Bayesian Networks

Javier Aguirre

22/12/2021

```r
library(bnlearn)
library(Rgraphviz)
library(ggplot2)
library(igraph)
library(gRain)
```

Exercise 1

During the last year and a half we have get used to be tested for COVID-19. Whenever we have compatible symptoms $(S \in y, n)$, we are requested to make a PCR to identify the infection. The result of this test can be positive or negative $(T \in p, n)$. The underlying relevant information, not directly known, is whether we have been infected with SARS-CoV-2 or not $(I \in y, n)$.

Reviewing the data available in Osakidetza for a given (quite bad) week, we know that the number of PCR tests taken in the population (of roughly 2 million inhabitants) is 40000, and we know that half those tests have been taken due to symptoms. Also, we know that, for those with symptoms, 2 out of 10 tests are positive, but for those without symptoms the ratio is only 1 out of 20.

According to the lab producing the tests, on the one hand, 85 of the tested patients that have a positive result are actually infected and, on the other hand, 99 of the tested patients with a negative test are not infected. Assuming that given the result of the test having the disease is independent of having compatible symptoms, answer these questions.

1. What is the probability of not having the disease, have symptoms and a positive PCR?
2. What is the probability of having the disease given that we have symptoms?

Solution

Information extracted from the problem:

$P(T = p|S = y) = 0.2$

$P(T = p|S = n) = 0.05$

$P(I = y|T = p) = 0.85$

$P(I = n|T = n) = 0.99$

$P(I = n|T = p) = 0.15$

$P(S = y) = 0.01$

$P(S = n) = 0.99$

Part 1

What is the probability of not having the disease, have symptoms and a positive PCR?

The probability of not having a disease, have symptoms and a positive PCR can be expressed as:

$P(I = n, T = p, S = y)$

in order to solve it we can apply the chain rule:

$P(I = n, T = p, S = y) = P(S = y) * P(T = p|S = y) * P(I = n|S = y, T = p) = 0.01 * 0.2 * 0.15 = 0.0003$

Notice that in this case $P(I = n|S = y, T = p)$ is equal to $P(I = n|T = p)$ which the problem states is equal to 0.15

Part 2

What is the probability of having the disease given that we have symptoms?

The probability of having the disease given that we have symptoms can be expressed as:

$P(I = y|S = y)$

In order to solve it we could perform the marginalization calculating in both cases $I = y$ and $S = y$ and calculating $T = n$ and $T = p$. This way just by adding them to get the probability of $P(I = y, S = y)$.

1. $P(I = y, T = n, S = y) = P(S = y) * P(T = n|S = y) * P(I = y|S = y, T = n) = 0.01 * 0.8 * 0.01 = 0.00008$
2. $P(I = y, T = p, S = y) = P(S = y) * P(T = p|S = y) * P(I = y|S = y, T = p) = 0.01 * 0.2 * 0.85 = 0.0017$

Marginalization:

$$P(I = y, S = y) = P(I = y, T = n, S = y) + P(I = y, T = p, S = y) = 0.00008 + 0.0017 = 0.00178$$

Finally in order to get $P(I = y|S = y)$ we can apply the bayes theorem:

$$P(A|B) = P\frac{A \cap B}{A}$$

Applied to our problem:

$$P(I = y|S = y) = \frac{P(I=y,S=y)}{P(S=y)} = 0.00178/0.01 = 0.178$$

Exercise 2

We are trying to model the time to commute to the faculty (T) basing the model in a simple Beta distribution. As we have determined that the minimum time required is 30 minutes and, at most, the travel time will be 240 minutes, we have proposed to model the time as $T = 30 + 210X$ where $X \sim \text{Beta}(\alpha, \beta)$.

1. Write the equation of the density function for the travel time, $f_T(t)$
2. Calculate $E[T]$ and $VAR[T]$
3. Get the estimators for α and β using the moments method

Solution

Part 1

Write the equation of the density function for the travel time, $f_T(t)$.

First we need the density function of X defined as follows in the literature:

$$f(X = x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\text{Beta}(\alpha,\beta)}$$

Taking into account that we are working with time and not just with a density function of X we have to remember as in the previous exercise that $T = 30 + 210X$ therefore:

$$g^{-1}(T = t) = X = \frac{t-30}{210}$$

Finally the substitution must be done as follows in order to obtain the density function:

$$f_T(t) = f(g^{-1}(t))|\frac{d}{dt}g^{-1}(t)| = \frac{1}{\text{Beta}(\alpha,\beta)}(\frac{t-30}{210})^{\alpha-1}(1 - \frac{t-30}{210})^{\beta-1}\frac{1}{210}$$

Part 2

Calculate $E[T]$ and $VAR[T]$.

Knowing that in a beta distribution the mean of the aleatory variable X is $E[X] = \frac{\alpha}{\alpha+\beta}$:

$$E[T] = 30 + 210E[X] = 30 + 210\frac{\alpha}{\alpha+\beta}$$

Knowing that in a beta distribution the variance of the aleatory variable X is $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$:

$$var[T] = var(30) + 210^2 var[X] = 210^2\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$$

Part 3

Get the estimators for α and β using the moments method, let us start with the mean:

$$E[\hat{T}] = \overline{T}$$

$$\overline{T} = 30 + 210\frac{\hat{\alpha}}{\hat{\alpha} + \hat{\beta}}$$

$$\hat{\alpha}\overline{T} + \hat{\beta}\overline{T} = 30\hat{\alpha} + 30\hat{\beta} + 210\hat{\alpha}$$

$$-30\hat{\beta} + \hat{\beta}\overline{T} = 240\hat{\alpha} - \hat{\alpha}\overline{T}$$

$$\hat{\beta}(\overline{T} - 30) = 240\hat{\alpha} - \hat{\alpha}\overline{T}$$

$$\hat{\beta} = \frac{\hat{\alpha}(240 - \overline{T})}{\overline{T} - 30}$$

Now we will resolve alpha:

$$var[\hat{T}] = S^2 = 210^2\frac{\hat{\alpha}\frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30}}{(\hat{\alpha} + \frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30})(\hat{\alpha} + \frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30} + 1)}$$

$$S^2 = 210^2\frac{\hat{\alpha}\frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30}}{(\hat{\alpha} + \frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30})(\hat{\alpha} + \frac{\hat{\alpha}(240-\overline{T})}{\overline{T}-30} + 1)} * \frac{(\overline{T} - 30)^3}{(\overline{T} - 30)^3}$$

$$S^2 = 210^2\frac{\hat{\alpha}^2(240 - \overline{T})(\overline{T} - 30)^2}{(\hat{\alpha}(\overline{T} - 30) + \hat{\alpha}(240 - \overline{T})^2((\hat{\alpha} + 1)(\overline{T} - 30) + \hat{\alpha}(240 - \overline{T}))}$$

$$S^2 = \frac{(240 - \overline{T})(\overline{T} - 30)^2}{210\hat{\alpha} + \overline{T} - 30}$$

$$S^2(210\hat{\alpha} - 30 - \overline{T}) = (240 - \overline{T})(\overline{T} - 30)^2$$

$$\hat{\alpha} = \frac{(240 - \overline{T})(\overline{T} - 30)^2 + 30S^2 - \overline{T}S^2}{210S^2}$$

Having both moments we isolated alpha and beta which are the parameters of the beta distribution.

Exercise 3

The Pareto distribution has this density function: $f(x; x0, \alpha) = \alpha x_0^{\alpha} x^{-(\alpha+1)}$, where x_0 and α are the parameters and X is defined in $\Omega_X = [x_0, \infty]$. The location parameter, x_0, is usually estimated as sample minimum $(\hat{x}_0 = \min(x_1, \ldots, x_n))$ while the maximum likelihood estimator for the α parameter is $\hat{\alpha} = n/\sum_{i=1}^{n} \ln(x_i/\hat{x}_o)$ .Given the sample:

```
pareto.sample <- c(2.53, 39.20, 2.53, 11.83, 6.73, 2.85, 2.27, 2.41, 2.94, 9.10)
```

Use the bootstrap approach to estimate the bias and the standard error of both estimators.

Solution

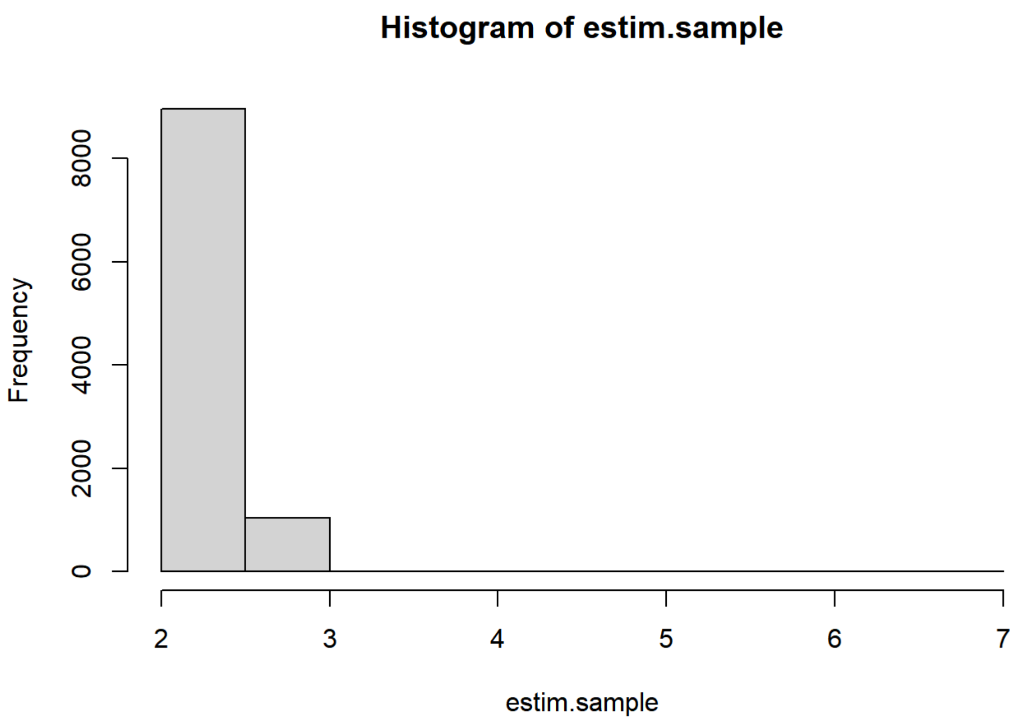
Starting from the pareto sample, boostrap samples are created and from each of them the minimum is calculated as the first estimator, therefore, obtaining a new sample with the minimum:

```
pareto.sample <- c(2.53, 39.20, 2.53, 11.83, 6.73, 2.85, 2.27, 2.41, 2.94, 9.10)

n.boot.samples <- 10000
set.seed(100)
estim.sample <- sapply(1:n.boot.samples,
                       FUN=function(i) {
```

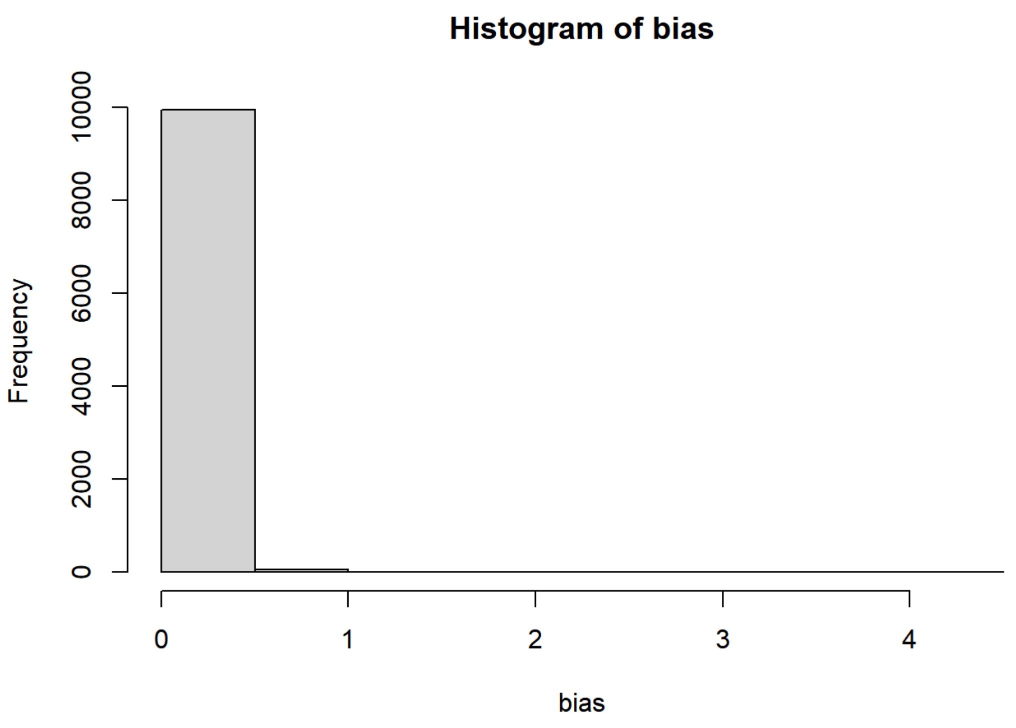
```
                                  return(min(sample(pareto.sample, size=length(pareto.sample), replace=TRUE)))
                                })
```

With the previous information the results are plotted, the standard error and mean are calculated:

```
hist(estim.sample)
```



```
bias<-estim.sample-min(pareto.sample)
hist(bias)
```



```
print("bias: ")
```

```
## [1] "bias: "
```

```
print(mean(bias))
```

```
## [1] 0.064317
```

```
standardError<-sd(estim.sample)/sqrt(n.boot.samples)

print("standardError: ")
```

```
## [1] "standardError: "
```

```
print(standardError)
```

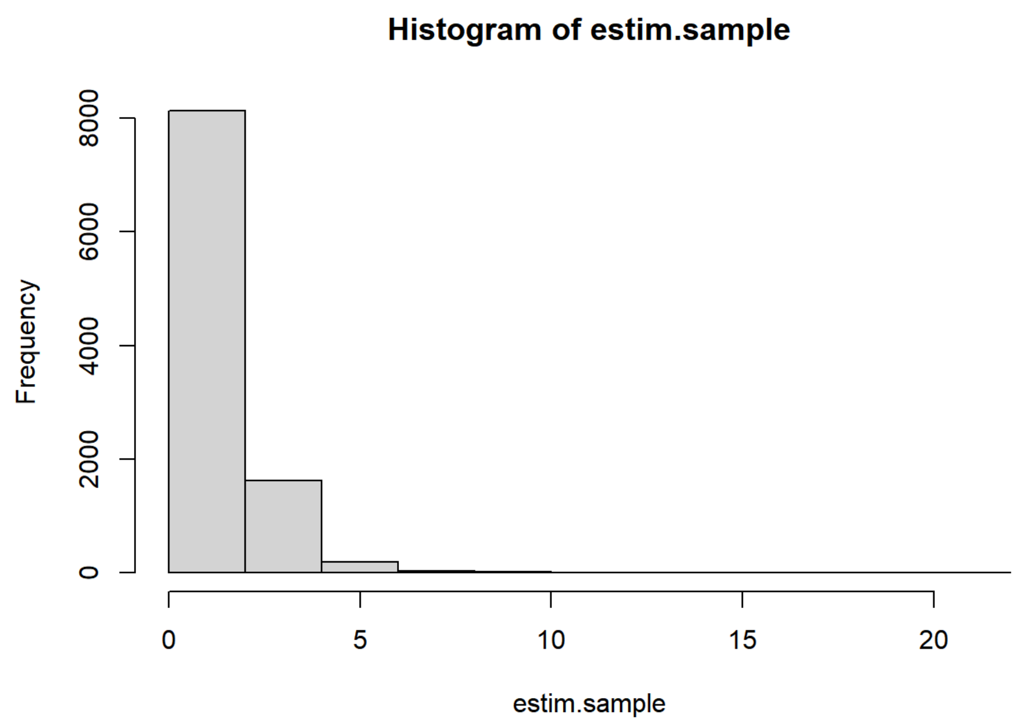
```
## [1] 0.001163993
```

In the case of the maximum likelyhood estimator we need to create a separate function (the coded representation of the function):

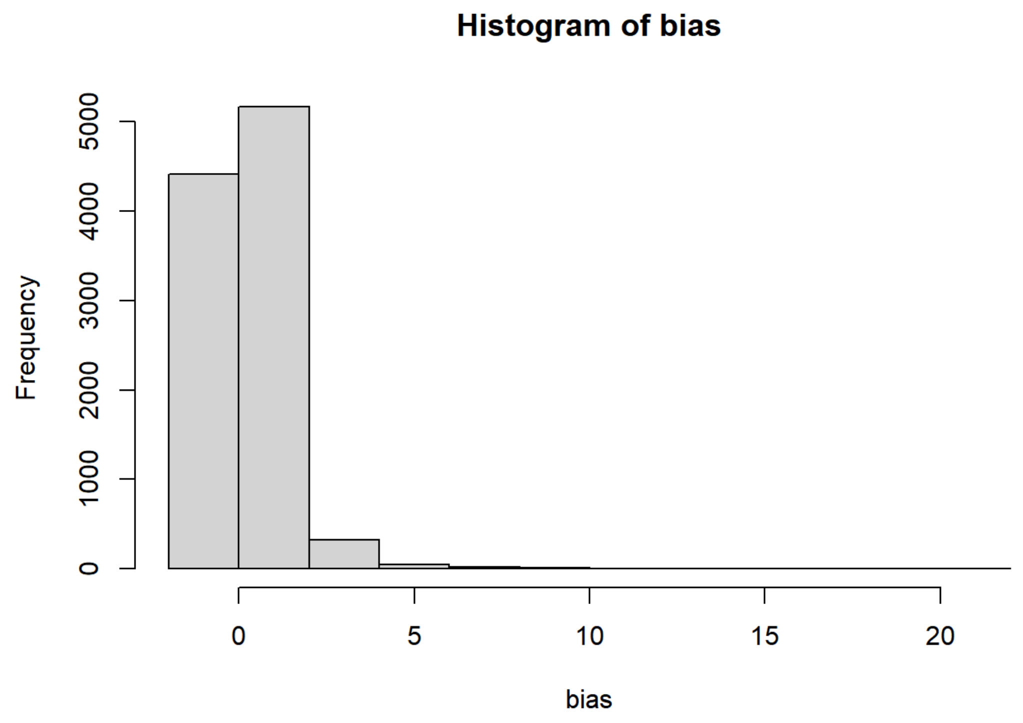
```
#MAXIMUM LIKELYHOOD
library("SciViews")
maximumLikelyhood <- function(sample) {
  n<-length(sample)
  x0<-min(sample)
  Summatory<-0
  for (xi in sample) {
    Summatory<-Summatory+ln(xi/x0)
  }
  maxim<-n/Summatory
  return(maxim)
}
```

The same process as the minimum estimator is followed but in this case with the maximum likelyhood estimator:

```
estim.sample <- sapply(1:n.boot.samples,
                       FUN=function(i) {
                         return(maximumLikelyhood(sample(pareto.sample, size=length(pareto.sample), re
place=TRUE)))
                       })
hist(estim.sample)
```



```
bias<-estim.sample-maximumLikelyhood(pareto.sample)
hist(bias)
```



```
print("bias: ")
```

```
## [1] "bias: "
```

```
print(mean(bias))
```

```
## [1] 0.3156019
```

```
standardError<-sd(estim.sample)/sqrt(n.boot.samples)

print("standardError: ")
```

```
## [1] "standardError: "
```

```
print(standardError)
```

```
## [1] 0.01001771
```

Exercise 4

In Exercise 2 we have obtained estimators for α and β from a sample of $T = 30 + 210X$ where $X \sim \mathrm{Beta}(\alpha, \beta)$. In this exercise you have to analyze the bias, variance and the standard error of these two estimators as the sample size increases. All these values are parameter-dependent, so you will have to do the analysis with only two sets of parameters, $(\alpha = 2.27, \beta = 5.32)$ and $(\alpha = 0.27, \beta = 0.32)$.

In both cases you will have to create plots for the bias, the standard deviation and the standard error of both estimators for increasing sample sizes. In particular, you should use these values $n \in 10, 25, 50, 100, 500, 1000, 10000$

Note that the estimators are random variable and, as such, each time a sample is used to produce an estimation, the result will be different. Therefore, you should design a proper way to evaluate the estimators and a suitable graphical representation to show the evolution of these properties.

Solution

The aim is to see how the bias, variance and standard error differ when the sample sizes get bigger, therefore, two for loops have been created. The first one goes running each sample size: 10, 25, 50… the second one performs the experiment for each of them. For that the *amountOfSamples* variable will be set to 2000 which will be the amount of samples to create. In that inner loop thanks to the moments method the approximation to an alpha and beta are achieved.

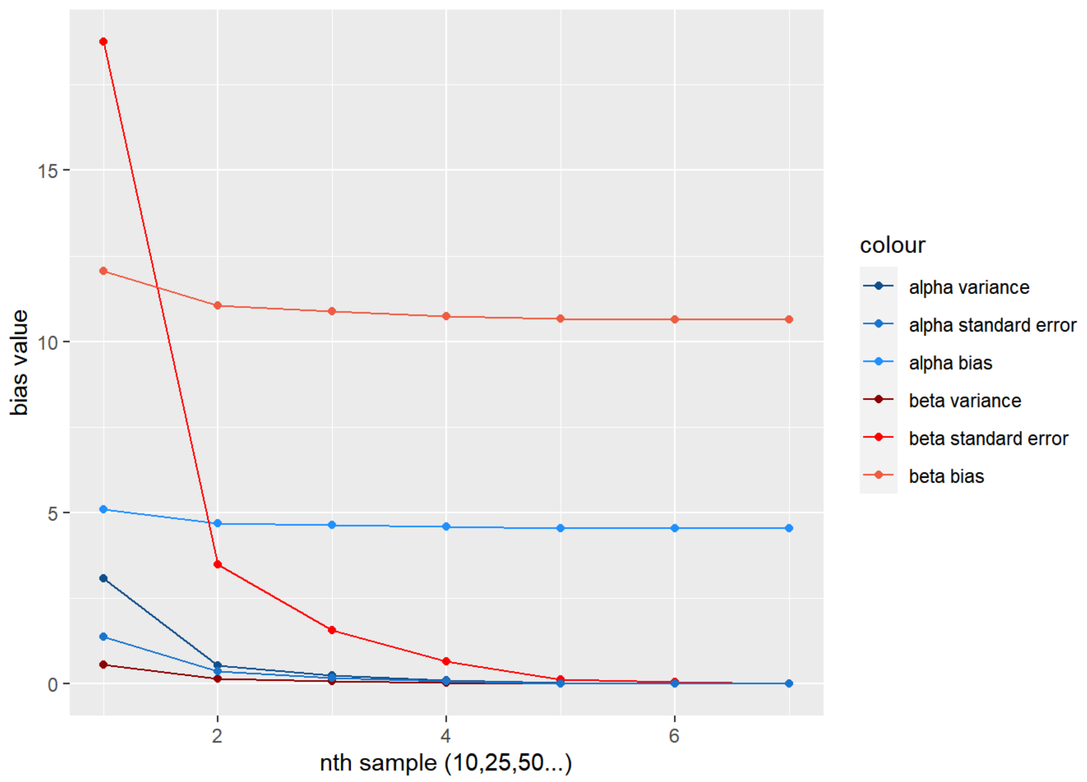
Outside of the loops the mean, variance and standard error are calculated as usual using the provisional list of alphas and betas achieved in the inner loops. Finally in the last vectors the result obtained is one per sample size.

```
alpha<-2.27
beta<-5.32

amountOfSamples<-2000
sampleSize<-c(10, 25, 50, 100, 500, 1000, 10000)

alphaVarianceList<-c()
betaVarianceList<-c()

alphaStandarErrorList<-c()
betaStandardErrorList<-c()

alphaBiasList<-c()
betaBiasist<-c()

#Run for each sample size
for (currentSampleSize in sampleSize){
  provisionalAlphaList<-c()
  provisionalBetaList<-c()

  #Creation of n samples, in this case 2000 and storing alphas and betas values on provisional array
  for (i in 1:amountOfSamples){
    sample<-30+210*rbeta(currentSampleSize,alpha,beta)
    variance<-sum((sample-mean(sample))^2)/(currentSampleSize-1)
    momentAlpha<-((240-mean(sample))*(mean(sample)-30)^2+30*variance-variance*mean(sample))/(-210*vari
ance)

    provisionalAlphaList[length(provisionalAlphaList)+1]<-momentAlpha
    provisionalBetaList[length(provisionalBetaList)+1]<-(momentAlpha*(240-mean(sample))/(mean(sample)-
30))
```

```
  }

  #calculation from provisional list the current n mean, variance, standard error and bias
  meanAlpha<-mean(provisionalAlphaList)
  VARAlpha<-var(provisionalAlphaList)
  SEAlpha<-sd(provisionalAlphaList)/sqrt(currentSampleSize)
  BIASAlpha<-abs(meanAlpha-alpha)

  #calculation from provisional list the current n mean, variance, standard error and bias
  meanBeta<-mean(provisionalBetaList)
  VARBeta<-var(provisionalBetaList)
  SEBeta<-sd(provisionalBetaList)/sqrt(currentSampleSize)
  BIASBeta<-abs(meanBeta-beta)

  #adding the previous values to arrays that will represent the sample sizes values (the alpha for 10,
25, 50 ... 10000)
  alphaVarianceList[length(alphaVarianceList)+1]<-VARAlpha
  alphaStandarErrorList[length(alphaStandarErrorList)+1]<-SEAlpha
  alphaBiasList[length(alphaBiasList)+1]<-BIASAlpha

  betaVarianceList[length(betaVarianceList)+1]<-VARBeta
  betaStandardErrorList[length(betaStandardErrorList)+1]<-SEBeta
  betaBiasist[length(betaBiasist)+1]<-BIASBeta
}
```

Now the data is stored in a dataframe and the results are plotted:

```
sampleSize<-c(1,2,3,4,5,6,7)
df<-data.frame(sampleSize,alphaVarianceList,alphaStandarErrorList,alphaBiasList,betaVarianceList,betaS
tandardErrorList,betaBiasist)
library(ggplot2)

p<-ggplot() +
  geom_line(data=df, mapping=aes(x = sampleSize, y = alphaVarianceList,color = "blue"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = alphaVarianceList,color = "blue"))+
  xlab("nth sample (10,25,50...)") +
  ylab("variance value")+

  geom_line(data=df, mapping=aes(x = sampleSize, y = alphaStandarErrorList, color = "lightblue"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = alphaStandarErrorList, color = "lightblue"))+
  xlab("nth sample (10,25,50...)")+
  ylab("standard error value")+

  geom_line(data=df, mapping=aes(x = sampleSize, y = alphaBiasList, color = "darkblue"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = alphaBiasList, color = "darkblue"))+
  xlab("nth sample (10,25,50...)")+
  ylab("bias value")+

  geom_line(data=df, mapping=aes(x = sampleSize, y = betaVarianceList, color = "red"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = betaVarianceList, color = "red"))+
  xlab("nth sample (10,25,50...)")+
  ylab("variance value")+

  geom_line(data=df, mapping=aes(x = sampleSize, y = betaStandardErrorList, color = "coral2"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = betaStandardErrorList, color = "coral2"))+
  xlab("nth sample (10,25,50...)")+
  ylab("standard error value")+

  geom_line(data=df, mapping=aes(x = sampleSize, y = betaBiasist, color = "red4"))+
  geom_point(data=df, mapping=aes(x = sampleSize, y = betaBiasist, color = "red4"))+
  xlab("nth sample (10,25,50...)")+
  ylab("bias value")

p + scale_color_manual(labels = c("alpha variance", "alpha standard error", "alpha bias","beta varianc
e", "beta standard error", "beta bias"),values=c("dodgerblue4","dodgerblue3","dodgerblue1","red4","red
1","tomato2"))
```



We can see as a conclusion that the results get smaller when the instance size increases. This is what it was expected since with more samples a more precise approximation to alpha and beta are achieved.

Exercise 5

Parametric modeling

During the last year we have collected information about commute times (T) together with the weather conditions (W). In the following two exercises you will have to model this data following to approaches, a parametric one and a non-parametric one.

The implemented functions have to be tested using the dataset sample, loaded with:

```
load("RRBB_travel_duration.Rdata")
```

In this exercise you have to implement two different functions, one to build a parametric model for the data and the other one to estimate probabilities from a given model. The definition for the first function has to be:

```
buildParametricModel <- function (data) {
# Build and return a parametric model
}
```

The data parameter will be a data.frame similar to the example data. That is, a data.frame with two columns, one named Weather, of type factor, and the other one named Time, of type numeric. The model has to assume that the two modeled variables (Weather and Time) are not independent. The model in this exercise has to be a parametric one, that is, it has to be based on known distributions. The format of the output model is free, but it has to work correctly with the function used to estimate probabilties.

The definition for this last function is:

```
estimateProbabilityParametric <- function (model, a, b, weather=NA){
  probability <- 0
  return(probability)
}
```

The parameters for this function are:

1. model Whatever type of element returned by buildParametricModel
2. a Lower time limit
3. b Upper time limit
4. weather Either a valid value (S or R) or NA

The function will return the probability of the trip taking a time between a and b, conditioned or not to the weather (depending on whether weather has a valid value or is NA). That is, if weather is NA, the function will return $P(a < T < b)$, while if weather takes a valid value w the function will return $P(a < T < b|W = w)$.

Solution

First of all, the data must be loaded and it will be separated by time being rainy, sunny or both:

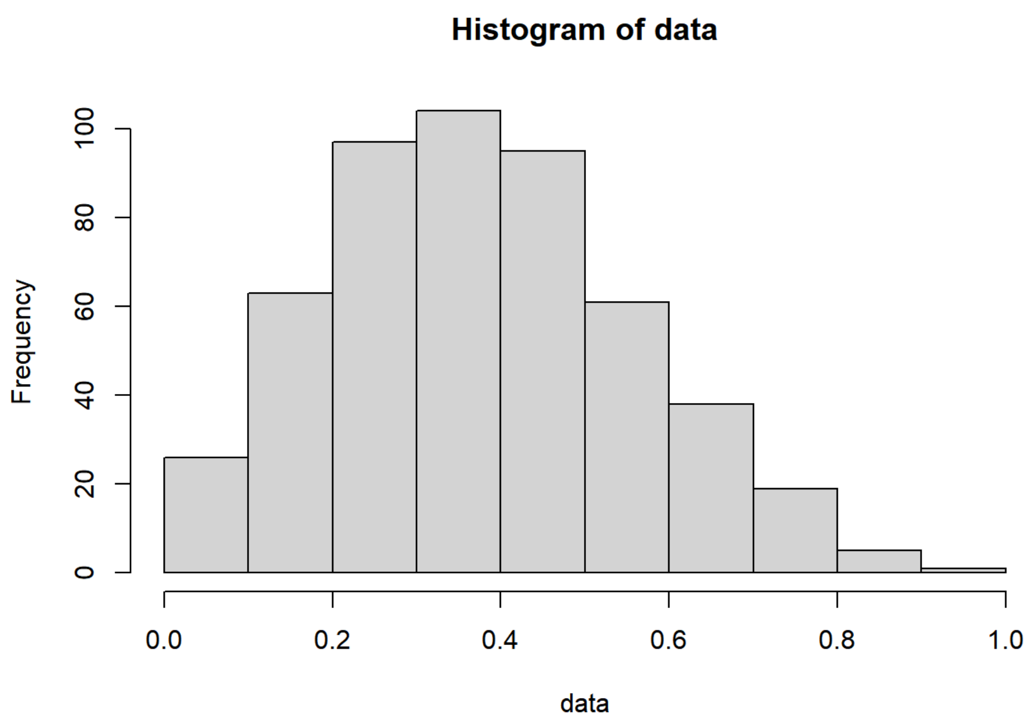
```r
library("fitODBOD")
library("EnvStats")
x<-load("RRBB_travel_duration.Rdata")
matrix<-get(x)
timeRainy<-matrix[matrix$Weather=="R",]$Time
timeSunny<-matrix[matrix$Weather=="S",]$Time
time<-matrix$Time
```

Now all the data must be normalized in order to construct the model

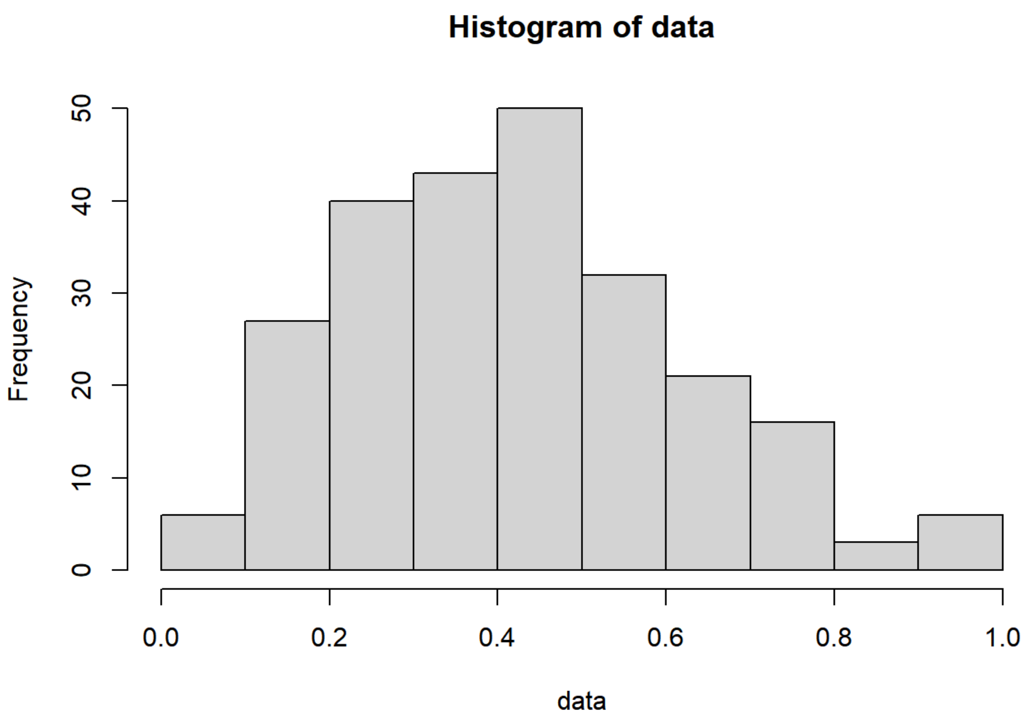
```r
normalize <- function(x) {
  return ((x - min(x)) / (max(x) - min(x)))
}

timeRainy<-normalize(timeRainy)
timeSunny<-normalize(timeSunny)
time<-normalize(time)
```

We build the parametric model, for that, we apply the ebeta function in order to get the alpha and beta using the moments method. Then with dbeta we get the density with the data and the alpha and beta. Finally approxfun is applied in order to perform an interpolation of the given data points.

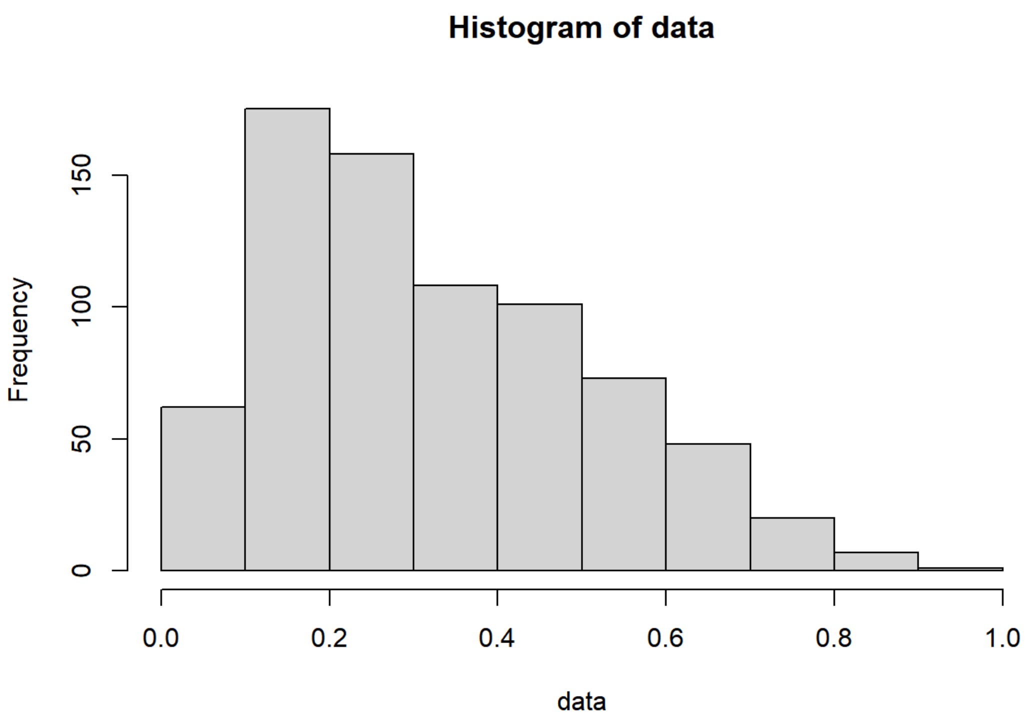
```r
buildParametricModel <- function (data) {
  # Build and return a parametric model
  hist(data)
  value<-ebeta(data, method = "mme")
  result<-dbeta(data,value$parameters[1],value$parameters[2])
  return(approxfun(result, rule=2))
}

model1<-buildParametricModel(timeRainy)
```



```
model2<-buildParametricModel(timeSunny)
```



```
model3<-buildParametricModel(time)
```



```
model<-list(model1, model2, model3)
```

Finally in order to estimate the probability, by integrating the model created above between certain points the result is obtained.

```
estimateProbabilityParametric <- function (model, a, b, weather="NA"){
  a2=(a-30)/210
  b2=(b-30)/210
  if(weather=="NA"){
    result<-integrate(model[[1]], a2, b2)
  }
  else if(weather=="R"){
    result<-integrate(model[[2]], a2, b2)
  }
  else{
    result<-integrate(model[[3]], a2, b2)
  }

  return(result)
}

estimateProbabilityParametric(model,40,50,"R")
```

```
## 0.08423019 with absolute error < 9.4e-16
```

```
estimateProbabilityParametric(model,40,50,"S")
```

```
## 0.05978384 with absolute error < 6.6e-16
```

```
estimateProbabilityParametric(model,40,50,"NA")
```

```
## 0.08174928 with absolute error < 9.1e-16
```

Exercise 6

In this exercise you have to do the same modeling as in the previous one, but using a non-parametric approach. The function definitions to be used are:

```
buildNonParametricModel <- function (data) {
# Build and return a parametric model
}
estimateNonProbabilityParametric <- function (model, a, b, weather=NA){
probability <- 0
return(probability)
}
```

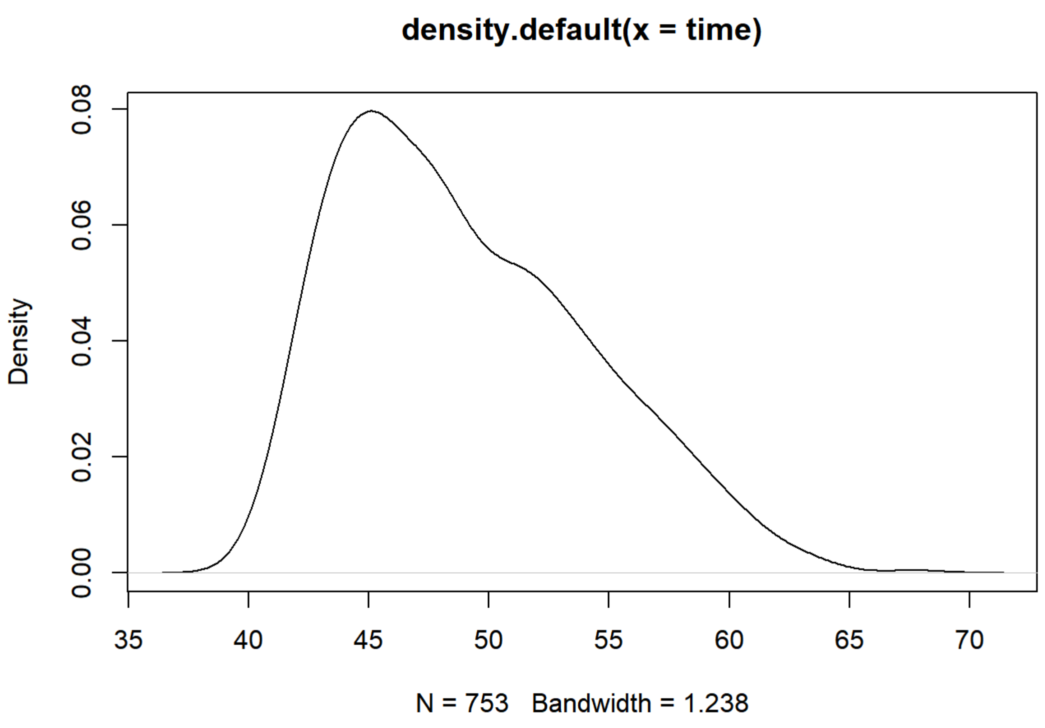
Solution

First of all we load the data and we separate it in general data *time* on data when it is rainy *timeRainy* and when it is sunny *timeSunny*.

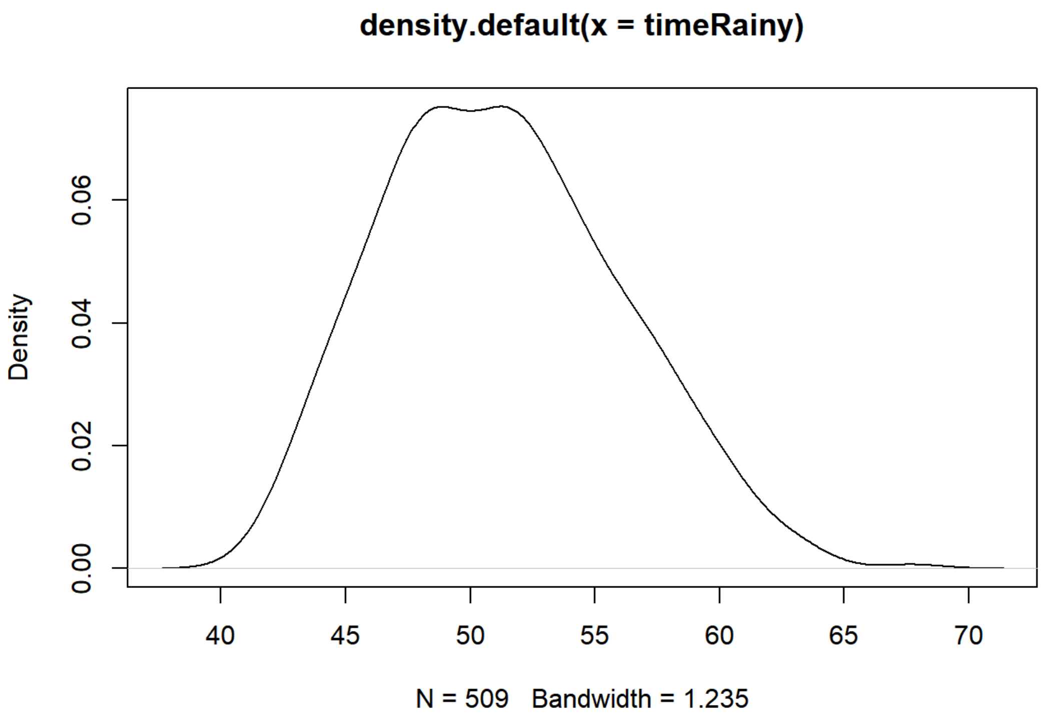
```
x<-load("RRBB_travel_duration.Rdata")
matrix<-get(x)
time<-matrix$Time
timeRainy<-matrix[matrix$Weather=="R",]$Time
timeSunny<-matrix[matrix$Weather=="S",]$Time
```

We plot the data, just to see it (it is not necessary) and we add it to a list called data.

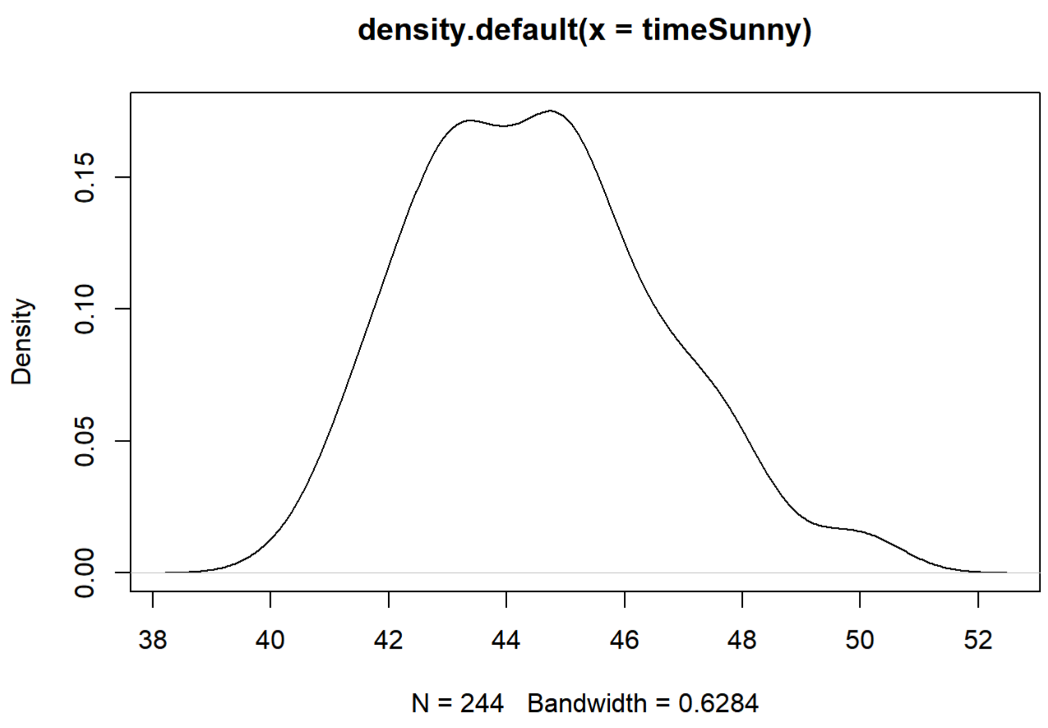
```
plot(density(time))
```



```
plot(density(timeRainy))
```



```
plot(density(timeSunny))
```



```
data<-list(time, timeRainy, timeSunny)
```

We create the function *buildNonParametricModel* in order to model our data, for that the function *approxfun* is used which returns a function performing an interpolation of the given data points. For a given set of x values, this function will return the corresponding interpolated values. And the list of all the three distributions is returned: Rainy, Sunny, Overall time.

```
buildNonParametricModel <- function (data) {
  one<-approxfun(density(data[[1]]), rule=2)
  two<-approxfun(density(data[[2]]), rule=2)
  three<-approxfun(density(data[[3]]), rule=2)
  lista<-list(one,two,three)
  return(lista)
}
```

In order to get the desired result and be able to estimate the probability between two given points the previous built model must be integrated.

```
estimateNonProbabilityParametric <- function (model, a, b, weather="NA"){

  if(weather=="NA"){
    result<-integrate(model[[1]], a, b)
  }
  else if(weather=="R"){
    result<-integrate(model[[2]], a, b)
  }
  else{
    result<-integrate(model[[3]], a, b)
  }

  return(result)
}
```

We build the model:

```
model<-buildNonParametricModel(data)
```

We calculate the probability of the value to be between 30 and 40 for the rainy weather:

```
estimateNonProbabilityParametric(model,30,40,"R")
```

```
## 0.001294331 with absolute error < 6.2e-06
```

We calculate the probability of the value to be between 30 and 40 for the sunny weather:

```
estimateNonProbabilityParametric(model,30,40,"S")
```

```
## 0.006039479 with absolute error < 1.6e-05
```

We calculate the probability of the value to be between 30 and 40 for the overall weather:

```
estimateNonProbabilityParametric(model,30,40,"NA")
```

```
## 0.00759484 with absolute error < 1.9e-05
```

Exercise 7

Bayesian Networks

The two exercises concerning Bayesian networks consist in implementing an R function. In both cases, the pdf document should include the details of the implementation and also one example of use.

In problems where there are many variables, slight modifications of the dataset can lead to very different networks. Beyond its use in the evaluation of estimators, bootstrap methods offer one way to increase the robustness of the results, when coupled with model averaging.

In this context, bootstrap resampling consists in, given a dataset with n variables and N instances, creating new datasets of the same size (n variables and N instances) by sampling the instances with replacement. As a result, for each boostrap resampling we will have a dataset where some of the original instances will appear once or more than once while some others won't appear at all.

If we use each of these newly created bootstrap datasets to learn a Bayesian network structure, as a result we will have different network structures. Then, we can combine those structures to create a new graph where only those edges that appear repeatedly are included. Note that, even if each of the networks are DAGs, the consensus network is not necessarily a DAG.

In this exercise you have to create an R function which, given a data matrix and a number of resamples (and, probably, some other parameters), creates a consensus network. The result can be of any type, but there should be the possibility to draw the result.

This is the structure for the function. Note that, as said, you can add other parameters to the function.

```
getConsensusNetwork <- function (dataset, n.bootstrap, ...) {
# dataset should be a matrix or a dataframe
# n.boostrap refers to the number of boostrap resamples and, thus,
# the number of networks averaged
}
```

Solution

First of all we have to select a dataset, the kaggle spotify dataset:

```
data<-read.csv("spotifyData.csv")
```

For the sake of simplicity and of computational cost we will just work with 5 columns which are probability based:

1. **danceability**: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable.
2. **energy**: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy. For example, death metal has high energy, while a Bach prelude scores low on the scale. Perceptual features contributing to this attribute include dynamic range, perceived loudness, timbre, onset rate, and general entropy.
3. **livelyness**: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.

4. **speechiness**: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.
5. **valence**: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

Therefore the rest of the columns will be erased:

```
data<-data[,-1]
data<-data[,-1]
data<-data[,-2]
data<-data[,-3]
data<-data[,-3]
data<-data[,-4]
data<-data[,-4]
data<-data[,-5]
data<-data[,-5]
data<-data[,-6]
data<-data[,-6]
data<-data[,-6]
```

Let us also define the *getConsensusNetwork* function. We initialize the *finalMatrix* by calculating the adjacency matrix of the bayesian network of the original data.

Given a data and a number of samples to create first an array of *2017* numbers is created in order. This array will be sampled to get a new array with replacements and based on that array the data of spotify will be resampled this way mantaining each rows attributes but some rows will disappear and others will get repeated.

Each time a sample is created a hill climbing is applied and the bayesian network is created. This results are then passed to an adjacency matrix format and added with the previous results each iteration.

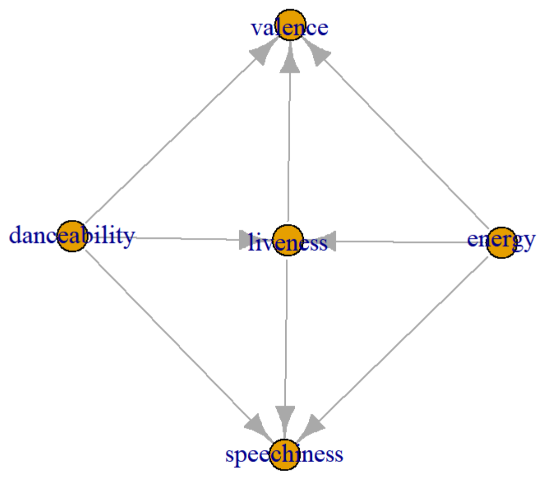
Outside of the *for* loop the final adjacency matrix (*finalMatrix*) is normalized and the results with a probability lesser than *n/3* are discarded, therefore remaining with a robust network.

The function plots that final network and returns the result as and adjacency matrix.

```
getConsensusNetwork <- function (data, n.bootstrap) {
# data should be a matrix or a dataframe
# n.boostrap refers to the number of boostrap resamples and, thus,
# the number of networks averaged

  #initializing finalmatrix with bayesian network of real data
  net<-hc(data)
  bn<-bn.fit(net, data=data)
  finalMatrix<-amat(bn)
  finalMatrix

  for(i in 1:n.bootstrap){
  array<-c(1:2017)
  sampleArray<-sample(array, 2017, replace=T)
  mat = matrix(ncol = 0, nrow = 0)
  newBoot=data.frame(mat)
    for(i in sampleArray){
      newBoot<-rbind(newBoot, data[i,])
    }

  net<-hc(newBoot)
  bn<-bn.fit(net, data=newBoot)
  matrix<-amat(bn)
  finalMatrix<-finalMatrix+matrix

}
#now we normalize the adjacency matrix with the values from 0 to 1
finalMatrix<-finalMatrix/max(finalMatrix)
result<-finalMatrix
#a threshold is stablished on n/3 and the smaller edges are discarded
finalMatrix[finalMatrix >= .33] <- 1

#finally the graph will be plotted
net<-graph_from_adjacency_matrix(finalMatrix)
plot(net)

return(result)

}
```

Execution of the algorithm, plots the consensus Network of all values > 0.33 and also de adjacency matrix:

```
getConsensusNetwork(data, 10)
```



```
##              danceability     energy   liveness speechiness    valence
## danceability            0 0.09090909 0.90909091   1.0000000 1.00000000
## energy                  0 0.00000000 0.90909091   0.3636364 0.90909091
## liveness                0 0.09090909 0.00000000   0.7272727 0.45454545
## speechiness             0 0.09090909 0.09090909   0.0000000 0.09090909
## valence                 0 0.09090909 0.27272727   0.0000000 0.00000000
```

Exercise 8

In one of the tutorials we show how the marginal probabilities can be approached by sampling a Bayesian network. In this exercise you will have to extend that idea to approximate conditional probabilities. That is, you will have to create an R function that, given a network, two nodes A and B, the value for node $\mathrm{B(b)}$ and the number of samples, gets an approximate value for $\mathrm{P(A|B = b)}$. This is the structure of the function to implement

```
getApprConditional <- function (net, node, conditional.node, evidence, nsamples) {
# net is the network, as in the tutorial
# node is the A node
# conditional.node is the B node
# evidence is the value b for B
# nsamples is the number of samples used to approximate the probability
}
```

Solution

First of all we load de mushroom data and store it in a variable called data:

```
data<-load("UCI_mushroom.Rdata")
data<-mushroom
```

We perform the hill-climbing algorithm on all the previous data and we create the bayesian entwork with *grain* function and also applying *smoothing* since there can be 0 values taken as *NAs*.

```
bn.data <- hc(data[, 1:22])
net.grain <- grain(as(amat(bn.data), "graphNEL"), data=data[, 1:22], smooth=1/dim(data)[1])
```

Given two nodes (one conditional to the other one) an evidence and a number of samples the goal is to approximate the values using the previously created bayesian network with *grain*. For that, the **simulate** function from the tutorial is used to create n amount of samples from the bayesian network. Then on the node the number of appearings of the evidence is taken into account to get the approximate value of the conditional probability

```
getApprConditional <- function (net, node, conditional.node, evidence, nsamples) {
# net is the network, as in the tutorial
# node is the A node
# conditional.node is the B node
# evidence is the value b for B
# nsamples is the number of samples used to approximate the probability
  sample <- simulate(net, nsim=nsamples)

  trueVec<-sample[, conditional.node]==evidence
  nodeCol<-sample[, node]
  result<-nodeCol[trueVec]
  finalResult<-summary(result)/length(result)

  return(finalResult)
}
```

Execution of the previous function being *EDIBILITY* the node where the values will be approximated.

```
getApprConditional(net.grain, "EDIBILITY", "BRUISES", "t", 10)
```

```
##         e         p
## 0.6666667 0.3333333
```