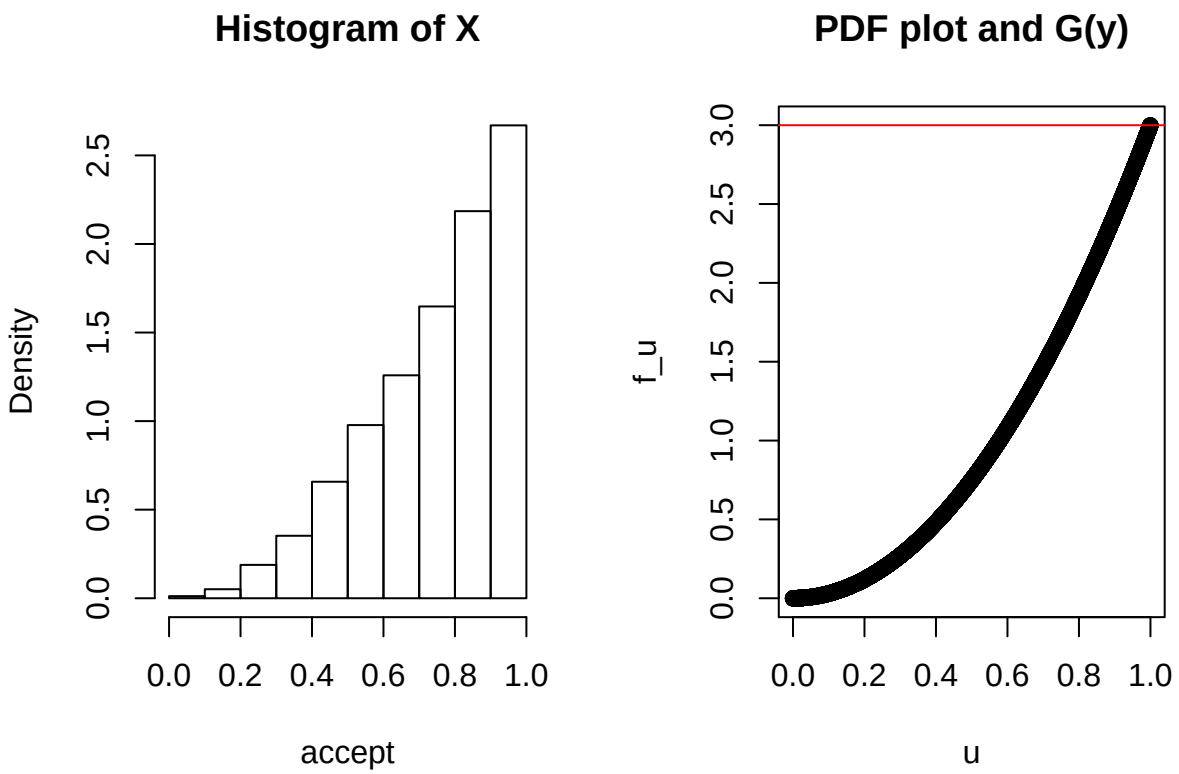
Pseudocode of Rejection Sampling method

The intuition of this method is producing samples under the curve of the envelope function G and accept only those that lie under the area of the f function.

```
u ~ U[0,1]
u_comp ~ U[0,1]
if u_comp <= f(u)/3 accept u
else reject
endif
Repeat 10000 times
```

```r
u = runif(10000,0,1)
accept = c()
for(i in 1:length(u)){
 u_comp = runif(1, 0, 1)
 if(u_comp <= (3*u[i]^2)/3*1)  {
   accept = append(accept,u[i],after=length(accept))
 }
}

f_u = 3*u^2
par(mfrow=c(1,2))
hist(accept,prob=TRUE,main = 'Histogram of X')
plot(u,f_u,main="PDF plot and G(y)")
abline(h=3,col='red')
```



Below, we can see that the mean and the standard deviation that we observed using this method is very close to the ones of the previous method and also the theoritical values that we found, which were 0.75 and around 0.1937.

```
cat('Rej. Sampling mean',mean(accept))
```

```
## Rej. Sampling mean 0.7486049
```

```
cat('Rej. Sampling s.d.',sd(accept))
```

```
## Rej. Sampling s.d. 0.1926955
```

Finally, we would like to check the theoritical acceptance probability which can be computed as the ratio of the reas ander the two curves. So we define it as follows:

$$\int_0^1 f(x)/Mg(x)dx = \int_0^1 \frac{3x^2}{3}dx = \int_0^1 x^2 dx = \int_0^1 (x^3/3)'dx = \frac{1}{3} = 0.33$$

While our algorithm will have a very similar probability. We calculate our algorithm's probability as $Accepted_samples/10000$ which was can be seen in the next R code.

```
cat('Acceptance Probability of our algorithm: '
      ,length(accept) / 10000)
```

```
## Acceptance Probability of our algorithm:  0.3345
```

Again algorithm's and theoritical results are very close to each other.

c) Importance Sampling

In the last sub-question we are going to use the method of Importance Sampling to calculate the following integral:

$$I = \int_{38}^{\infty} e^{-x} x^2 dx$$

Before that, we will calculate it by hand. So we have:

$$I = \int_{38}^{\infty} e^{-x} x^2 dx = -e^{-x} x^2 - \int_{38}^{\infty} -2e^{-x} x dx =$$

$$-e^{-x} x^2 - (-2(-e^{-x} x - e^{-x})) = [-e^{-x} x^2 + 2(-e^{-x} x - e^{-x})]\Big|_{38}^{\infty} =$$

$$0 + 4.7777601e - 14 = 4.7777601e - 14$$

This method is used when we want to calculate integrals of this form $I = \int f(x)p(x)dx = E[f(x)]$, where x $\sim p(x)$. But sometimes it is difficult to sample from $p(x)$. That's why we introduce the important function $g(x)$ that is non-zero whenever $p(x)$ is non-zero so that we avoid division by 0. Then, we can transform I as follows:

$$I = \int \frac{f(x)p(x)}{g(x)} g(x)dx = E[\frac{f(x)p(x)}{g(x)}]$$

where $x \sim g(x)$.

This means that we can estimate I as:

$$\hat{I} = \frac{1}{m} \sum_{i=1}^{m} \frac{f(x_i)p(x_i)}{g(x_i)}$$

when x $\sim g(x)$ instead of $\frac{1}{m} \sum_{i=1}^{m} f(xi)$ where x $\sim p(x)$

In our case $f(x) = e^{-x} x^2$ and $p(x) = I_{(x \geq 38)}$. Now we will take two cases for two important function as we are asked. Below we are showing the pseudocode with the description of the algorithm's steps.

Pseudocode

```
1. Find G which is the cdf of g
2. Inverse G
3. Generate 1000 samples ~ U(0,1)
4. Calculate I_hat as defined previously

Repeat steps 3 and 4, 100 times and return the mean and variance
of the results
```

First important function: $g(x) = e^{-(x-38)}$

We will use the inverse transformation method to sample from $g(x)$ and calculate with importance sampling the integral that we are interested in. First, we will find the c.d.f of $g(x)$ for $x \geq 38$.

$$G(x) = \int_{38}^{\infty} e^{-x+38} dx = [-e^{-x+38}]\Big|_{38}^{x} = -e^{-x+38} + e^{0} = -e^{-x+38} + 1$$

Then, we set $x = G^{-1}(u)$ so that

$$u = -e^{-x+38} + 1 <=> e^{-x+38} = 1 - u <=>$$
$$-x + 38 = ln(1-u) <=> x = 38 - ln(1-u)$$

```
integral_list = c()
for(i in 1:100){
  u = runif(1000, min = 0, max = 1) ## generating 1000 values from uniform ~ [0,1]
  x = 38 - log(1-u) # sampling from g(x)
  x = x[x>38] # unecessary cause all values in X are higher than 38 ..
  fraction = 0
  for(i in 1:length(x)){
    fraction = fraction + (exp(-(x[i]))*x[i]^2)/(exp(-x[i]+38)) # calc.. Integral
  }
  I =  fraction/length(x) # divide by M=1000
  integral_list = append(integral_list,I,after=length(integral_list))
}
first_g_avg = mean(integral_list)
first_g_var = var(integral_list)
```

Second important function called pareto distribution: $g(x) = 4 * 38^4 * x^{-5}$

Again by using the inverse transformation method to sample from G(x), we proceed to the following calculations.

$$G(x) = \int_{38}^{\infty} 4 * 38^4 x^{-5} dx = [-(\frac{38}{x})^4]\Big|_{38}^{x} = 1 - (\frac{38}{x})^4$$

Then,

$$u = 1 - (\frac{38}{x})^4 <=> u - 1 = -\frac{38^4}{x^4} <=> x^4 =$$

$$-\frac{38^4}{u - 1^4} <=> x^4 = \frac{38^4}{1-u} <=>$$

$$x = \frac{38}{(1-u)^{1/4}}$$

```
integral_list = c()
for(i in 1:100){
  u = runif(1000, min = 0, max = 1) ## generating 1000 values from uniform ~ [0,1]
  x = 38/((1-u)^(1/4)) # sampling from g(x)
  x = x[x>38] # unecessary cause all values in X are higher than 38 ..
  fraction = 0
  for(i in 1:length(x)){
    fraction = fraction + (exp(-(x[i]))*x[i]^2)/(4*38^4*(x[i]^(-5)))  # calc.. Integral
  }
  I =  fraction/length(x)  # divide by M=1000
  integral_list = append(integral_list,I,after=length(integral_list))
}
second_g_avg = mean(integral_list)
second_g_var = var(integral_list)
```

Conclusion ~ comparing results

We observe that the by using the first important function $g(x) = e^{-(x-38)}$ we get a mean value which is very close to I's value calculated by hand 4.7777601e-14. Then, by using the second important function which is the pareto distribution we found a mean which is also very close to I. Although, both results are very close, we can observe in the below outputs that the 1st important function provides better estimation with lower variance than the 2nd, which makes it more stable too.

```
cat('Mean value of I from 1st g(x): ', first_g_avg)
```

```
## Mean value of I from 1st g(x):  4.777606e-14
```

```
cat('Variance of I from 1st g(x): ', first_g_var)
```

```
## Variance of I from 1st g(x):  4.706369e-33
```

```
cat('Mean value of I from 2nd g(x): ', second_g_avg)
```

```
## Mean value of I from 2nd g(x):  4.776329e-14
```

```
cat('Variance of I from 2nd g(x): ', second_g_var)
```

```
## Variance of I from 2nd g(x):  8.455396e-30
```

```
cat('I value: 4.7777601e-14')
```

```
## I value: 4.7777601e-14
```

Now, we can check how the estimations differ from the real value of I. Also, we can see exactly how much the variances of the two important functions differ from each other.

```
cat('1st important function differs from I for: ', abs(first_g_avg - 4.7777601e-14) )
```

```
## 1st important function differs from I for:  1.541511e-18
```

```r
cat('2nd important function differs from I for: ', abs(second_g_avg - 4.7777601e-14))
```

```
## 2nd important function differs from I for:  1.430776e-17
```

```r
cat('Variance difference of 2 imp. Functions ', abs(first_g_var - second_g_var))
```

```
## Variance difference of 2 imp. Functions  8.45069e-30
```

Exercise 2

In this part of the assignment we will simulate 50 values using rnorm and then with these values we will estimate the density with the epanechnikov kernel. Kernel Density Estimation is a non-parametric method to estimate the probability density of a $r.v.$. We define kernel density estimator as follows:

$$\hat{f} = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{x - x_i}{h})$$

Where K is the kernel – a non-negative function – and h > 0 is a smoothing parameter called the bandwidth.

Now we need to choose the optimal h, which we will do with the use of leave-one-out cross validation. The likelihood for the density estimation with a specific value of h is:

$$L(h) = \prod_{i=1}^{n} f(x_i)$$

In the case of leave-one-out cross validation the likelihood is calculated as:

$$LCV(h) = \prod_{i=1}^{n} \hat{f}(x_{h,i}) = \sum_{i=1}^{n} log[\hat{f}(x_{h,i})]$$

Where the estimate $\hat{f}(x_{h,i})$ on the subset $X_{j_{i!=j}}$ can be written in the following way:

$$\hat{f}(x_{h,i}) = \frac{1}{(n-1)h}\sum_{j=1,j!=i}^{n} K(\frac{x_j - x_i}{h})$$

This means that

$$LCV(h) = \sum_{i=1}^{n} log(\frac{\sum_{j=1,j!=i}^{n} K(\frac{x_j - x_i}{h})}{(n-1)h}) =$$

$$\sum_{i=1}^{n} log(\sum_{j=1,j!=i}^{n} K(\frac{x_j - x_i}{h})) - \sum_{i=1}^{n} log[(n-1)h] =$$

$$\sum_{i=1}^{n} log(\sum_{j=1,j!=i}^{n} K(\frac{x_j - x_i}{h})) - n * log[(n-1)h]$$

Finally, we will chose the optimal h, hopt as:

$$argmax_h LCV(h) = argmax_h \sum_{i=1}^{n} log(\sum_{j=1, j!=i}^{n} K(\frac{x_j - x_i}{h})) - n * log[(n-1)h]$$

In our exercise n=50 and the kernel is epanechnikov, which means that $K(\frac{x_j - x_i}{h}) = 0.75 * (1 - (\frac{x_j - x_i}{h})^2)$, because $K(z) = 0.75 * (1 - z^2)$ for $|z| <= 1$ otherwise $K(z) = 0$.

```
x = rnorm(50, mean = 0, sd = 1) # standarized normal distr.
best_likelihood = -9999
hopt = -1 # randomly
for (h in seq(0.01,5,0.01)){
sum_i = 0
for (i in 1:50){
  sum_j = 0
  for (j in 1:50){
  if (j!=i){
    z = (x[j]-x[i])/h
    if (abs(z)<=1){
      sum_j = sum_j + 0.75*(1-z^2)
    }
  }
  }
  sum_i = sum_i + log(sum_j)
}
likelihood = (sum_i) - 50*log((50-1)*h)

if (likelihood >= best_likelihood){
  best_likelihood = likelihood
  hopt = h
}
}
hopt
```

```
## [1] 1.05
```

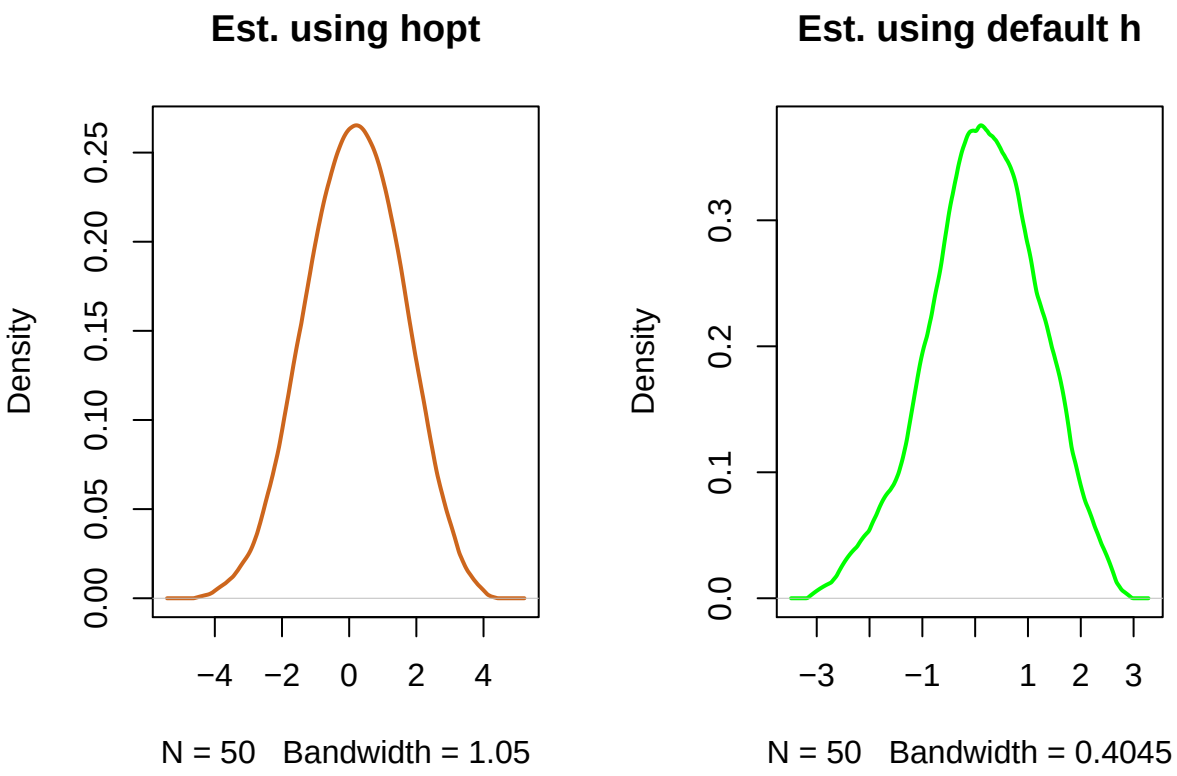
After we calculated the optimal h, we can plot for that value the density estimation of the p.d.f. using epanechnikov kernel. We can see that we have a very good approximation of the density of our generated data. Since our data are generated from a standard normal distribution, our density estimation will look very similar to that distribution.

Below in the first plot, we see our estimation using epanechnikov kernel using the optimal h that we found before. We observe that is very close and smooth to the normal standard distribution as it should be.

Then the second plot, is the same as the first with the difference of using a default h so that we can see how different the output is. Now we are not getting a smooth estimation of the normal standard distribution.

```
par(mfrow=c(1,2))
plot(density(x,kernel="epanechnikov",bw=hopt),lwd=2,col="chocolate3",
                     main="Est. using hopt")

plot(density(x,kernel="epanechnikov"),lwd=2,col="green",
               main="Est. using default h")
```



Exercise 3

The EM algorithm is used for obtaining maximum likelihood estimates of parameters when some of the data is missing or censored. More generally, however, the EM can also be applied when there is latent, i.e. unobserved data which was never intended to be observed in the first place.

Our problem consists of a sample of length $n = 20$ and we have observed $m = 16$ values while $k = 4$ values are censored, although we know that their value is higher than 1.7. We consider that the sample $n = (m, k)$ $\sim exp(\theta)$ with $\theta > 0$. Our goal is to estimate the parameter θ. In such cases, that we have observed and latent(or censored) data it is appropriate to use the Expectation-Maximization(EM) algorithm. Firstly, we will describe our theoritical model and then pseudocode and code will follow.

The EM algorithm consists of the following two steps. The E-step(expectation step), which estimates the censored data k given the observed data m and the current distribution parameter θ^r. Then it performs the M-step(Maximization), which maximizes over the new parameter θ the quantity that was calculated from E-step.

Finally, we will plot the log likelihood of the observed data over the range of iterations until the convergence of EM, so that we see that it increases.

Notations

Before the analysis we will provide the following notations:

1. $f(x) = \theta e^{-\theta x}$ the p.d.f. of exponential distribution
2. $F(x) = 1 - e^{-\theta x}$ the c.d.f. of exponential distribution

log likelihood of the observed data

At first, we will define the likelihood of the completed data(propotionally) as:

$$L(\theta|m,k) \propto \prod_{i=1}^{n} f(n_i|\theta) = \prod_{i=1}^{16} f(m_i|\theta) \prod_{i=1}^{4} f(k_i|\theta)$$

Then we define the observed data likelihood by integrating the previous likelihood with respect to $k \in (1.7, \infty)$.

$$L(\theta|m) \propto \int_k L(\theta|m,k)dk =$$

$$\prod_{i=1}^{16} f(m_i|\theta) \int_{1.7}^{\infty} \prod_{i=1}^{4} f(k_i|\theta)dk = \prod_{i=1}^{16} f(m_i|\theta) \prod_{i=1}^{4} 1 - F(1.7|\theta)$$

Thus, the log likelihood of the observed data is defined as following:

$$log(L(\theta|m)) \propto \sum_{i=1}^{16} log(\theta e^{-\theta m}) + \sum_{i=1}^{4} log(e^{-1.7\theta})$$

or

$$log(L(\theta|m)) \propto 16log\theta - \theta \sum_{i=1}^{16} m_i - 6.8\theta$$

E-step

We will denote $Q(\theta, \theta^r)$ as the expected value of the log likelihood function of parameter θ with respect to the current condistional distribution of k given m and current distribution parameter θ^r. Our exercise considers that $n = (m, k) \sim exp(\theta)$ Hence:

$$Q(\theta|\theta^r) = E_{k|m,\theta^r}[logL(\theta; m, k)] =$$

$$E_{k|m,\theta^r}[\sum_{i=1}^{16} logf(m_i) + \sum_{i=1}^{4} logf(k_i)] =$$

$$\sum_{i=1}^{16} logf(m_i) + E_{z|y,\theta^r}[\sum_{i=1}^{4} log\theta e^{-\theta k}] =$$

$$\sum_{i=1}^{16} log + \sum_{i=1}^{16} loge^{-\theta x} + E_{z|y,\theta^r}[\sum_{i=1}^{4} log\theta e^{-\theta k}] =$$

$$16log\theta - \theta \sum_{i=1}^{16} m_i + \sum_{i=1}^{4} E_{z|y,\theta^r}[log\theta e^{-\theta k}] =$$

$$16log\theta - \theta \sum_{i=1}^{16} m_i + 4log\theta - 4\theta E_{z|y,\theta^r}[k_i]$$

It is important in this step to calculate the expectation of k, given that its values are higher than 1.7 ($k > 1.7$) and that it follows an truncated exponential distribution with domain space $(1.7, \infty)$ with the current parameter θ^r. Therefore we have:

$$E_{k|m,\theta^r}[k] = \frac{\theta^r}{1 - F(1.7)} \int_{1.7}^{\infty} k e^{-\theta^r k}$$

After calculating the integral we can write the above equation as:

$$E_{k|m,\theta^r}[k] = \frac{\theta^r}{1 - F(1.7)} * [\frac{1.7\theta^r e^{-1.7\theta^r} + e^{-1.7\theta^r}}{\theta^{r^2}}] =$$

$$\frac{\theta^{r^2} 1.7 e^{-1.7\theta^r} + \theta^r e^{-1.7\theta^r}}{\theta^{r^2} e^{-1.7\theta^r}} = 1.7 + \frac{1}{\theta^r}$$

Finally, we estimate the E-step's Q function by combining the previous equations.

$$Q(\theta, \theta^r) = 20 log\theta - \theta \sum_{i=1}^{16} m_i - 6.8\theta - \frac{4\theta}{\theta^r}$$

M-step

Now we will compute the new $\theta^{r+1} = argmax Q(\theta; \theta^r)$ by taking the derivative of Q function and set it to zero.

$$Q'(\theta; \theta^r) = 0 <=> \frac{20}{\theta} - \sum_{i=1}^{16} m_i - 6.8 - \frac{4}{\theta^r} = 0 <=>$$

$$\theta = \frac{20}{\sum_{i=1}^{16} m_i + 6.8 + \frac{4}{\theta^r}}$$

That means that our new parameter estimation equals :

$$\theta^{r+1} = \frac{20}{\sum_{i=1}^{16} m_i + 6.8 + \frac{4}{\theta^r}}$$

Pseudocode for EM

"' $\theta^0 = 1/mean(m)$, we initialize our θ^0 with the mean of the observed values. It is like implementing a maximum likelihood estimation on this parameter with the known data.

Then, we estimate the new θ using the M-step, to update our estimation. The E-step was calculated once we use its result immediately every time on M-step. $\theta^1 == \frac{20}{\sum_{i=1}^{16} m_i + 6.8 + \frac{4}{\theta^0}}$

Run the code below until convergence to a defined tolerance while $(\theta^{r-1} - \theta^r)^2 > tol$:

$\theta^{r-1} = \theta^r$ update old parameter with new

perform M-step and find the new parameter $\theta^r = \frac{20}{\sum_{i=1}^{16} m_i + 4 \frac{1.7\theta^{r-1}+1}{\theta^{r-1} e^{1.7\theta^{r-1}}}}$

calculate the log likelihood of observed data -> $log(L(\theta|m)) \propto 16 log\theta - \theta \sum_{i=1}^{16} m_i - 6.8\theta$

"'

For the EM algorithm we will use a tolerance of 10^{-10}.

```r
m = c(0.05511269, 0.07437246, 0.11098159, 0.13552999,
0.20371014, 0.22090690, 0.23699706, 0.27435875,
0.28669966, 0.47155521, 0.96822690, 0.97200651,
1.04514368, 1.37989648, 1.49109121, 1.53370336)

theta_0 = 1/mean(m)
sum_m_obs = sum(m)

#calculate log likelihood of observed data with theta_0
ll.0 = 16*log(theta_0) - theta_0*(sum_m_obs  + 4*1.7)
cat('in iter 0:',' theta', theta_0, '\n')
```

```
## in iter 0:  theta 1.69128
```

```r
theta_old= theta_0
theta_new = 20 / (sum_m_obs + 6.8 + (4/theta_old))
cat('in iter 1:',' theta', theta_new, '\n')
```

```
## in iter 1:  theta 1.073804
```

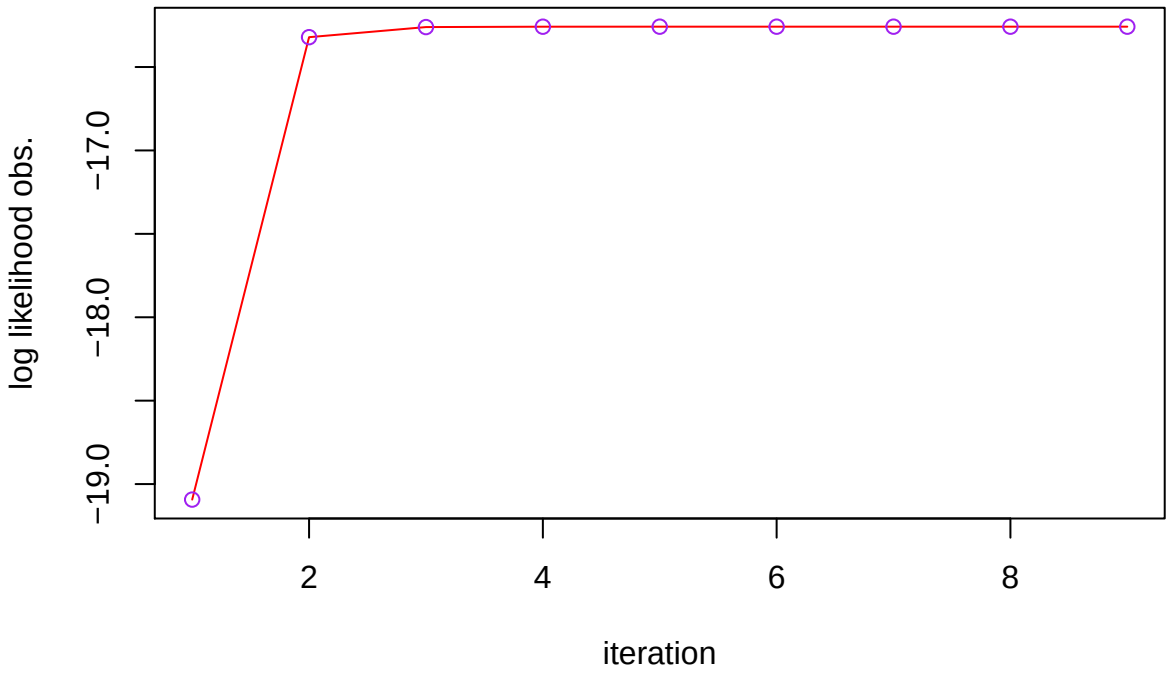
```r
#likelihood after 1st iteration
ll =  16*log(theta_new) - theta_new*(sum_m_obs  + 4*1.7)
lls= c(ll.0,ll)  #list of log likelihoods

while ((theta_new-theta_old)**2 > 10**(-10)) {
  theta_old = theta_new
  theta_new = 20 / (sum_m_obs + 6.8 + (4/theta_old))

  ll = 16*log(theta_new) - theta_new*(sum_m_obs + 4*1.7)
  lls = append(lls,ll,after=length(lls))
  cat('in iter:', length(lls)-1, ', theta', theta_new, '\n')
}
```

```
## in iter: 2 , theta 1.000732
## in iter: 3 , theta 0.9872952
## in iter: 4 , theta 0.984651
## in iter: 5 , theta 0.9841238
## in iter: 6 , theta 0.9840185
## in iter: 7 , theta 0.9839974
## in iter: 8 , theta 0.9839932
```

```r
par(mfrow=c(1,1))
plot(seq(1,length(lls),1),lls,col='purple',
     lines(seq(1,length(lls),1),lls,col='red'),
     xlab = 'iteration', ylab = 'log likelihood obs.')
```



From the algorithm's output we see that we converged to $\theta = 0,9839932$ while doing 8 iterations. Then see that the log likelihood of the observed data has a rapid increase over the first iteration of EM and then gradually converges to $-16,25820$.

Exercise 4

In the last part of our assignment, we are going to study the case of variable selection problem for a multiple linear regression model. The model, which we are going to study has 15 predictor variables($p = 15$) fitted on $n = 50$ samples.

We will use rnorm with zero mean value($\mu = (0, .., 0)$) and identical covariance matrix($\Sigma = I$) to simulate the first 10 predictors from the multidimensional standard normal distribution. So we have $X = (X1, ...X10) \sim N(\mu, \Sigma)$ or equivalently $X_n \sim N(0, 1)$ for $n \in [0, 1]$. Below we can see the part of the code which executes what we stated.

```
n=50 # samples
samples = rnorm(n*10) # for 10 predictors
x_1_10 = matrix(samples, nrow = 50, byrow = TRUE)
```

The rest predictor variables will be simulated as follows:

$X_{ij} \sim N(0.2X_{i1} + 0.4X_{i2} + 0.6X_{i3} + 0.8X_{i4} + 1.1X_{i5}, 1)$, $j = 11, ..., 15$ $i = 1, .., 50$

```
x_11_15 = matrix(0, nrow=50, ncol=5)# 50,5 matrix with 0.
for (i in 1:nrow(x_1_10)){
  mean = 0.2*x_1_10[i, 1] + 0.4*x_1_10[i, 2] + 0.6*x_1_10[i, 3] +
          0.8*x_1_10[i, 4] + 1.1*x_1_10[i , 5]

  x_11_15[i, ] = rnorm(5, mean= mean, sd=1)
}
#bind columns
x<- cbind(x_1_10, x_11_15) # 50x15 size
```

Finally, we will simulate the response variable Y according to:

$Y_i \sim N(4 + 2X_{i1} - X_{i5} + 2.5X_{i7} + 1.5X_{i11} + 0.5X_{i13}, 1.5^2)$, $i = 1, ..50$

```
y = matrix(0, nrow=50, ncol=1)
for (i in 1:nrow(x)){
  mean_y = 4 + 2*x[i, 1] - x[i, 5] + 2.5*x[i, 7] +
           1.5*x[i, 11] + 0.5*x[i, 13]

y[i] = rnorm(1, mean= mean_y, sd=1.5) # 50,1
}
```

In the sequel, we can consider the following multiple linear model:

$Y = \beta + \beta_0 X_1 + ... + \beta_{15} X_{15} + \epsilon$, $\epsilon \sim N(0, \sigma^2)$

```
pred_vars =as.data.frame(x)
colnames(pred_vars) =c("X1", "X2", "X3", "X4", "X5","X6", "X7", "X8",
                       "X9", "X10","X11", "X12", "X13", "X14", "X15")

full_mod =  lm(y~ ., data=pred_vars)
summary(full_mod)
```

```
##
## Call:
## lm(formula = y ~ ., data = pred_vars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9952 -0.7642  0.0836  0.6608  3.6212
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  3.98217    0.25081  15.877  < 2e-16 ***
## X1           2.38635    0.30403   7.849 3.88e-09 ***
## X2          -0.05094    0.36329  -0.140   0.8893
## X3          -0.03734    0.44904  -0.083   0.9342
## X4           0.48861    0.55647   0.878   0.3861
## X5          -0.89809    0.70223  -1.279   0.2096
## X6           0.28666    0.24495   1.170   0.2500
## X7           2.16831    0.35009   6.194 4.83e-07 ***
## X8           0.03746    0.25755   0.145   0.8852
## X9          -0.12612    0.29777  -0.424   0.6746
## X10          0.22971    0.24370   0.943   0.3525
## X11          1.38062    0.23139   5.967 9.52e-07 ***
## X12          0.46211    0.31309   1.476   0.1492
## X13          0.35041    0.23269   1.506   0.1413
## X14          0.21331    0.24162   0.883   0.3835
## X15         -0.59986    0.29688  -2.021   0.0513 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.6 on 34 degrees of freedom
## Multiple R-squared:  0.9154, Adjusted R-squared:  0.8781
## F-statistic: 24.54 on 15 and 34 DF,  p-value: 6.364e-14
```

(i)

Here, we are asked to implement a function to perform a variable selection over all possible models. This means $2^p = 2^{15}$ number of models. We will choose that model with the best preddictor variables according to BIC metric. BIC is defined as: $nln(RSS/n) + kln(n)$ where n is the number of data samples, k is the number of model parameters and RSS is the residual sum of squares.

Below we can see the code that implements what we described.

```
minBIC = +Inf # init very small..
for (i in 1:15) {
    #generate all combinations
    #of 1:15 taken i every time
    combs = combn((1:15), i)# dim(combn(n, i)) == c(i, choose(n, i))
    for (j in 1:ncol(combs)){
        predictors = x[, combs[, j]]#bring all rows of a column j in combs
        mod = lm(y~predictors)
        mod_BIC = BIC(mod)
    if (mod_BIC < minBIC){
      best_model = mod
      best_predictors = combs[, j]
      minBIC = mod_BIC
    }
  }
}
cat('Minimum bic is: ', minBIC, '\n')
```

```
## Minimum bic is:  209.7375
```

```
cat('Best predictor is: ', best_predictors, '\n')
```

```
## Best predictor is:  1 5 7 11 13
```

On the previous output we can see what is the minimum BIC and the best variables to select. Below we show a description of the best model.

```
#renaming the columns making it "Xi"
col_names_list = c()
for (i in 1:length(best_predictors)){
  col_name = paste(c("X",best_predictors[i]),collapse="")
  col_names_list = append(col_names_list,col_name,after=length(col_names_list))
}

#best model with min BIC description
pred_vars = as.data.frame(x[, best_predictors])
colnames(pred_vars) =col_names_list
best_mod= lm(y~ ., data=pred_vars)
summary(best_mod)
```

```
##
## Call:
## lm(formula = y ~ ., data = pred_vars)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2754 -1.0611 -0.1029  1.0487  4.3622
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.0416     0.2378  16.995  < 2e-16 ***
## X1            2.3632     0.2804   8.427 9.90e-11 ***
## X5           -1.2440     0.3057  -4.070 0.000193 ***
## X7            2.3105     0.3041   7.599 1.53e-09 ***
## X11           1.6160     0.1793   9.013 1.49e-11 ***
## X13           0.4473     0.1853   2.414 0.019983 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.598 on 44 degrees of freedom
## Multiple R-squared:  0.8909, Adjusted R-squared:  0.8785
## F-statistic: 71.84 on 5 and 44 DF,  p-value: < 2.2e-16
```

We observe from the summary of the model we found, that all p values are significantly small(on both p-test and f-test), which means we can reject the null hypothesis for every coefficient and the model without any coefficient at all(all variables are zero.). Also, we can check that our results are mostly the variables we used to simulate the response Y.

(ii)

Next, we are going to apply the $LASSO$ method for variable selection instead of the previous method we implemented using the BIC metric. $LASSO$ is mostly used in models, which one independent variable can be linearly predicted from others (Multicollinearity). The way we have generated our model is exactly this case. This is why we believe that $LASSO$ will perform very well.

Description over LASSO method

The method we are discussing for variable selection penalizes the $l1$ norm of the model's coefficients leading to their shrinkage. In $LASSO$ method some coefficients can shrink to zero if these independent variables are less important, which can work also as a variable selection. What it does is to add a penalty term to the $OrdinaryLeastSquares(OLS)$ function to be minimized. Specifically, we are looking for the coefficients β that minimizes

$(y - Z\beta)^\top (y - Z\beta) + \lambda \sum_{i=1}^{p} |\beta_i|$ where $p = 15$ in our case and Z is the matrix of standarised covariates and λ is a tuning parameter which controls the amount of regularization.A large enough λ will set some coefficients exactly to zero, so $LASSO$ is like performing variable selection. Finally, in this method there is a shrinkage factor:

$s = \frac{\sum_{j=1}^{p} |\beta_j|}{max \sum_{j=1}^{p} |\beta_j|}$ where $s \in [0, 1]$.

that presents the ratio of the sum of the absolute current estimate over the sum of the absolute OLS estimates and takes values in $[0, 1]$; when is equal to 1 there is no penalization and we have the OLS solution and when is equal to 0 all the β_js are equal to zero.

implementaton

After explaining the theoritical part of $LASSO$ method now we will use an iterative algorithm to estimate coefficients as $l1$ norm increases or equivalently the tuning parameter λ decreases.

```
library(glmnet) # package for LASSO
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

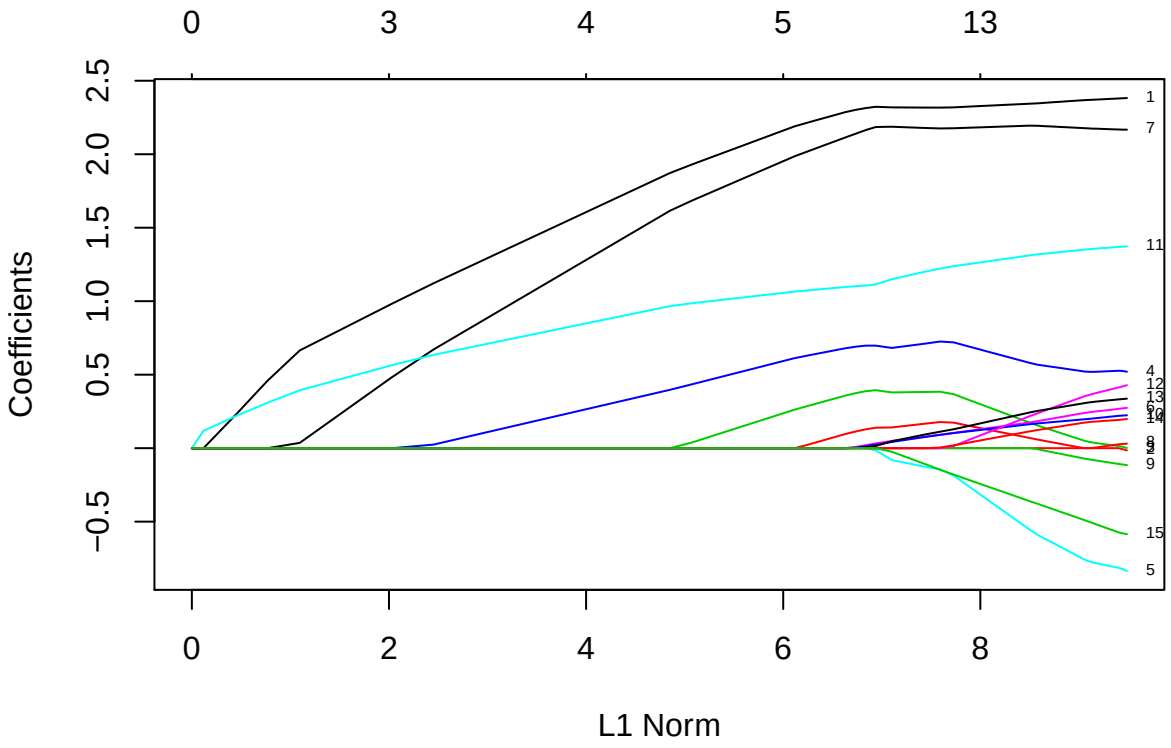
```
## The following objects are masked from 'package:pracma':
##
##     expm, lu, tril, triu
```

```
## Loaded glmnet 4.0
```

```
lasso <- glmnet(x, y)
```

By the plot below, we can observe that the coefficients that 'survive' in our model are very close to what we found in BIC's exhaustive search or the ones in the model that generated our Y variables. Each line of the graph corresponds to a coefficient β_j. At the end of each line you can see an indicator with the number of the coefficient. Also the x-axis line shows the L1 Norm as it increases and the upper horizontal line shows the number of coefficients included in the model based on L1 norm's value. We can notice that for $L1$ norm equal to zero we have no coefficients and as L1 norm increases, which means λ decreases the solution reaches the OLS one.

```
par(mfrow=c(1,1))
plot(lasso, label=TRUE)
```



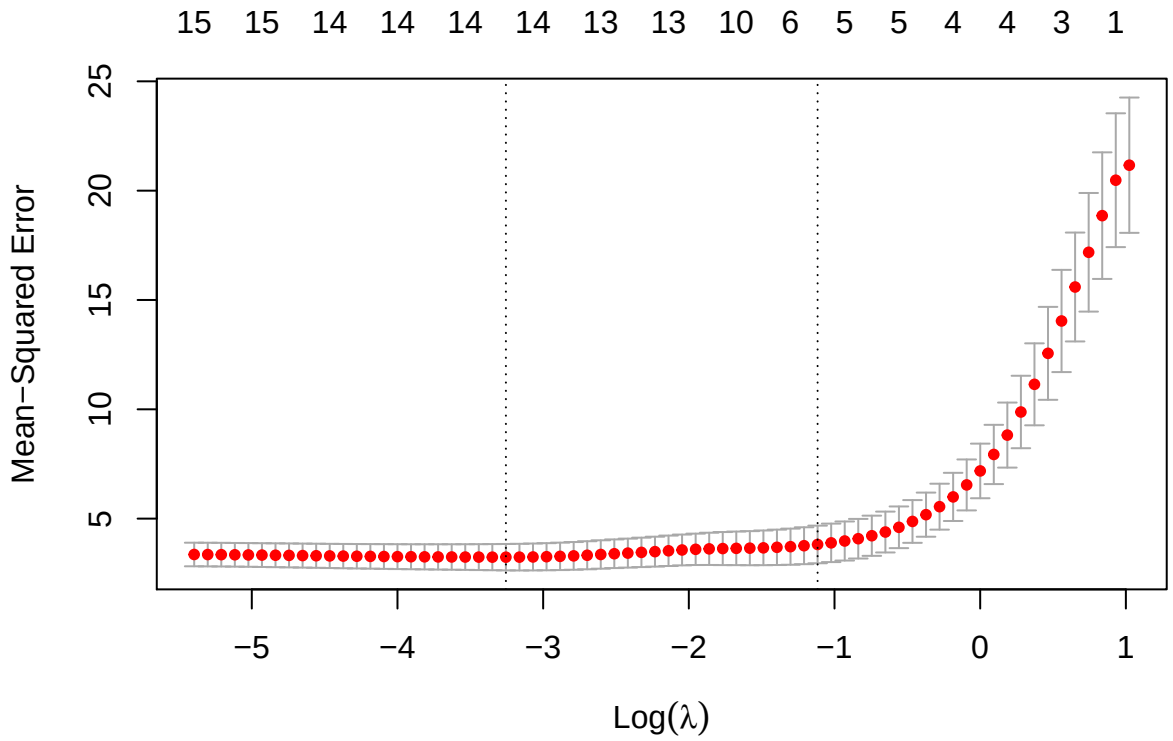
In this part of the exercise we are asked to find the optimal value of λ using Cross-Validation and decide the best value for the tuning parameter based on Mean Squared Error (MSE). Next, follows the part of the code that implements what we want. In cv.glmnet function the default parameter for the number of folds is 10.

```
lasso_cv = cv.glmnet(x, y)
cat("The lambda with min. CV-MSE: ",
      lasso_cv$lambda.min ,"\n",
    "the largest value of lambda", "\n",
    "so that the error is within 1se is :", lasso_cv$lambda.1se)
```

```
## The lambda with min. CV-MSE:  0.03855054
##  the largest value of lambda
##  so that the error is within 1se is : 0.3275845
```

In the output of the above code, we can observe the value for our tuning parameter λ with the minimum CV-MSE and also we can see the largest value of our parameter so that the error is about 1 standard error. In the plot below, we can see on the first vertical dotted line the minimum λ and on the second vertical dotted line we can see the λ $1se$. Also, we are able to observe the number of predictors variables that are included in our model from the top of the plot. We can see how the increase of the tuning parameter λ makes the model remove more variables. While when λ is smaller, all variables are added since we are approximating the OLS method as we go close to zero.

```
plot(lasso_cv)
```



Finally, we would like to see the coefficients of our model for the best λ, the one with the minimum CV-MSE for the unstandarised predictors.

```
blasso = coef(lasso_cv, s="lambda.min")
blasso
```

```
## 16 x 1 sparse Matrix of class "dgCMatrix"
##                        1
## (Intercept)  3.98278737
```

```
## V1           2.35948050
## V2           0.02335559
## V3           0.09256641
## V4           0.54065142
## V5          -0.68933289
## V6           0.21702069
## V7           2.18399398
## V8           .
## V9          -0.04513208
## V10          0.18507832
## V11          1.33756199
## V12          0.30624261
## V13          0.28467794
## V14          0.15367991
## V15         -0.44435162
```

Next we are asked to find the shrinkage factor s for the optimal λ. To calculate the shrinkage factor it is necessary to multiply the coefficients with the standard deviations of x by columns. This needs to be done in $LASSO$ model and OLS too. The code for this is shown below.

```
mfull = lm(y~ x)
zblasso = blasso[-1] * apply(x, 2, sd)
zbols =coef(mfull)[-1] * apply(x, 2 ,sd)
# caclulate s
s <- sum(abs(zblasso))/ sum(abs(zbols))
cat("The value of the shrinkage factor for the lasso model", "\n",
"with the minimum CV-MSE is", s)
```

```
## The value of the shrinkage factor for the lasso model
##  with the minimum CV-MSE is 0.8885217
```

After that we apply the same procedure but now for the $\lambda.1se$ which is as we said the maximum λ so that CV-MSE is withing 1 standard error.

```
blasso1 <- coef(lasso_cv, s="lambda.1se")
blasso1
```

```
## 16 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept) 3.98624955
## V1          2.21759856
## V2          0.02543473
## V3          0.28943061
## V4          0.63201251
## V5          .
## V6          .
## V7          2.02235533
## V8          .
## V9          .
## V10         .
## V11         1.07425356
## V12         .
## V13         .
```

```
## V14          .
## V15          .
```

```
zblasso1 = blasso1[-1] * apply(x, 2, sd)
zbols = coef(mfull)[-1] * apply(x, 2 ,sd)
s <- sum(abs(zblasso1))/ sum(abs(zbols))
cat("The value of the shrinkage factor for the lasso model", "\n",
"within 1se CV-MSE is", s)
```

```
## The value of the shrinkage factor for the lasso model
##  within 1se CV-MSE is 0.5487873
```

(iii)

In this part of the exercise we are called to implement a function to check which model fits better in our data, between the full model and the model that generated the response variables Y. We will use 5 fold cross-validation and the PRESS metric.

The full model is : $X = (X1, ...X10) \sim N(\mu, \Sigma)$ and $X_{ij} \sim N(0.2X_{i1} + 0.4X_{i2} + 0.6X_{i3} + 0.8X_{i4} + 1.1X_{i5}, 1)$, $j = 11, ..., 15$ $i = 1, .., 50$

And the model that generated our response variable is: $Y_i \sim N(4+2X_{i1}-X_{i5}+2.5X_{i7}+1.5X_{i11}+0.5X_{i13}, 1.5^2)$, $i = 1, ..50$

Also we will show the definition of the PRESS function. $PRESS = \sum_i^n (y_i - \hat{y}_{i,-i})^2$ This is the sum of squared difference between the response variable y_i and the variable $\hat{y}_{i,-i}$ corresponds to the estimation of $i-th$ observation when this observation is missing from the fitted model. Below, we can see the code that implements what we are asked. In our case since we are going to combine $PRESS$ and 5 fold cross-validation, we are going to predict on the validation set whose samples will be missing for the fitted model, which will use only the training set.

This part of code makes a shuffles the data. Shuffling is good when we are testing because it reduces variance and makes sure that models remain general and overfit less. An obvious case where shuffling is needed is when our data is sorted by a specific pattern like classes/targets. Then we would like to shuffle because we want or train/test sets to be representative of the overall distribution of the data.

```
#shuffling
shuff_ind = sample(nrow(x))
#shuffle rows of obs. x
X_shuff = x[shuff_ind, ]
#shuffle response y as x
Y_shuff = y[shuff_ind]
```

Now we are stating the function $PRESS$ that will help us calculate the metric we are intrested in.

```
#factors that gen. Y response
gen_factors = c(1 ,5 ,7, 11, 13)
#define  press function
press <- function (y, x, model){
      preds<- predict.lm(model, x)
      PRESS <- sum((preds - y)^2)
      return(PRESS)
}
```

Finally, we are providing the algorithm that computes $PRESS$ over a 5 fold CV. In this case we will have 5 groups of our data each consisted of 10 samples since we have 50 in total. This approach involves randomly dividing the set of observations into K(5 in the ex.) groups, of approximately equal size. The K-1 folds are used for training/fitting while the remaining is used for testing. And this happens K times.

```
full_mod_press = 0
gen_mod_press = 0

# iterate over 5 folds

for (i in seq(1, 50, by=10)){
    train_X = as.data.frame(X_shuff[-c(i:(i+9)), ]) # 40,15
    train_Y = Y_shuff[-c(i:(i+9))]                  # 40,15
    val_X = as.data.frame(X_shuff[c(i:(i+9)), ])    # 10,15
    val_Y = Y_shuff[c(i:(i+9))]                     # 10,15
    # fit full model
    full_mod = lm(train_Y~., data=train_X)
    # calculate its PRESS
    press_score = press(val_Y, val_X, full_mod)
    full_mod_press = full_mod_press + press_score
    # fit generated model
    gen_mod = lm(train_Y~., data=train_X[gen_factors])
    # calculate its PRESS
    press_score = press(val_Y, val_X[gen_factors], gen_mod)
    gen_mod_press = gen_mod_press + press_score
}
cat("The full model's PRESS is:",full_mod_press/5, "\n",
"The data generating model's PRESS is :", gen_mod_press/5)
```

```
## The full model's PRESS is: 46.00202
##  The data generating model's PRESS is : 32.0582
```

Obviously, the model that generated the response variable is much more accurate as it has a lower $PRESS$ value. That is reasonable, since the full model has additional covariates which are not needed as we saw when we implemented $LASSO$ method or BIC exhaustive search.

(iv)

In the last part of the exercise, we are asked to give estimators and standard errors for the coefficients of the model that generated our response variable. Before moving forward to the code, will describe the method of bootstrap regression using residuals.

1. We estimate the regression coefficients $b_0, b_1, b_5, b_7, b_{11}, b_{13}$ for the original sample, and calculate the fitted value and residual for each observation. $e_i = y_i - \hat{y_i}$.
2. Then we select bootstrap samples of the residuals. $e_b^* = [e_{b1}, e_{b2}, .., e_{b50}]$. We choose randomly 50 samples with replacement. And from these residuals we calculate the following $Y_b^* = [Y_{b1}^*, Y_{b2}^*, ..., Y_{b50}^*]$ where $Y_{bi}^* = \hat{y_i} + e_{bi}^*$.
3. After that we regress the bootstrapped Y values which we note as Y_b^* on the fixed x values to obtain the bootstrap coefficients.

The above method is repeated 1000 times. The resampled coefficients can be used to construct bootstrap standard errors.

This part of the code fits the model that generated the response variable Y and computes the residuals.

```
library(bootstrap)
X_gen <- as.data.frame(x)[gen_factors]
gen_model <- lm(y~.,data=X_gen)
residuals <- gen_model$residuals
```

Here we are creating a function to use it on the bootstrap function as a parameter. Function's input are residuals, model and the data with the variables that generated our model. On the inside of the function, we are creating the new Y values as $Y^*_{bi} = \hat{y}_i + e^*_{bi}$.

```
boot_coef <- function(residuals,model,xgen) {

  Y_syn <- model$fitted.values + residuals
  coef(lm(Y_syn~., data=xgen))
}
```

Here we call the function bootstrap from the bootstrap package. It is performing bootstrapping with replacement sampling from the residuals 1000 times and calculates the coefficients every time. Finally, it computes the estimated coefficients and the corresponding standard error.

```
bs_coefficients <- bootstrap::bootstrap(gen_model$residuals, 1000,
            function(x) boot_coef(x, gen_model, X_gen))
```

Below, the bootstrap coefficients and standard errors are computed. We do that, by averaging the value *thetastar* from the bootstrap package and for the standard errors we apply the standard deviation function over *thetastar*. By the documentation we can see that *thetastar* consists of the 1000 bootstrap values of our model coefficients.

```
# calculate coefficient estimates
coef_est <- apply(bs_coefficients$thetastar, 1, mean)
# caclulate standard errors
coef_sd <- apply(bs_coefficients$thetastar, 1, sd)
```

Comparison of the OLS method and Bootstrap method

Below, we can see the results of the coefficients from the bootstrap method and the OLS one.

```
cat("Bootstrap est. of coefficients: "
    ,coef_est)
```

```
## Bootstrap est. of coefficients:  4.04515 2.375511 -1.238542 2.304385 1.61207 0.4471817
```

```
cat("Data gen. coefficients(w/o bs): ",gen_model$coefficients)
```

```
## Data gen. coefficients(w/o bs):  4.041597 2.363219 -1.243954 2.310465 1.615967 0.4473327
```

In this part of the R code we can see the difference between the standard errors of the 2 models we are comparing.

```r
cat("Bootstrap est. of stand. error: "
    ,coef_sd)
```

```
## Bootstrap est. of stand. error:  0.2287749 0.2762713 0.2847219 0.2838575 0.1622844 0.169963
```

```r
cat("standard error of gen. model: ",
    coef(summary(gen_model))[, "Std. Error"])
```

```
## standard error of gen. model:  0.2378074 0.2804208 0.3056673 0.3040627 0.1792967 0.1852704
```

Conclusion

From the results we saw before, we cannot see any big difference on the coefficients, which is reasonable since the model that generated our data is very accurate to our new synthetic data ($y + error_{boot}$).

However in the standard error terms, we have some really noticeable differences. We can observe that the values in the bootstrap method are lower than of the generated model. That is expected, because the bootstrap method fitted 1000 models and justifies why it can estimate the coefficients more precisely.