Tipología y ciclo de vida de los datos

Iván Maseda Zurdo y Lucas Rey Pitaluga

Junio 2021

Descripción del dataset.

Presentación

Hemos elegido el dataset “Water Quality” de Kaggle (https://www.kaggle.com/adityakadiwal/water-potability). En él, tenemos datos de potabilidad del agua. A partir de estos datos, vamos a utilizar distintos métodos de imputación de valores vacíos o nulos, múltiples formas de tratar los valores extremos y aplicaremos varios algoritmos de clasificación.

Compararemos nuestros resultados de nuestra clasificación con los obtenidos por otros usuarios de Kaggle hasta la fecha.

Carga de datos:

```
data <- read.csv("water_potability.csv", sep = ",", strip.white = TRUE,
header = TRUE, na.strings = "")
#data_plot <- read.csv("water_potability.csv", sep = ",", strip.white =
TRUE, header = TRUE, na.strings = "")

library(skimr)
library(Hmisc)

## Loading required package: lattice

## Loading required package: survival

## Loading required package: Formula

## Loading required package: ggplot2

##
## Attaching package: 'Hmisc'

## The following objects are masked from 'package:base':
##
##     format.pval, units

skim(data)
```

Data summary

Name	data
Number of rows	3276
Number of columns	10

_____

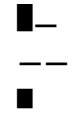
Column type frequency:

numeric	10

_____

Group variables	None

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
ph	491	0.85	7.08	1.59	0.00	6.09	7.04	8.06	14.00	
Hardness	0	1.00	196.37	32.88	47.43	176.85	196.97	216.67	323.12	
Solids	0	1.00	22014.09	8768.57	320.94	15666.69	20927.83	27332.76	61227.20	
Chloramines	0	1.00	7.12	1.58	0.35	6.13	7.13	8.11	13.13	
Sulfate	781	0.76	333.78	41.42	129.00	307.70	333.07	359.95	481.03	
Conductivity	0	1.00	426.21	80.82	181.48	365.73	421.88	481.79	753.34	
Organic_carbon	0	1.00	14.28	3.31	2.20	12.07	14.22	16.56	28.30	
Trihalomethanes	162	0.95	66.40	16.18	0.74	55.84	66.62	77.34	124.00	
Turbidity	0	1.00	3.97	0.78	1.45	3.44	3.96	4.50	6.74	
Potability	0	1.00	0.39	0.49	0.00	0.00	0.00	1.00	1.00	

En nuestro conjunto de datos tenemos 3276 registros con 10 variables. Para cada registro tenemos distintas medidas de calidad del agua relacionadas con su acidez,

dureza o sólidos en suspensión entre otras. A partir de estas variables, obtenemos una clasificación de cada registro en cuanto a su potabilidad, donde 1 significa que es potable para el consumo humano y 0 que no.

A continuación mostramos las variables de nuestro conjunto de datos.

valor de pH: El PH es un parámetro importante en la evaluación del equilibrio ácido-base del agua. También es el indicador de la condición ácida o alcalina del estado del agua. La OMS ha recomendado el límite máximo permisible de pH de 6,5 a 8,5. Los rangos de investigación actuales fueron de 6,52 a 6,83, que se encuentran en el rango de los estándares de la OMS.

Dureza: La dureza es causada principalmente por sales de calcio y magnesio. Estas sales se disuelven a partir de depósitos geológicos a través de los cuales viaja el agua. El tiempo que el agua está en contacto con el material que produce dureza ayuda a determinar cuánta dureza hay en el agua cruda. La dureza se definió originalmente como la capacidad del agua para precipitar el jabón causado por el calcio y el magnesio.

Sólidos (sólidos disueltos totales - TDS): El agua tiene la capacidad de disolver una amplia gama de minerales o sales inorgánicos y algunos orgánicos, como potasio, calcio, sodio, bicarbonatos, cloruros, magnesio, sulfatos, etc. Estos minerales producen un sabor no deseado y un color diluido en apariencia de agua. Este es un parámetro importante para el uso del agua. El agua con alto valor de TDS indica que el agua está altamente mineralizada. El límite deseable de TDS es de 500 mg / L y el límite máximo es de 1000 mg / L que se prescribe para beber.

Cloraminas: El cloro y la cloramina son los principales desinfectantes que se utilizan en los sistemas públicos de agua. Las cloraminas se forman con mayor frecuencia cuando se agrega amoníaco al cloro para tratar el agua potable. Los niveles de cloro de hasta 4 miligramos por litro (mg / L o 4 partes por millón (ppm)) se consideran seguros en el agua potable.

Sulfato: Los sulfatos son sustancias naturales que se encuentran en minerales, suelo y rocas. Están presentes en el aire ambiente, el agua subterránea, las plantas y los alimentos. El principal uso comercial del sulfato es en la industria química. La concentración de sulfato en el agua de mar es de aproximadamente 2700 miligramos por litro (mg / L). Varía de 3 a 30 mg / L en la mayoría de los suministros de agua dulce, aunque se encuentran concentraciones mucho más altas (1000 mg / L) en algunas ubicaciones geográficas.

Conductividad: El agua pura no es un buen conductor de corriente eléctrica, más bien es un buen aislante. El aumento de la concentración de iones mejora la conductividad eléctrica del agua. Generalmente, la cantidad de sólidos disueltos en el agua determina la conductividad eléctrica. La conductividad eléctrica (EC) en realidad mide el proceso iónico de una solución que le permite transmitir corriente. Según los estándares de la OMS, el valor de CE no debe exceder los 400 μS / cm.

Carbón orgánico: El carbono orgánico total (TOC) en las fuentes de agua proviene de la materia orgánica natural en descomposición (NOM), así como de fuentes sintéticas. TOC es una medida de la cantidad total de carbono en compuestos orgánicos en agua pura. Según la EPA de EE. UU. <2 mg / L como TOC en agua tratada / potable y <4 mg / L en el agua de origen que se utiliza para el tratamiento.

Trihalometanos: Los THM son sustancias químicas que se pueden encontrar en el agua tratada con cloro. La concentración de THM en el agua potable varía según el nivel de material orgánico en el agua, la cantidad de cloro necesaria para tratar el agua y la temperatura del agua que se está tratando. Los niveles de THM de hasta 80 ppm se consideran seguros en el agua potable.

Turbiedad: La turbiedad del agua depende de la cantidad de materia sólida presente en estado suspendido. Es una medida de las propiedades emisoras de luz del agua y la prueba se utiliza para indicar la calidad de la descarga de desechos con respecto a la materia coloidal. El valor medio de turbidez obtenido para Wondo Genet Campus (0,98 NTU) es inferior al valor recomendado por la OMS de 5,00 NTU.

Potabilidad: Indica si el agua es segura para el consumo humano, donde 1 significa potable y 0 significa no potable.

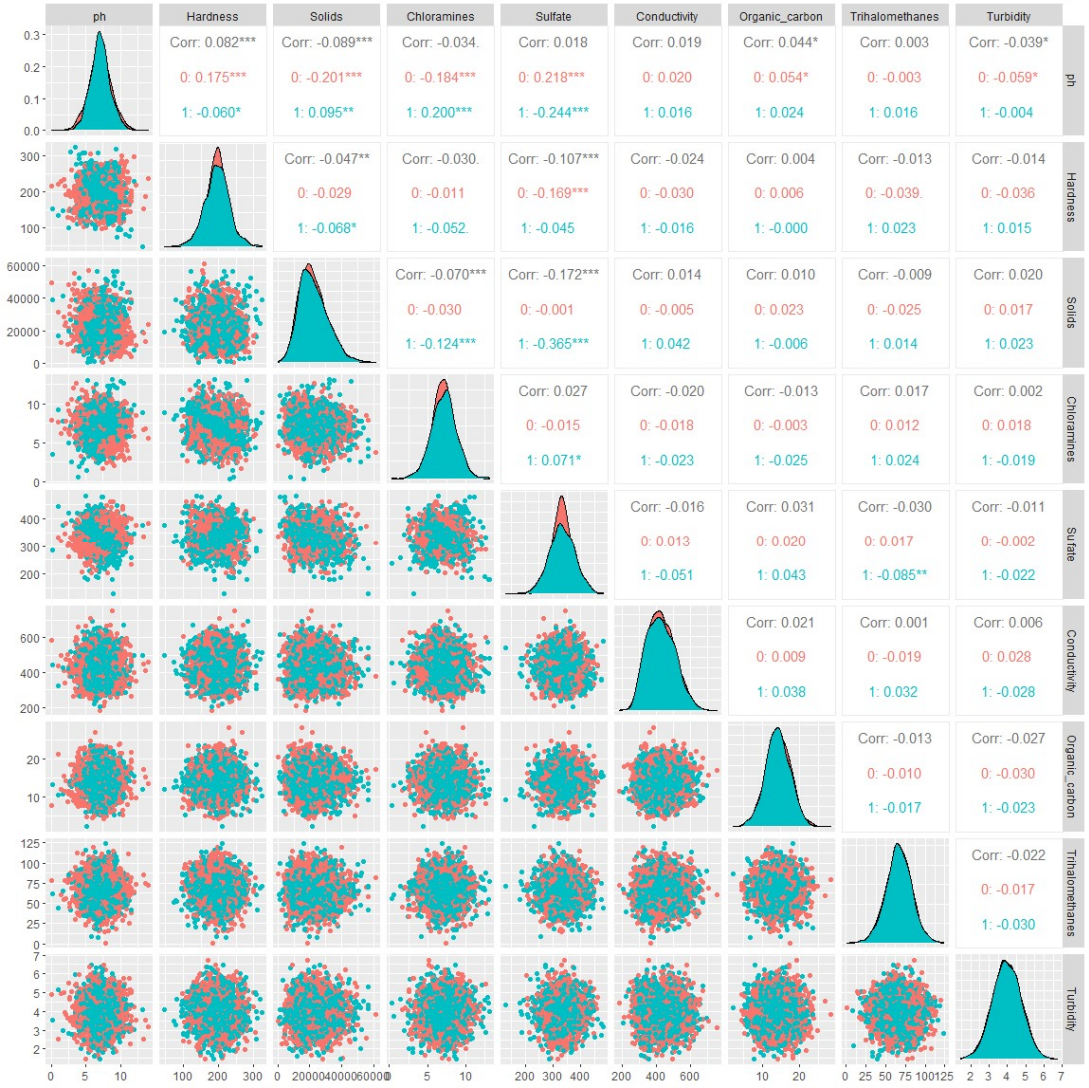
Distribución

Pasamos a hacer una descripción de nuestras variables en función de la variable objetivo, "Potability":

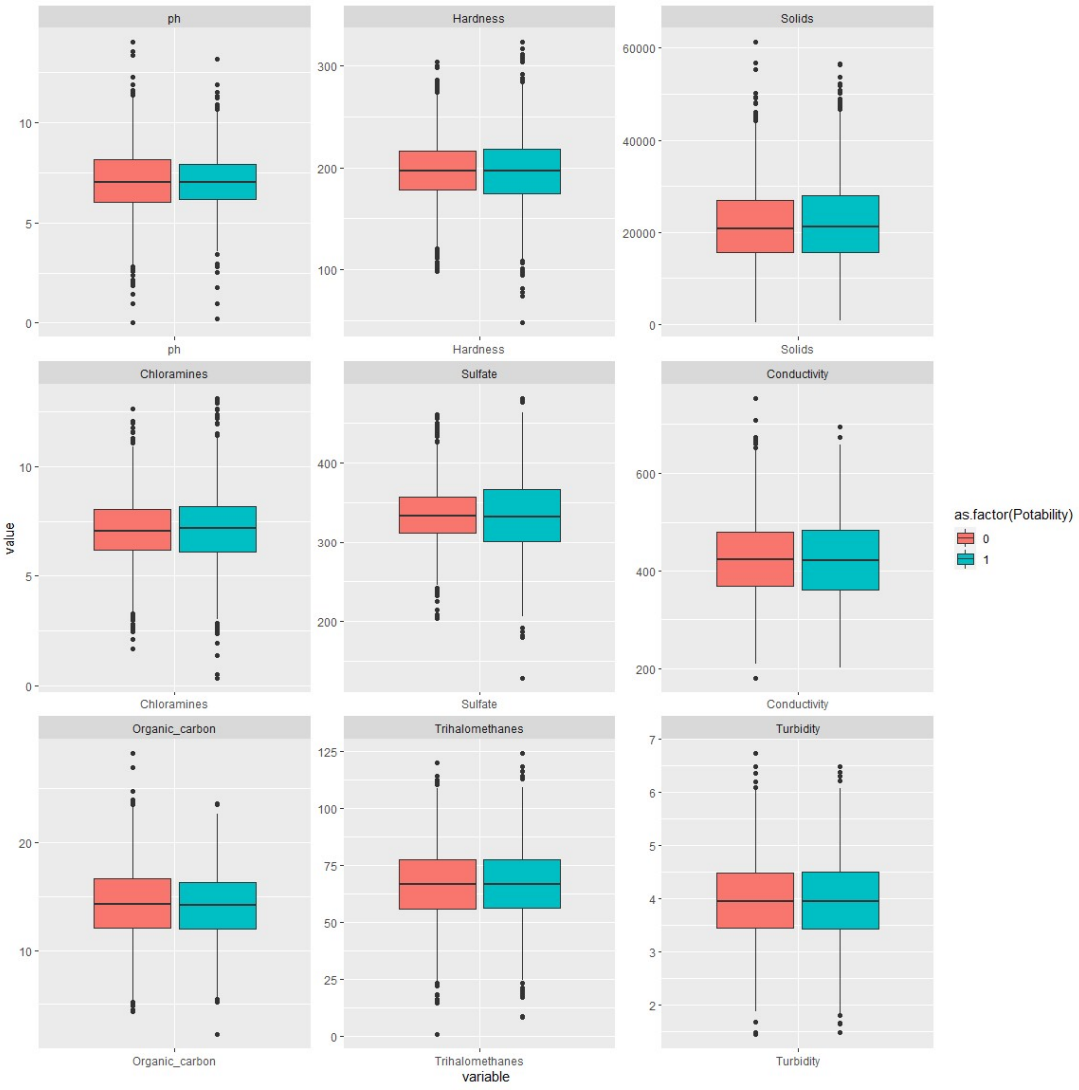
```
describe(as.factor(data$Potability))
```

```
## as.factor(data$Potability)
##        n  missing distinct
##     3276        0        2
##
## Value         0    1
## Frequency  1998 1278
## Proportion 0.61 0.39
```

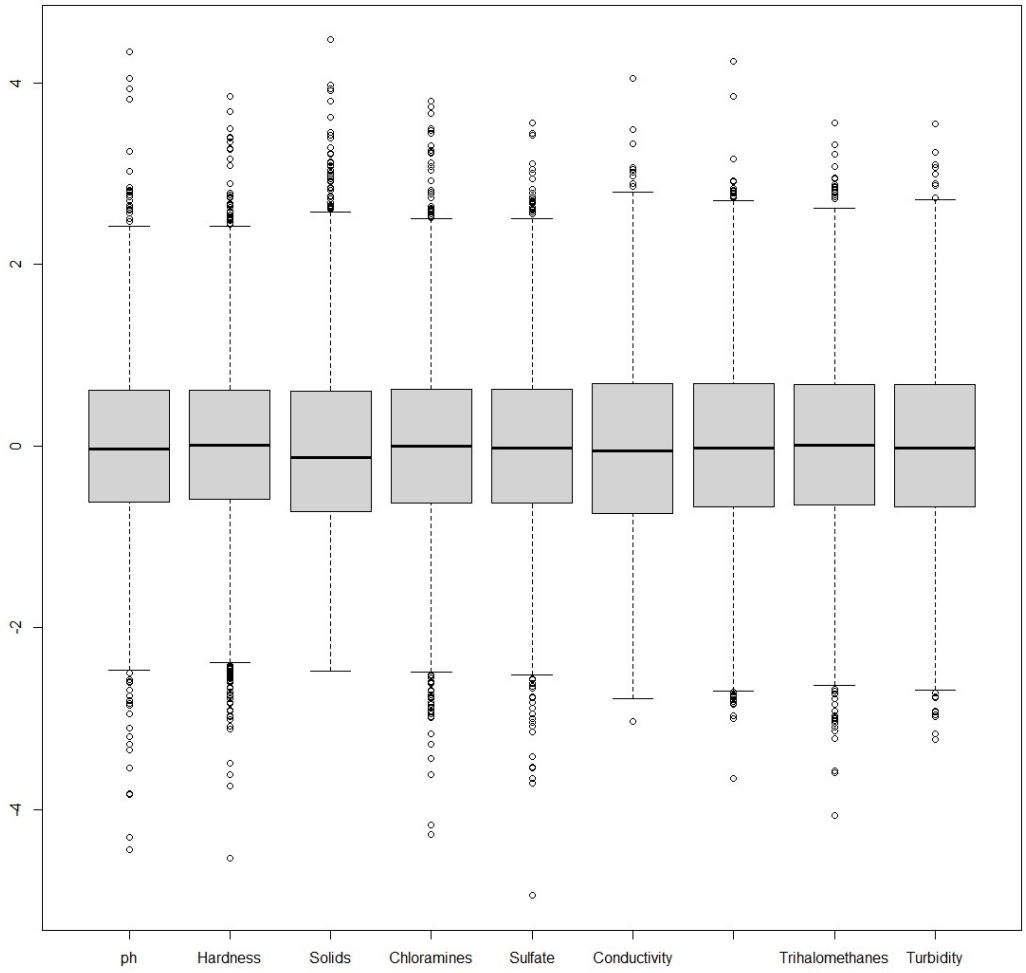
```
library(GGally)
ggpairs(data, columns = 1:9, ggplot2::aes(colour=as.factor(Potability)),
progress = FALSE)
```




```
library(reshape2)
data_m <- melt(data, id.vars = "Potability")

ggplot(data = data_m, aes(x=variable, y=value,
fill=as.factor(Potability))) +
             geom_boxplot() + facet_wrap(~variable, scales="free")
```




```
boxplot(scale(data[,1:9]))
```

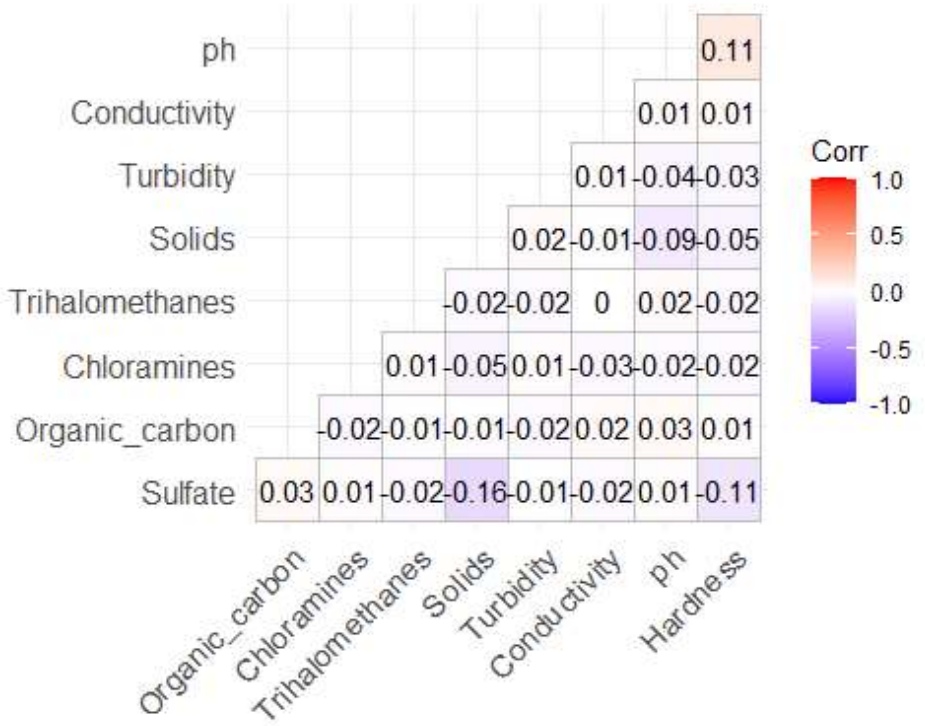


Correlación

Mostramos las correlaciones existentes entre las distintas variables de nuestro conjunto de datos.

```
cor_mat <- cor(data[1:9], use="complete.obs")
cor_mat

##                          ph    Hardness       Solids  Chloramines
Sulfate
## ph                1.00000000  0.10894811 -0.087614993 -0.024768491
0.010524348
## Hardness          0.10894811  1.00000000 -0.053268885 -0.022684975 -
0.108520618
## Solids           -0.08761499 -0.05326888  1.000000000 -0.051789064 -
0.162769204
## Chloramines      -0.02476849 -0.02268498 -0.051789064  1.000000000
```

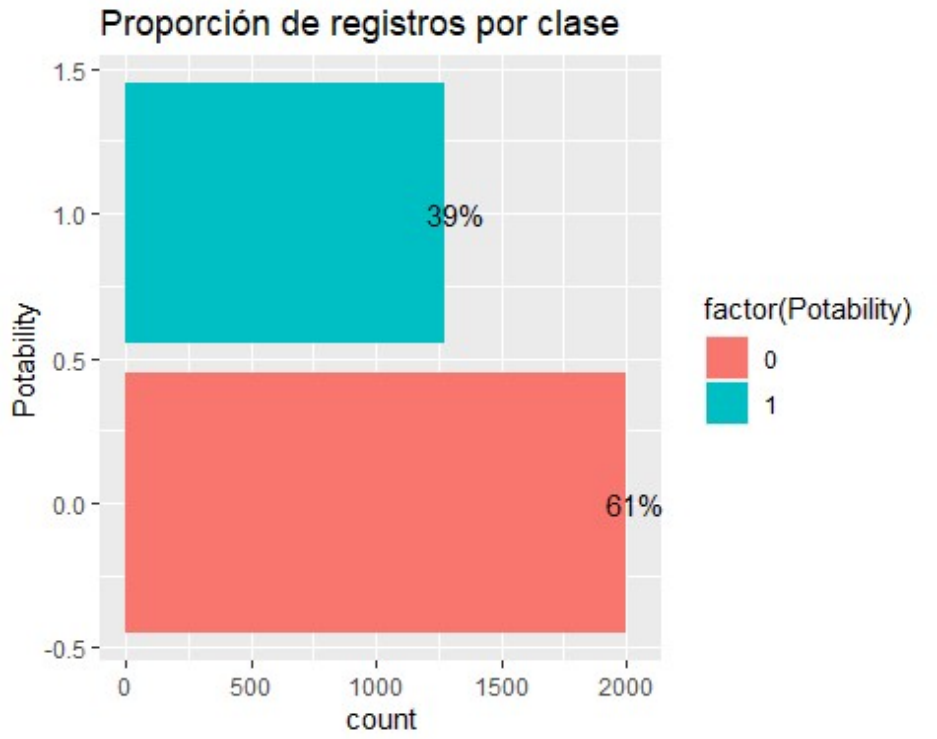
```
0.006254057
## Sulfate           0.01052435 -0.10852062 -0.162769204  0.006254057
1.000000000
## Conductivity      0.01412785  0.01173055 -0.005197862 -0.028276649 -
0.016192287
## Organic_carbon    0.02837522  0.01322386 -0.005484046 -0.023807630
0.026775563
## Trihalomethanes   0.01827788 -0.01540038 -0.015667788  0.014989930 -
0.023346904
## Turbidity        -0.03584899 -0.03483094  0.019409428  0.013136570 -
0.009933881
##                 Conductivity Organic_carbon Trihalomethanes
Turbidity
## ph                0.014127848    0.028375219     0.018277876 -
0.035848994
## Hardness          0.011730548    0.013223861    -0.015400382 -
0.034830942
## Solids           -0.005197862   -0.005484046    -0.015667788
0.019409428
## Chloramines      -0.028276649   -0.023807630     0.014989930
0.013136570
## Sulfate          -0.016192287    0.026775563    -0.023346904 -
0.009933881
## Conductivity      1.000000000    0.015646727     0.004888475
0.012494892
## Organic_carbon    0.015646727    1.000000000    -0.005667486 -
0.015428291
## Trihalomethanes   0.004888475   -0.005667486     1.000000000 -
0.020497369
## Turbidity         0.012494892   -0.015428291    -0.020497369
1.000000000

library(ggcorrplot)
ggcorrplot(cor_mat, hc.order = TRUE, type = "lower", lab = TRUE, insig =
"blank")
```



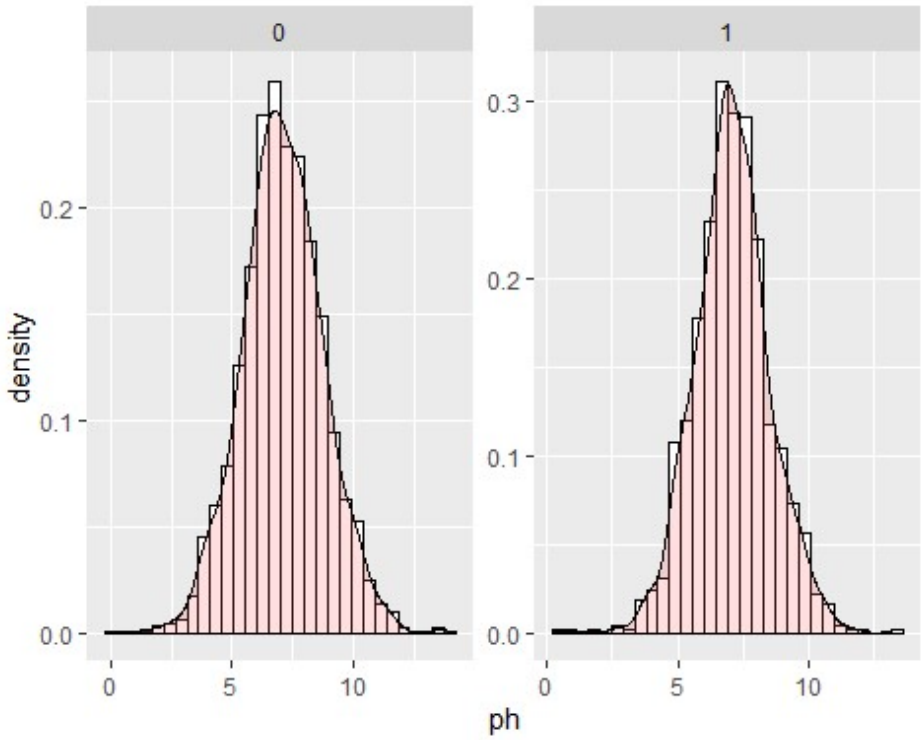
Proporción y densidad

A continuación mostramos la proporción para cada uno de los posibles valores de "Potability", las densidades e histogramas de distribución.

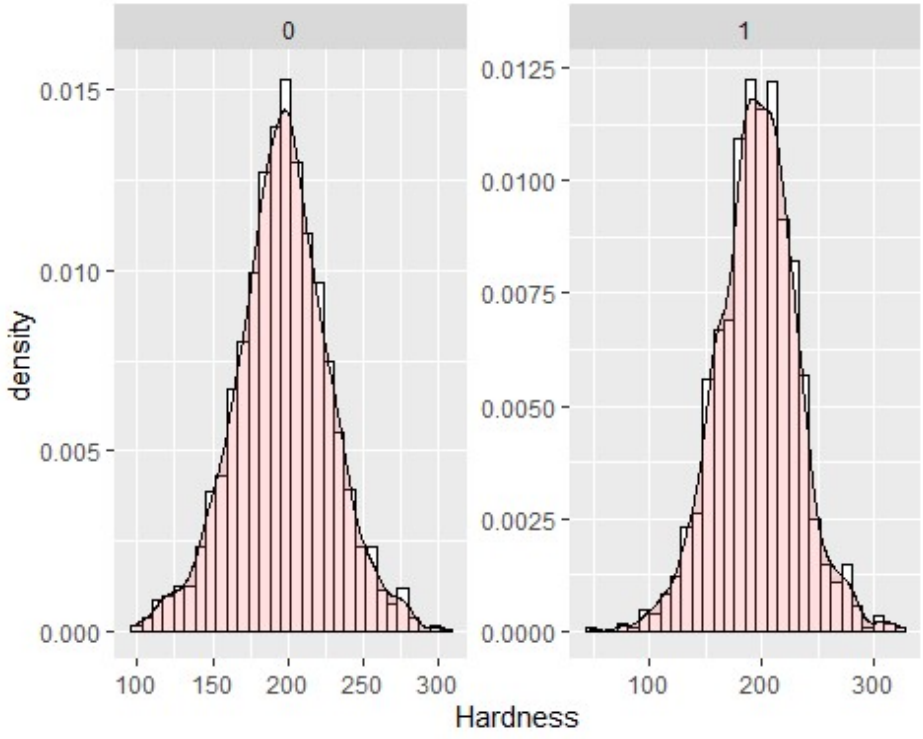
```
ggplot(data, aes(y = Potability, fill = factor(Potability))) + geom_bar()
+ labs(title = "Proporción de registros por clase") + geom_text(stat =
"count", aes(label = scales::percent(..count../sum(..count..))), nudge_x
= 40)
```



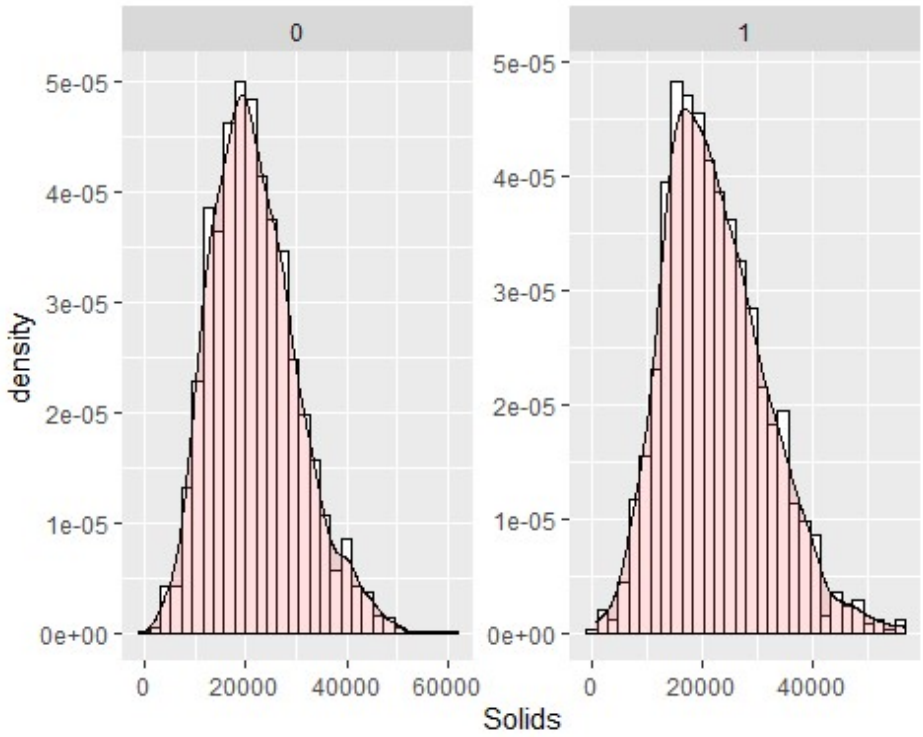

```
library(plyr)

##
## Attaching package: 'plyr'

## The following objects are masked from 'package:Hmisc':
##
##     is.discrete, summarize

for (i in names(data)){
  plt <- ggplot(data, aes_string(x=i)) +
         geom_histogram(aes(y=..density..), colour="black",
fill="white")+
         geom_density(alpha=.2, fill="#FF6666") + facet_wrap(~Potability,
scales="free")
  print(plt)
}

## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.

## Warning: Removed 491 rows containing non-finite values (stat_bin).

## Warning: Removed 491 rows containing non-finite values (stat_density).
```



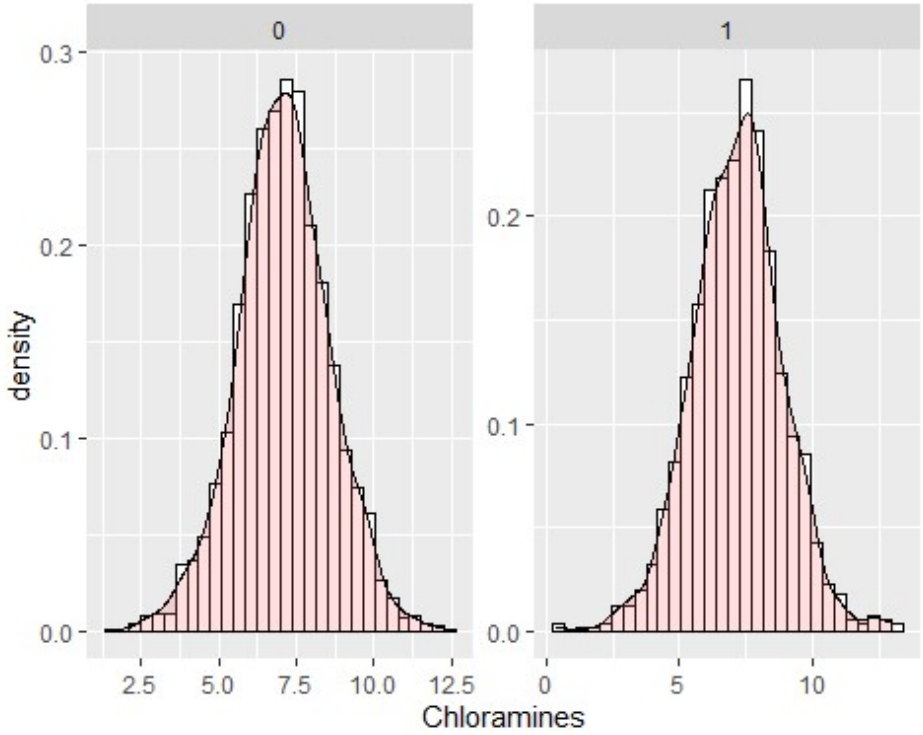

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



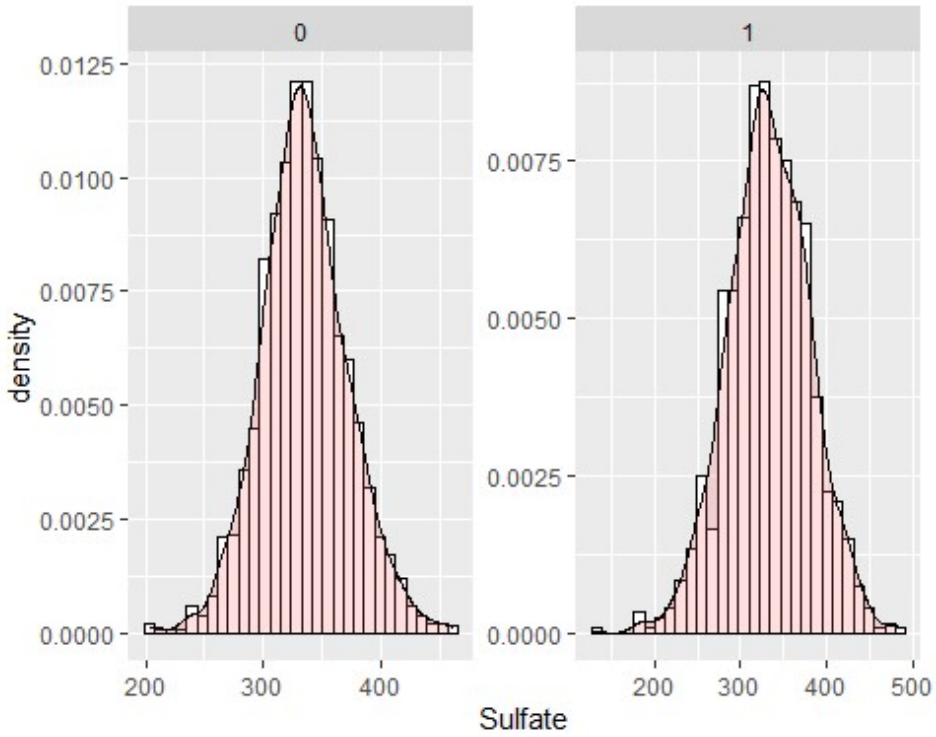

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



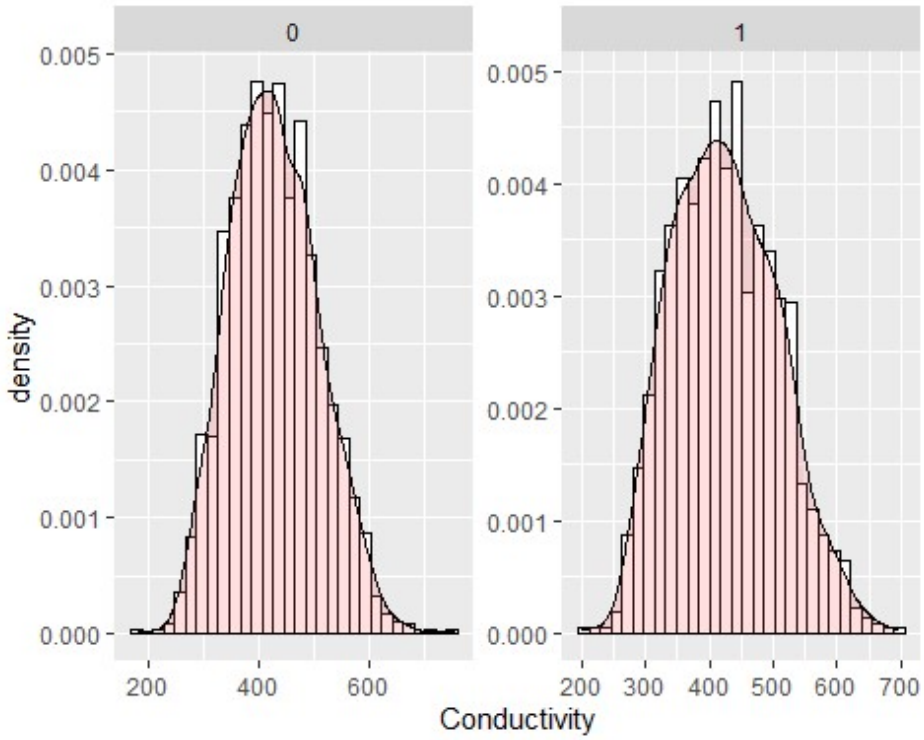

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 781 rows containing non-finite values (stat_bin).
```

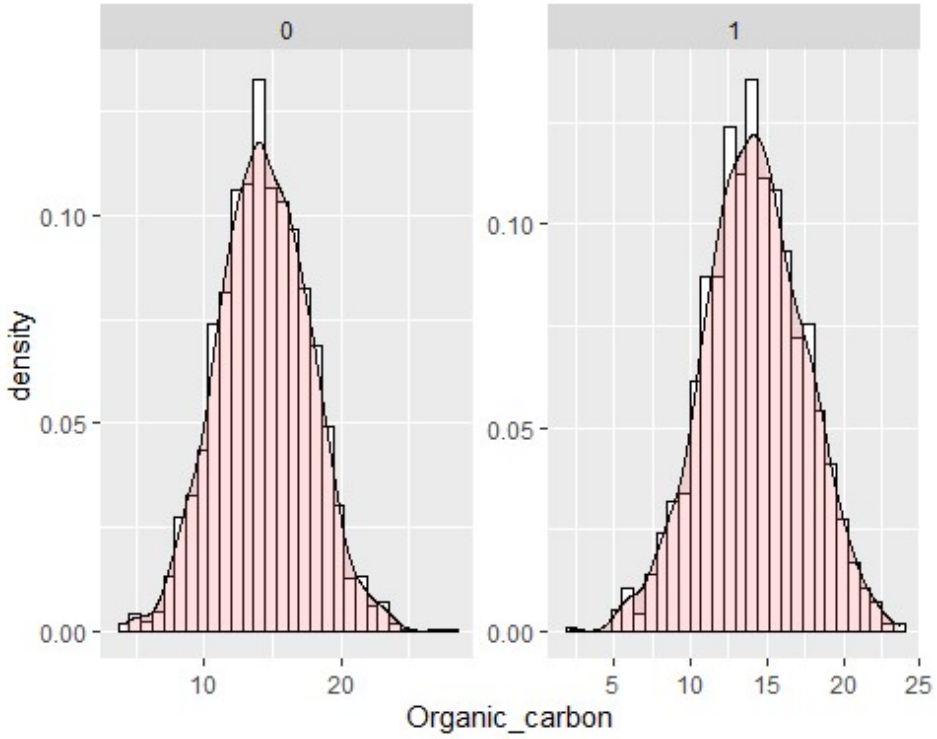
```
## Warning: Removed 781 rows containing non-finite values (stat_density).
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



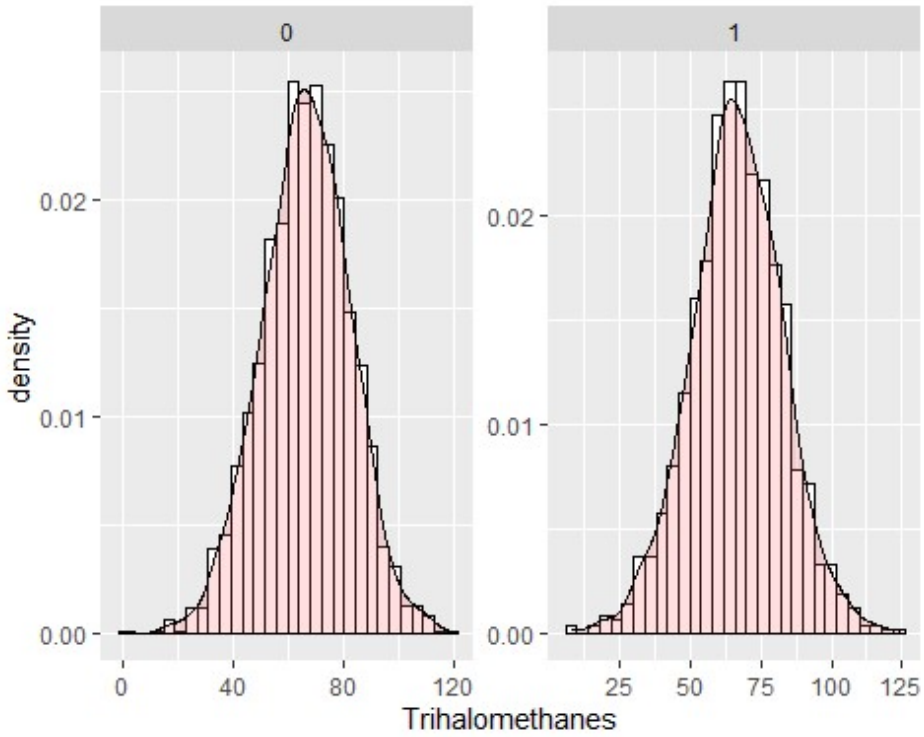

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



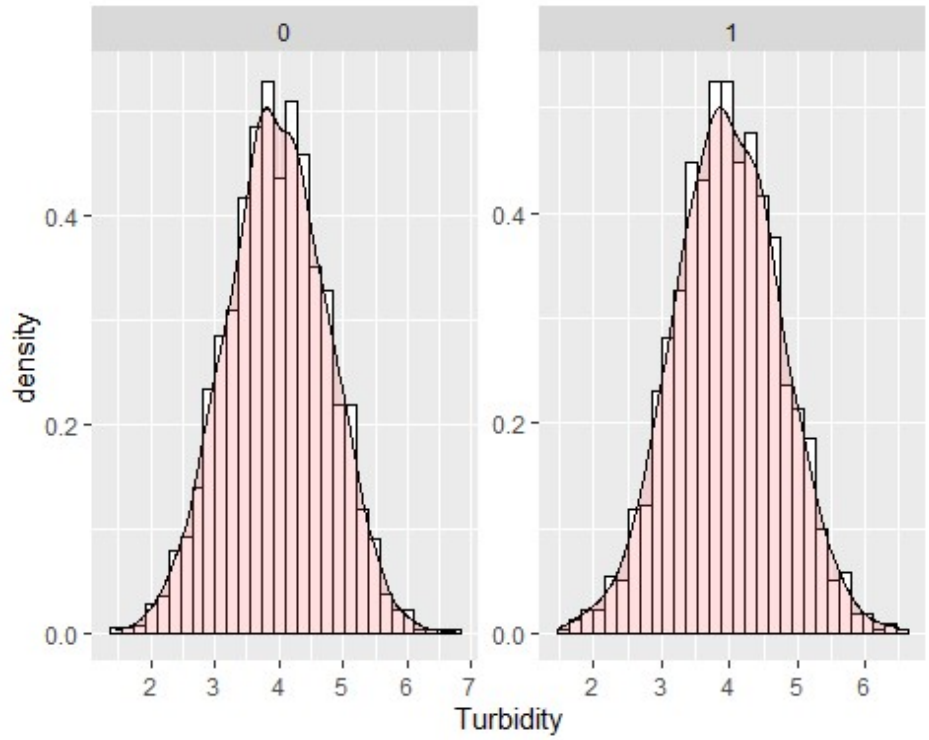

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 162 rows containing non-finite values (stat_bin).
```

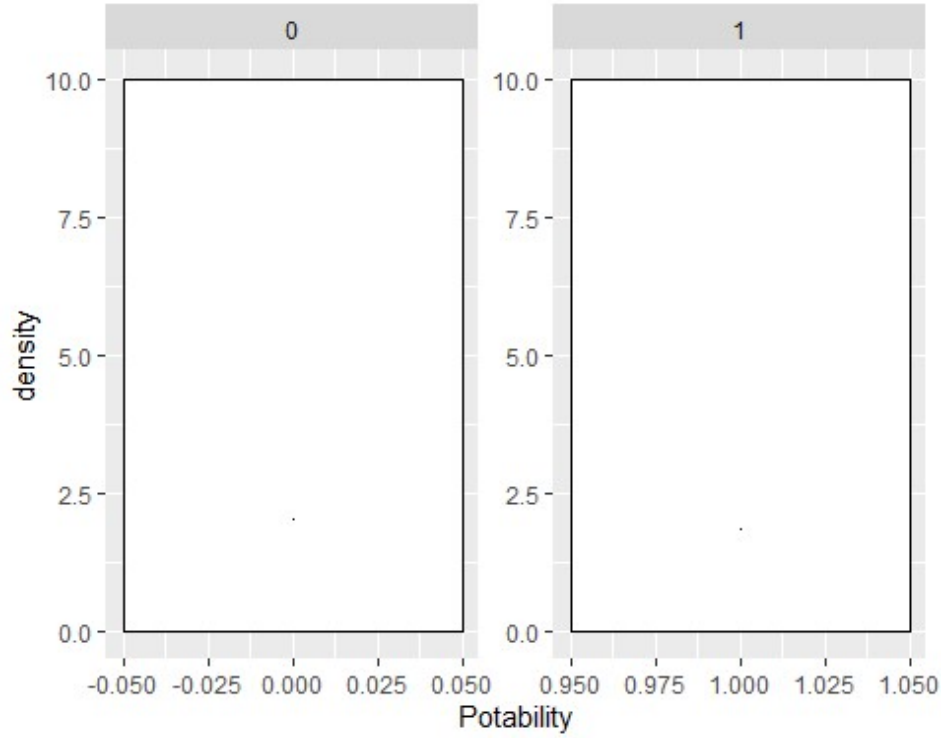
```
## Warning: Removed 162 rows containing non-finite values (stat_density).
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Integración y selección de los datos de interés a analizar.

Tras el análisis preliminar que hemos realizado, hemos podido observar que nuestros datos están correctamente integrados, no tenemos diferencias de medidas o rangos, por lo que no hemos tenido que realizar ninguna manipulación de los datos en este sentido.

En cuanto a la selección de los datos, guardaremos todos los registros y todas las variables que tenemos, puesto que cada una de ellas es importante a la hora de analizar la potabilidad del agua.

Haremos un tratamiento de los elementos vacíos y de los valores extremos en el siguiente apartado.

Limpieza de los datos.

¿Los datos contienen ceros o elementos vacíos? ¿Cómo gestionarías cada uno de estos casos?

Tratamiento de valores nulos, el número máximo de elementos vacíos por registro es:

```
max(rowSums(is.na(data)))
```

```
## [1] 3
```

El número de filas que presentan al menos un valor nulo:

```
sum(!complete.cases(data))
```

```
## [1] 1265
```

Añadiremos una columna, que llamaremos “na_count”, con el número de valores nulos por registro y presentaremos un top 20:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:plyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize
```

```
## The following objects are masked from 'package:Hmisc':
##
##     src, summarize

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
data_na_count <- data
data_na_count$na_count <- apply(data, 1, function(x) sum(is.na(x)))
head(data_na_count %>% slice_max(na_count, n = 20), 20)
```

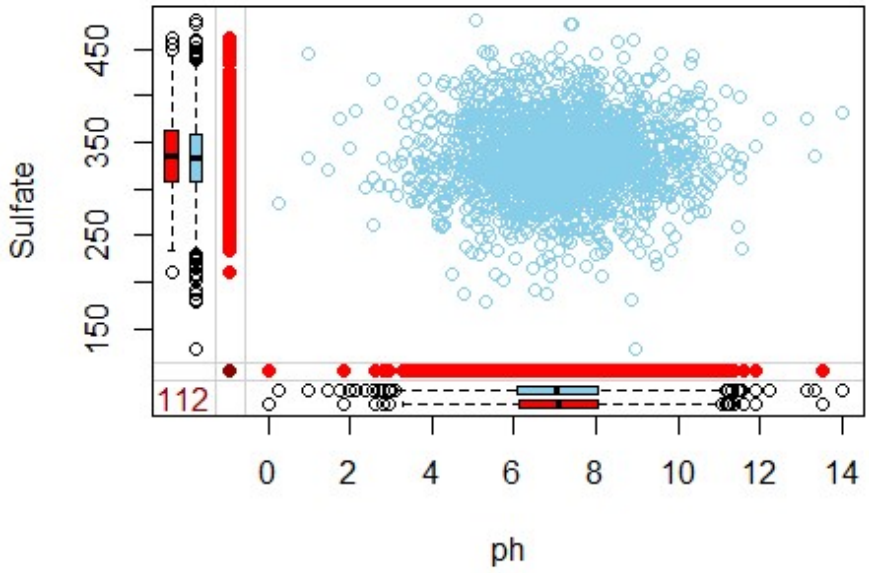
```
##          ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1        NA 167.3861 20944.62    4.963124       NA     566.3393
11.318807
## 2        NA 229.7713 16162.26    4.933662       NA     448.8460
8.816487
## 3        NA 143.3002 16263.17    6.229737       NA     503.6641
19.585497
## 4        NA 221.6201 11954.70    6.657053       NA     391.2387
12.961433
## 5        NA 202.0799 12519.09    7.627524       NA     399.8834
12.748217
## 6        NA 226.7656 39942.95    8.594715       NA     538.6218
13.744749
## 7        NA 184.9937 19764.63    7.505092       NA     293.4780
12.934160
## 8        NA 209.5317 44982.73    8.898024       NA     349.3852
15.653299
## 9        NA 193.0913 17777.10    6.087949       NA     543.1150
10.717588
## 10       NA 227.4350 22305.57   10.333918       NA     554.8201
16.331693
## 11       NA 232.2805 14787.21    5.474915       NA     383.9817
12.166937
## 12       NA 143.4537 19942.27    5.890755       NA     427.1307
22.469892
## 13       NA 155.0557 20557.24    8.187319       NA     290.1810
16.622255
## 14       NA 229.4857 35729.69    8.810843 384.9438     296.3975
16.927092
## 15       NA 103.4648 27420.17    8.417305       NA     485.9745
11.351133
## 16 5.519126 168.7286 12531.60    7.730723       NA     443.5704
18.099078
## 17 7.812804 196.5839 42550.84    7.334648       NA     442.5458
```

```
14.666917
## 18       NA 146.9702 16367.46    7.687037       NA     468.9450
18.601381
## 19       NA 157.0013 20067.11    8.071051       NA     490.9097
17.322946
## 20       NA 247.6616 19549.50    8.769670       NA     438.8658
16.624654
##    Trihalomethanes Turbidity Potability na_count
## 1               NA  3.679795          0        3
## 2               NA  4.600928          0        3
## 3               NA  3.451740          1        3
## 4               NA  3.282061          0        3
## 5               NA  4.439215          0        3
## 6               NA  4.863378          0        3
## 7               NA  3.017985          0        3
## 8               NA  3.832479          0        3
## 9               NA  3.878664          0        3
## 10        45.38282  4.133423          0        2
## 11        86.08073  5.029167          0        2
## 12        53.12409  2.907564          0        2
## 13        59.62206  4.089908          0        2
## 14              NA  3.855602          0        2
## 15        67.86996  4.620793          0        2
## 16              NA  3.758996          0        2
## 17              NA  6.204846          0        2
## 18        49.56197  4.664350          0        2
## 19        71.96926  4.417560          0        2
## 20        64.94901  3.777737          0        2
```

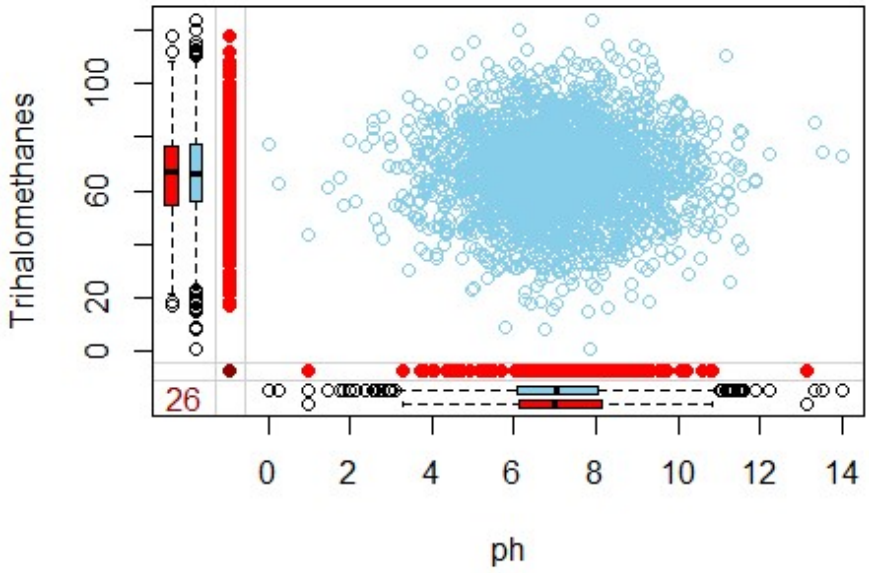
Mediante la librería “VIM” podemos visualizar algunos gráficos interesantes sobre los valores nulos. Hemos visto que los atributos “ph”, “Sulfate” y “Trihalomethanes” son los que presentan valores nulos, analizaremos la co-ocurrencia de estos combinando los 3 atributos:

```
library(VIM)

## Loading required package: colorspace

## Loading required package: grid

## VIM is ready to use.

## Suggestions and bug-reports can be submitted at:
https://github.com/statistikat/VIM/issues

##
## Attaching package: 'VIM'

## The following object is masked from 'package:datasets':
##
##     sleep
```

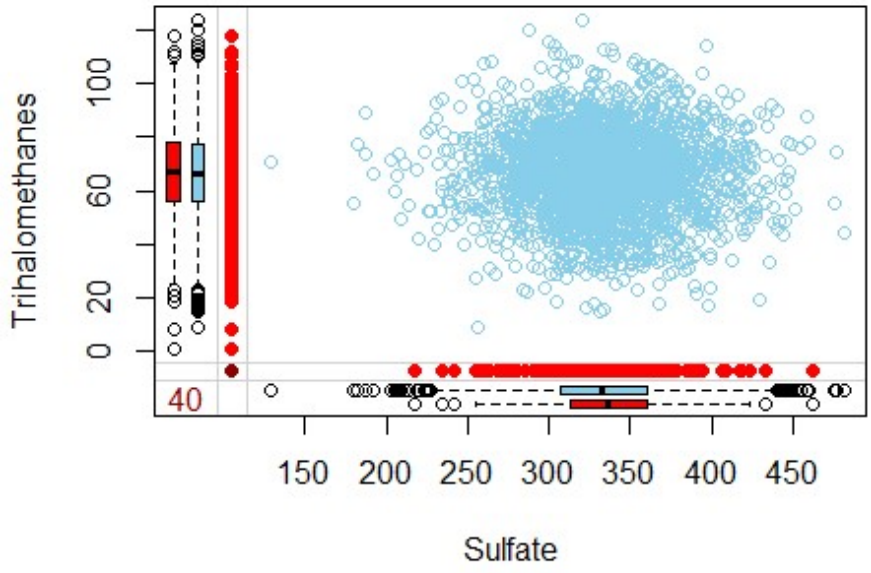
```
marginplot(data[c(1,5)])
```




```
marginplot(data[c(1,8)])
```



```
marginplot(data[c(5,8)])
```



Vemos que hay una co-ocurrencia importante entre ph y Sulfate, debemos tenerlo presente. Una buena opción quizás sería descartar esos registros, como última opción siempre tenemos la posibilidad, pero tratemos de lidiar con ello y ver los resultados que nos ofrece. Veamos otras opciones de visaulización, mediante la función md.pattern() podemos ver la co-ocurrencia de valores nulos, el número de registros por patrón y obtener una imagen intuitiva de estos hechos:

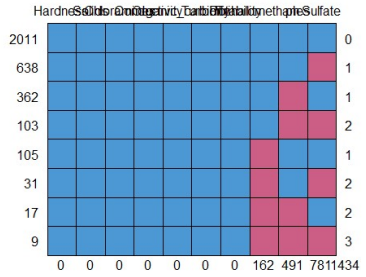
```
library(mice)
```

```
##
## Attaching package: 'mice'
```

```
## The following object is masked from 'package:stats':
##
##     filter
```

```
## The following objects are masked from 'package:base':
##
##     cbind, rbind
```

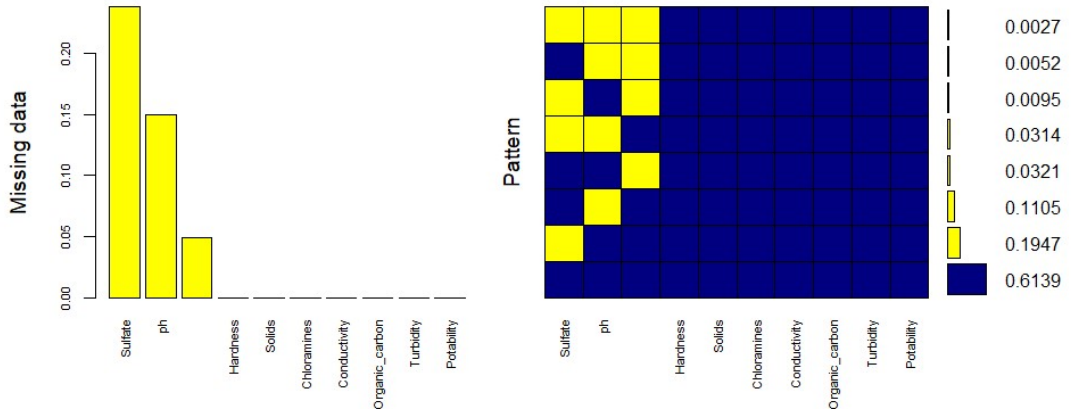
```
md.pattern(data)
```




```
##      Hardness Solids Chloramines Conductivity Organic_carbon Turbidity
## 2011        1      1           1            1              1          1
## 638         1      1           1            1              1          1
## 362         1      1           1            1              1          1
## 103         1      1           1            1              1          1
## 105         1      1           1            1              1          1
## 31          1      1           1            1              1          1
## 17          1      1           1            1              1          1
## 9           1      1           1            1              1          1
##             0      0           0            0              0          0
##      Potability Trihalomethanes  ph Sulfate
## 2011          1               1   1       1    0
## 638           1               1   1       0    1
## 362           1               1   0       1    1
## 103           1               1   0       0    2
## 105           1               0   1       1    1
## 31            1               0   1       0    2
## 17            1               0   0       1    2
## 9             1               0   0       0    3
##               0             162 491     781 1434
```

Esta es otra opción similar pero expresada en porcentajes, en función de las variables y sus valores nulos:

```
mice_plot <- aggr(data, col=c('navyblue','yellow'),
                  numbers=TRUE, sortVars=TRUE,
                  labels=names(data), cex.axis=.7,
                  gap=3, ylab=c("Missing data","Pattern"))
```




```
##
##  Variables sorted by number of missings:
##         Variable      Count
##          Sulfate 0.23840049
##               ph 0.14987790
##  Trihalomethanes 0.04945055
##         Hardness 0.00000000
##           Solids 0.00000000
##      Chloramines 0.00000000
##     Conductivity 0.00000000
##   Organic_carbon 0.00000000
##        Turbidity 0.00000000
##       Potability 0.00000000
```

Antes de abordar el tratamiento de estos valores, es conveniente observar la distribución de nuestros datos, en el apartado de presentación hemos podido ver gráficamente la distribución de todos los atributos, incluso en función de su clase. Ahora estudiaremos la normalidad multivariante, esto nos ayudará en las decisiones futuras, puesto que no podemos tratar los datos de igual manera si la distribución no es normal, por ejemplo.

Para esto emplearemos las librerias “QuantPsyc”(para la simetría), “energy” y “MVN”:

```
library(QuantPsyc)
```

```
## Loading required package: boot
```

```
##
## Attaching package: 'boot'
```

```
## The following object is masked from 'package:survival':
##
##     aml
```

```
## The following object is masked from 'package:lattice':
##
##     melanoma
```

```
## Loading required package: MASS

##
## Attaching package: 'MASS'

## The following object is masked from 'package:dplyr':
##
##     select

##
## Attaching package: 'QuantPsyc'

## The following object is masked from 'package:base':
##
##     norm
```

```
mult.norm(data)$mult.test
```

```
##            Beta-hat      kappa p-val
## Skewness   2.800127 938.509151     0
## Kurtosis 125.939849   8.596977     0
```

En el caso de "energy" debemos estudiar los registros completos, sin valores nulos. Nos servirá de referencia para el total de los datos:

```
library(energy)

complete_data <- data[complete.cases(data),]
mvnorm.etest(complete_data, R=100)
```

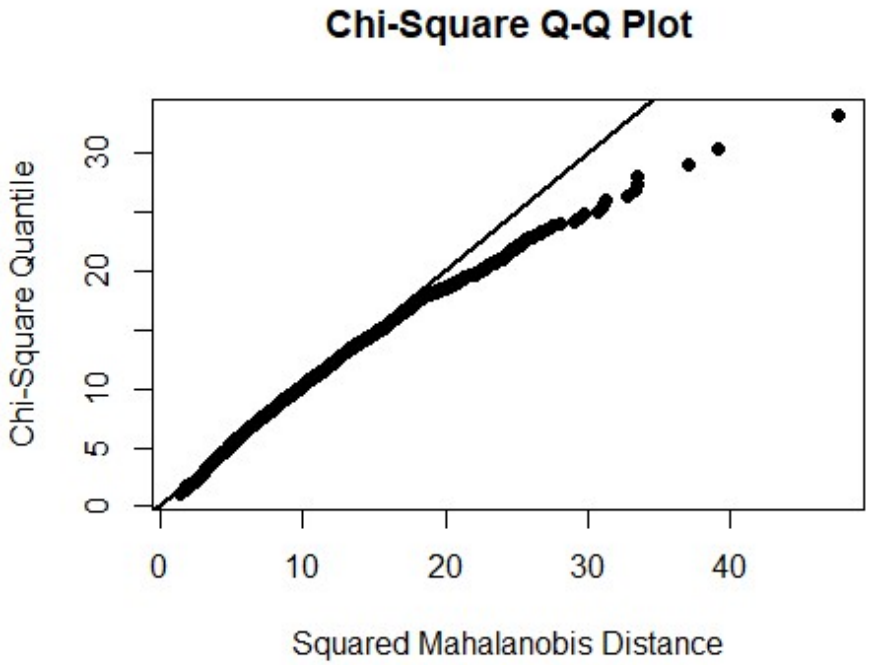
```
##
##  Energy test of multivariate normality: estimated parameters
##
## data:  x, sample size 2011, dimension 10, replicates 100
## E-statistic = 5.0004, p-value < 2.2e-16
```

La librería "MVN" nos ofrece mucha información del conjunto y atributo a atributo:

```
library(MVN)
```

```
## sROC 0.1-2 loaded
```

```
result <- mvn(complete_data, multivariatePlot = "qq", showOutliers =
TRUE)
```




```
result$multivariateNormality

##              Test        Statistic             p value Result
## 1 Mardia Skewness 939.910608187819 1.2879138782565e-89     NO
## 2 Mardia Kurtosis 8.77839347376921                   0     NO
## 3             MVN             <NA>                <NA>     NO

result$univariateNormality

##            Test        Variable Statistic   p value Normality
## 1  Shapiro-Wilk       ph           0.9964   1e-04      NO
## 2  Shapiro-Wilk    Hardness        0.9956  <0.001      NO
## 3  Shapiro-Wilk     Solids         0.9788  <0.001      NO
## 4  Shapiro-Wilk   Chloramines      0.9960  <0.001      NO
## 5  Shapiro-Wilk     Sulfate        0.9945  <0.001      NO
## 6  Shapiro-Wilk  Conductivity      0.9928  <0.001      NO
## 7  Shapiro-Wilk Organic_carbon     0.9993  0.6362      YES
## 8  Shapiro-Wilk Trihalomethanes    0.9987  0.1137      YES
## 9  Shapiro-Wilk    Turbidity       0.9993  0.6875      YES
## 10 Shapiro-Wilk   Potability       0.6229  <0.001      NO

result$Descriptives

##                    n         Mean      Std.Dev       Median
Min
## ph              2011 7.085990e+00    1.5733367     7.027297
0.2274991
## Hardness        2011 1.959681e+02   32.6350845   197.191839
```

```
73.4922337
## Solids          2011 2.191744e+04 8642.2398152 20933.512750
320.9426113
## Chloramines     2011 7.134338e+00    1.5848199     7.143907
1.3908709
## Sulfate         2011 3.332247e+02   41.2051720   332.232177
129.0000000
## Conductivity    2011 4.265264e+02   80.7125724   423.455906
201.6197368
## Organic_carbon  2011 1.435771e+01    3.3249587    14.322019
2.2000000
## Trihalomethanes 2011 6.640086e+01   16.0771095    66.542198
8.5770129
## Turbidity       2011 3.969729e+00    0.7803462     3.968177
1.4500000
## Potability      2011 4.032819e-01    0.4906785     0.000000
0.0000000
##                          Max         25th         75th        Skew
Kurtosis
## ph                 14.000000     6.089723     8.052969  0.04887379
0.61497853
## Hardness          317.338124   176.744938   216.441070 -0.08511031
0.51941063
## Solids          56488.672413 15615.665390 27182.587067  0.59500535
0.33787097
## Chloramines        13.127000     6.138895     8.109726  0.01295693
0.54366684
## Sulfate           481.030642   307.632511   359.330555 -0.04648827
0.77859915
## Conductivity      753.342620   366.680307   482.373169  0.26647085 -
0.24466795
## Organic_carbon     27.006707    12.124105    16.683049 -0.01998781
0.02638812
## Trihalomethanes   124.000000    55.952664    77.291925 -0.05134540
0.21787868
## Turbidity           6.494749     3.442915     4.514175 -0.03300219 -
0.05510014
## Potability          1.000000     0.000000     1.000000  0.39402568 -
1.84566080
```

En ambos casos rechazamos la hipótesis nula del test $p<0.05$, nuestras variables no siguen una distribución normal. También es importante observar el coeficiente de simetría, puede ser que la distribución esté condicionada por la simetría de los datos y sea conveniente aplicar una transformación logarítmica, por ejemplo, para tratar de corregirlo, pero vemos que no es el caso, también podemos ver que la diferencia entre la mediana y la media no es significativa en ninguno de los atributos.

En este punto sería conveniente tratar de eliminar outliers, en ocasiones, la falta de normalidad puede atribuirse a estos valores que se desvían mucho del resto y afectan

a la distribución de los datos, y hemos visto en el apartado de presentacón, en los boxplots concretamente, la presencia de muchos valores de este tipo.

Para esto crearemos dos funciones, una para realizar el conocido como test de Tukey, y otra en función de la desviación típica, las llamaremos “outliers_Tukey” y “outliers_sd”, hemos visto en la función “MVN” que parecen no afectar a la distribución, pero dejaremos las funciones creadas, ya que tras la imputación de valores, sería conveniente estudiarlo con el total de los datos:

```
outliers_Tukey <- function(x) {

  Q1 <- quantile(x, probs=.25)
  Q3 <- quantile(x, probs=.75)
  iqr = Q3-Q1

 upper_limit = Q3 + (iqr*1.5)
 lower_limit = Q1 - (iqr*1.5)

 x > upper_limit | x < lower_limit
}

remove_outliers_Tukey <- function(df, cols = names(df)) {
  for (col in cols) {
    df <- df[!outliers_Tukey(df[[col]]),]
  }
  df
}

outliers_sd <- function(x) {

    upper_limit = mean(x) + 3*sd(x)
    lower_limit = mean(x) - 3*sd(x)

    x > upper_limit | x < lower_limit
}

remove_outliers_sd <- function(df, cols = names(df)) {
  for (col in cols) {
    df <- df[!outliers_sd(df[[col]]),]
  }
  df
}
```

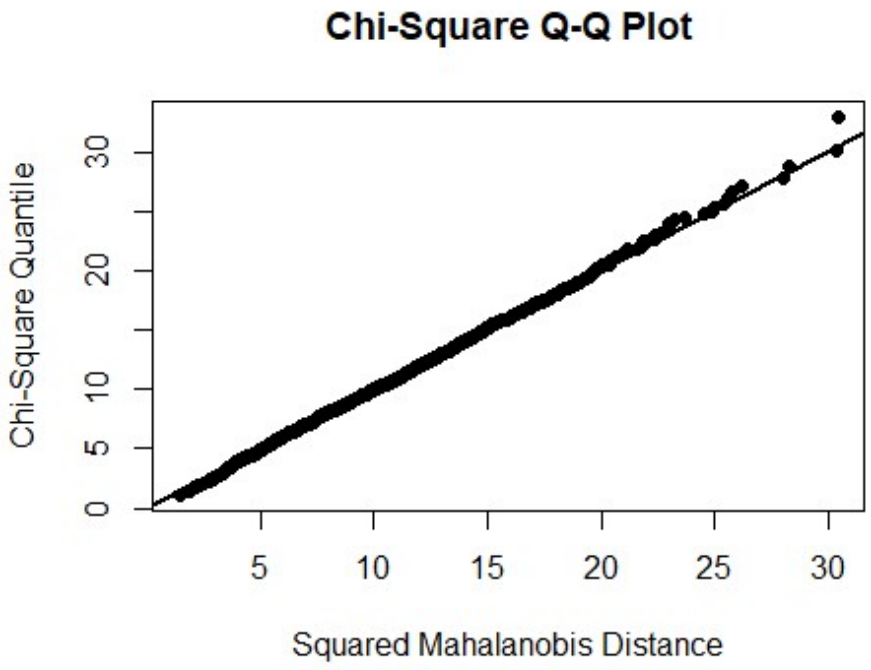
En el enunciado de la práctica vemos que se pide el estudio de outliers después de gestionar los valores nulos, pero en nuestro caso nos ayudará a decidir los metodos de imputación que aplicaremos.

Para evitar problemas con valores nulos de momento tomaremos de referencia todos los registros completos.

```
data_Tukey_outliers <- remove_outliers_Tukey(complete_data)
data_sd_outliers <- remove_outliers_sd(complete_data)
```

Aplicaremos la función “mvn”(mostrando los datos de interés únicamente) para ambos conjuntos, la función “summary” y un boxplot para apreciar gráficamente las diferencias con el conjunto original:

```
result <- mvn(data_Tukey_outliers, multivariatePlot = "qq", showOutliers
= TRUE)
```



Vemos una clara diferencia en la gráfica, pero seguimos rechazando la hipótesis nula.

```
result$multivariateNormality
```

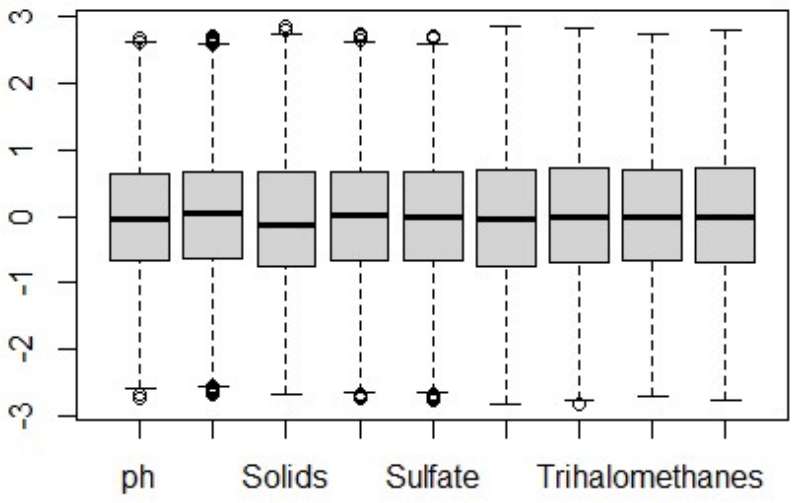
```
##              Test          Statistic              p value Result
## 1 Mardia Skewness   661.831824756402 8.85780055460113e-46     NO
## 2 Mardia Kurtosis -0.994102484711584    0.320172975816403    YES
## 3             MVN               <NA>                 <NA>     NO
```

```
summary(data_Tukey_outliers)
```

```
##        ph            Hardness         Solids         Chloramines
##  Min.   : 3.149   Min.   :117.8   Min.   :  320.9   Min.   : 3.268
##  1st Qu.: 6.115   1st Qu.:177.7   1st Qu.:15457.5   1st Qu.: 6.186
##  Median : 7.026   Median :197.4   Median :20503.4   Median : 7.138
##  Mean   : 7.073   Mean   :196.3   Mean   :21430.4   Mean   : 7.130
##  3rd Qu.: 7.993   3rd Qu.:215.4   3rd Qu.:26678.4   3rd Qu.: 8.066
##  Max.   :10.905   Max.   :275.7   Max.   :44069.3   Max.   :11.000
```

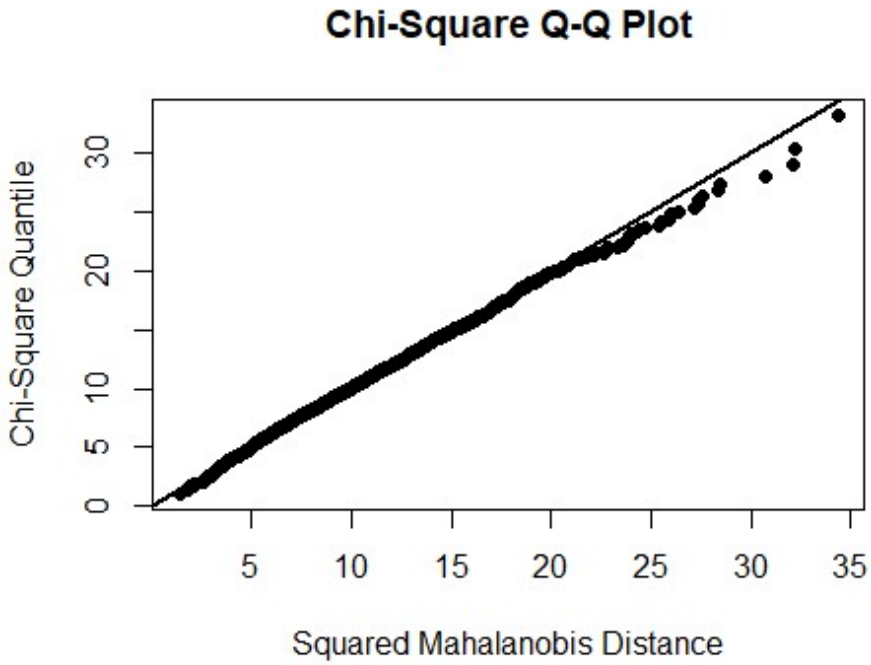
```
##     Sulfate        Conductivity   Organic_carbon   Trihalomethanes
##  Min.   :231.7   Min.   :201.6   Min.   : 5.315   Min.   : 24.53
##  1st Qu.:309.0   1st Qu.:366.5   1st Qu.:12.231   1st Qu.: 56.12
##  Median :332.8   Median :421.4   Median :14.355   Median : 66.22
##  Mean   :333.7   Mean   :425.7   Mean   :14.427   Mean   : 66.41
##  3rd Qu.:358.3   3rd Qu.:481.9   3rd Qu.:16.787   3rd Qu.: 77.29
##  Max.   :433.6   Max.   :652.5   Max.   :23.604   Max.   :108.85
##    Turbidity       Potability
##  Min.   :1.873   Min.   :0.0000
##  1st Qu.:3.444   1st Qu.:0.0000
##  Median :3.969   Median :0.0000
##  Mean   :3.971   Mean   :0.3918
##  3rd Qu.:4.510   3rd Qu.:1.0000
##  Max.   :6.084   Max.   :1.0000
```

```
boxplot(scale(data_Tukey_outliers[,1:9]))
```



Veamos con los outliers eliminados por la distancia en desviaciones típicas de la media:

```
result <- mvn(data_sd_outliers, multivariatePlot = "qq", showOutliers =
TRUE)
```




```
result$multivariateNormality
```

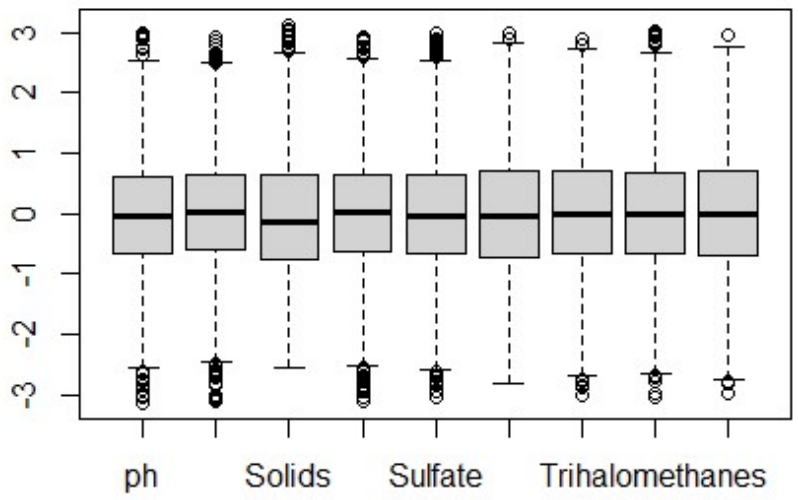
```
##              Test        Statistic              p value Result
## 1 Mardia Skewness 758.700075810924 2.25265316150777e-60     NO
## 2 Mardia Kurtosis 2.28194756983412   0.0224924358768734     NO
## 3             MVN             <NA>                 <NA>     NO
```

```
summary(data_sd_outliers)
```

```
##        ph            Hardness          Solids         Chloramines
##  Min.   : 2.377   Min.   : 98.45   Min.   :  320.9   Min.   : 2.398
##  1st Qu.: 6.106   1st Qu.:176.99   1st Qu.:15465.4   1st Qu.: 6.141
##  Median : 7.029   Median :197.12   Median :20564.4   Median : 7.137
##  Mean   : 7.088   Mean   :196.08   Mean   :21645.7   Mean   : 7.111
##  3rd Qu.: 8.026   3rd Qu.:215.94   3rd Qu.:26852.8   3rd Qu.: 8.087
##  Max.   :11.569   Max.   :287.98   Max.   :47581.0   Max.   :11.543
##     Sulfate       Conductivity   Organic_carbon   Trihalomethanes
##  Min.   :214.5   Min.   :201.6   Min.   : 4.467   Min.   : 17.92
##  1st Qu.:308.4   1st Qu.:366.6   1st Qu.:12.189   1st Qu.: 55.94
##  Median :332.8   Median :422.4   Median :14.351   Median : 66.33
##  Mean   :334.0   Mean   :425.9   Mean   :14.391   Mean   : 66.37
##  3rd Qu.:359.3   3rd Qu.:481.9   3rd Qu.:16.721   3rd Qu.: 77.21
##  Max.   :450.9   Max.   :666.7   Max.   :23.918   Max.   :114.21
##    Turbidity       Potability
##  Min.   :1.681   Min.   :0.0000
##  1st Qu.:3.445   1st Qu.:0.0000
##  Median :3.969   Median :0.0000
```

```
##  Mean   :3.968   Mean   :0.3959
##  3rd Qu.:4.507   3rd Qu.:1.0000
##  Max.   :6.227   Max.   :1.0000
```

```
boxplot(scale(data_sd_outliers[,1:9]))
```



Vemos una clara diferencia entre ambos métodos, debido a la cantidad de datos que eliminan, como podemos ver:

```
cat("data_sd_outliers: ",nrow(data_sd_outliers), "\n")
```

```
## data_sd_outliers:  1930
```

```
cat("data_Tukey_outliers: ",nrow(data_Tukey_outliers))
```

```
## data_Tukey_outliers:  1789
```

Por tanto, trabajaremos con la hipótesis de no normalidad de nuestros datos.

Preparación de los conjuntos de datos para imputación de valores.

Para la imputación de los datos, y teniendo en cuenta el estudio anteriror de la distribución, no podemos emplear métodos que asuman la normalidad de nuestros datos, como puede ser el caso de “AMELIA”.

Para tratar de evaluar la precisión de la imputación haremos un estudio previo de los diferentes métodos y su error cuadrático medio respecto a valores conocidos. Para

eso, vamos a imputar NAs en complete_data para aplicar distintos modelos de imputación de valores y poder evaluarlos y compararlos entre sí.

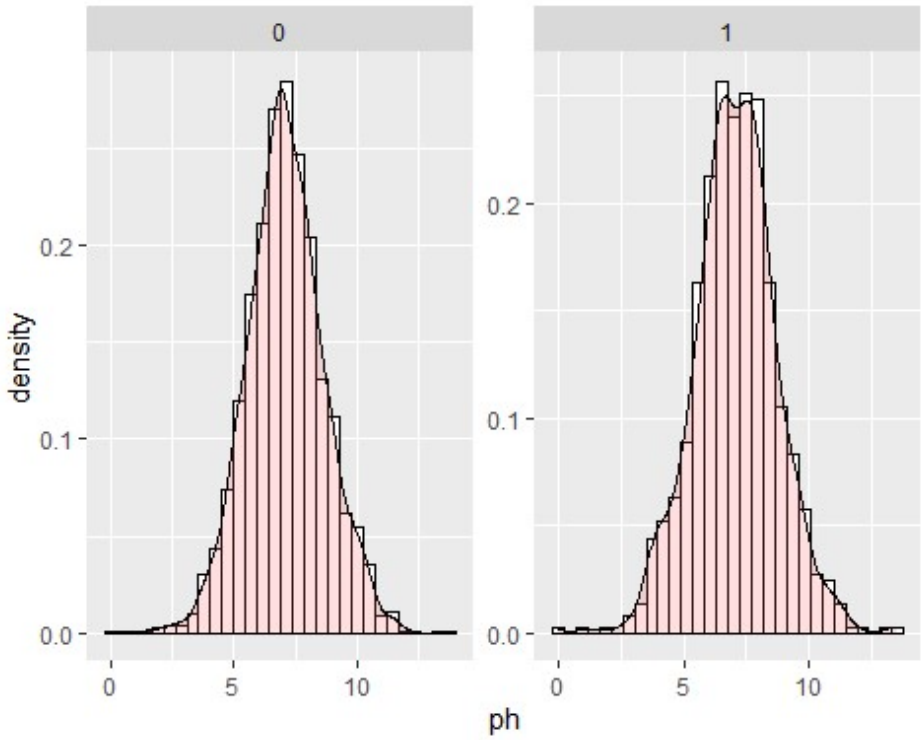
Pero antes de nada hemos considerado comprobar que la distribución de los datos en los que existen valores nulos es similar al resto de registros, para esto introduciremos un nuevo atributo binario en nuestro conjunto de datos (concretamente crearemos uno nuevo, "data_test") y compararemos la distribución atributo a atributo:

```
library(plyr)
library(ggplot2)

data_test <- data
data_test$na_count <- apply(data, 1, function(x) sum(is.na(x)))
data_test$na_count[data_test$na_count>0] <- 1

for (i in names(data)){
  plt <- ggplot(data_test, aes_string(x=i)) +
         geom_histogram(aes(y=..density..), colour="black",
fill="white")+
         geom_density(alpha=.2, fill="#FF6666") + facet_wrap(~na_count,
scales="free")
  print(plt)
}
```

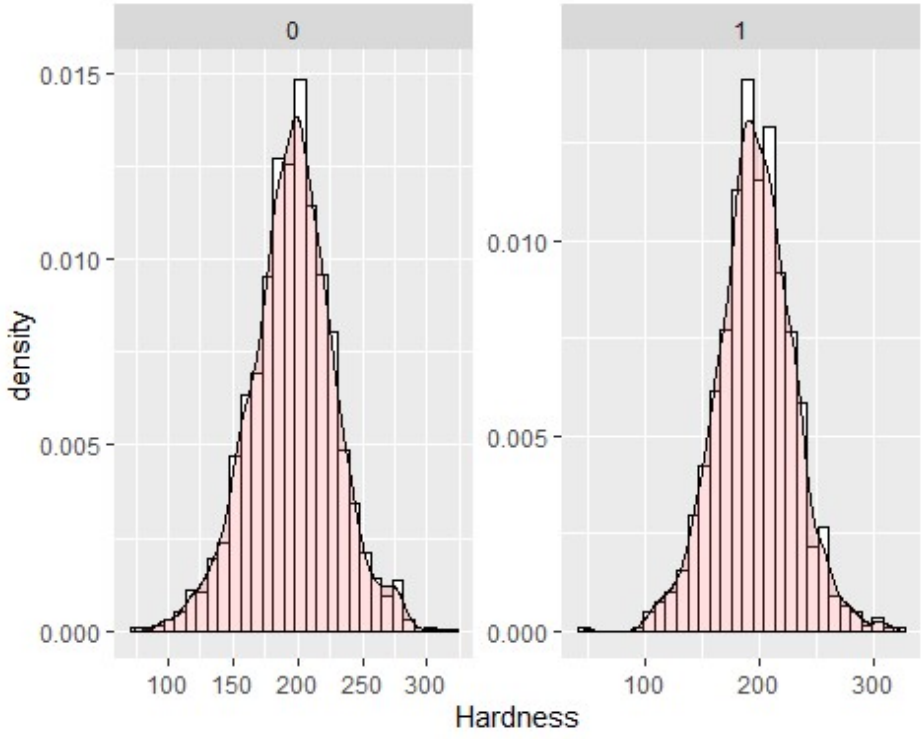
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 491 rows containing non-finite values (stat_bin).
```

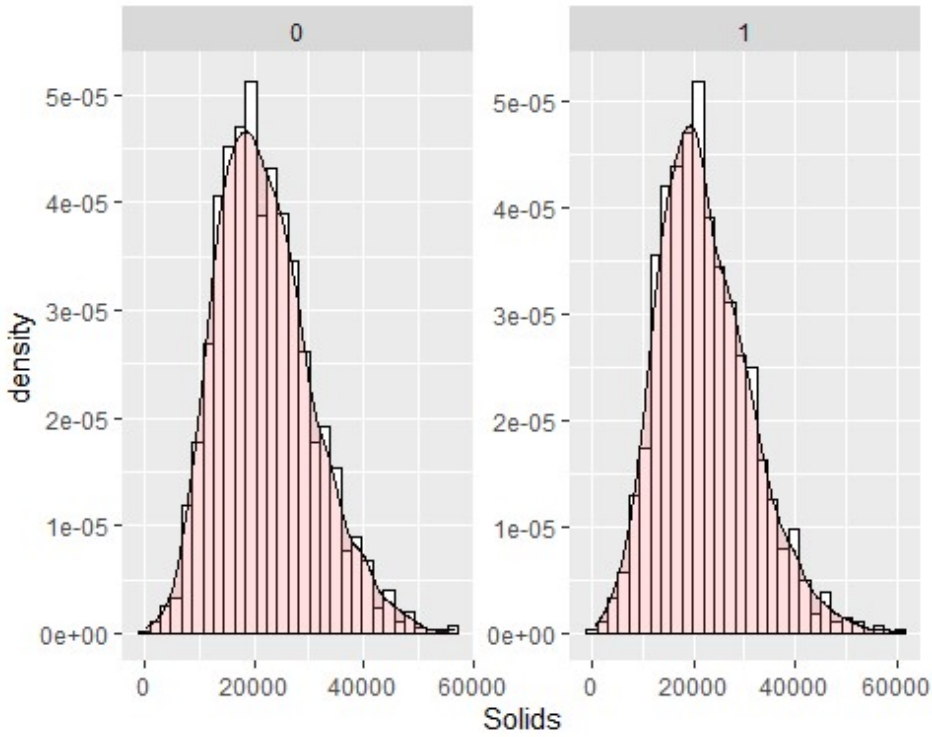
```
## Warning: Removed 491 rows containing non-finite values (stat_density).
```



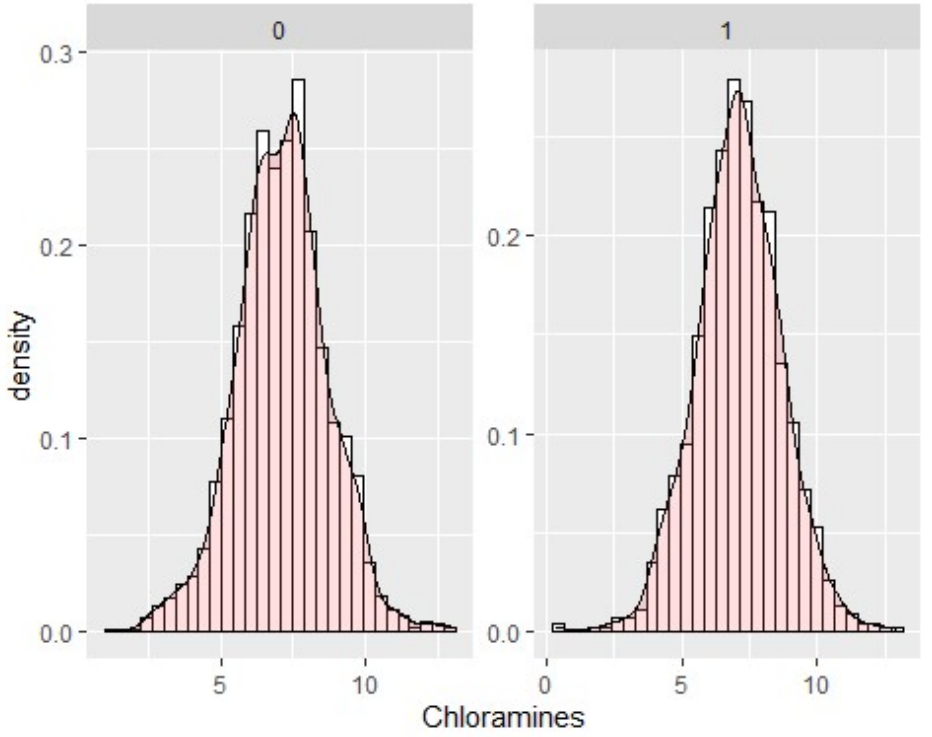

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



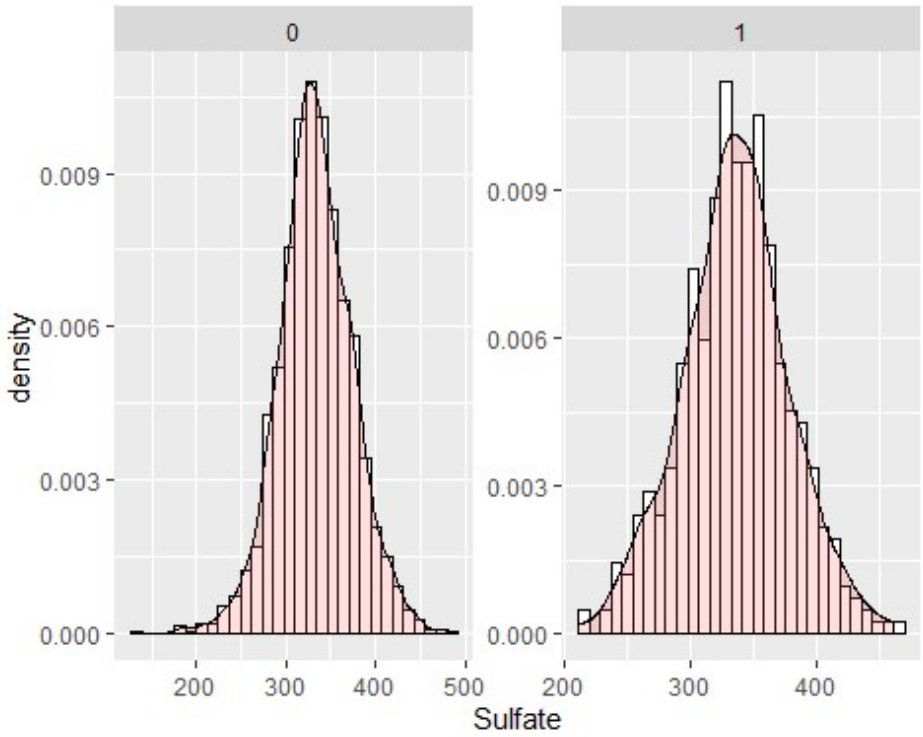

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



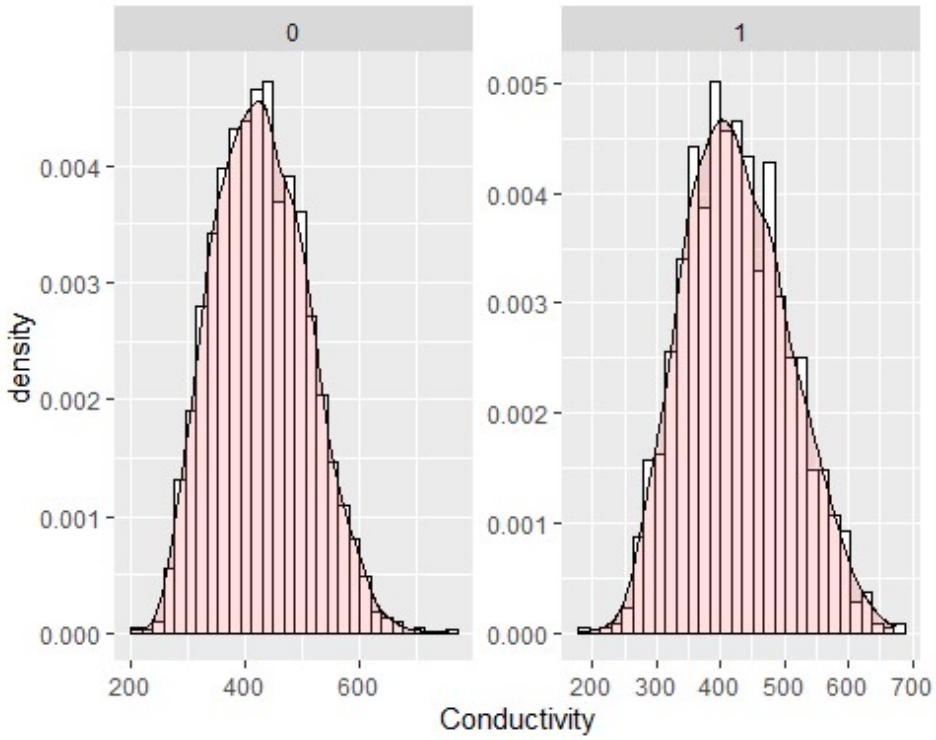

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 781 rows containing non-finite values (stat_bin).
```

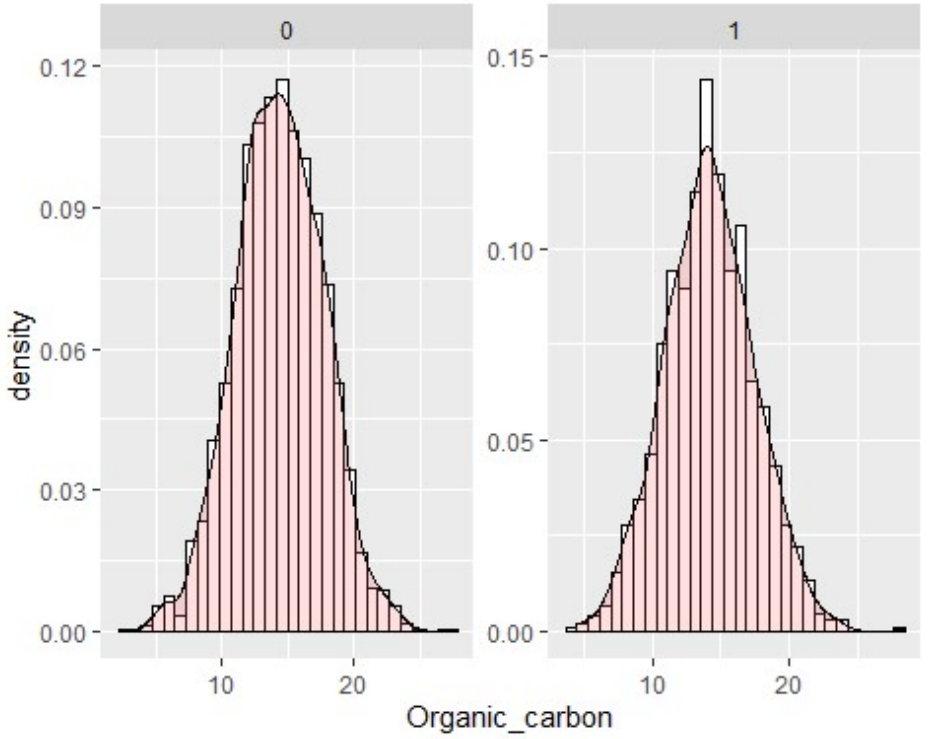
```
## Warning: Removed 781 rows containing non-finite values (stat_density).
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



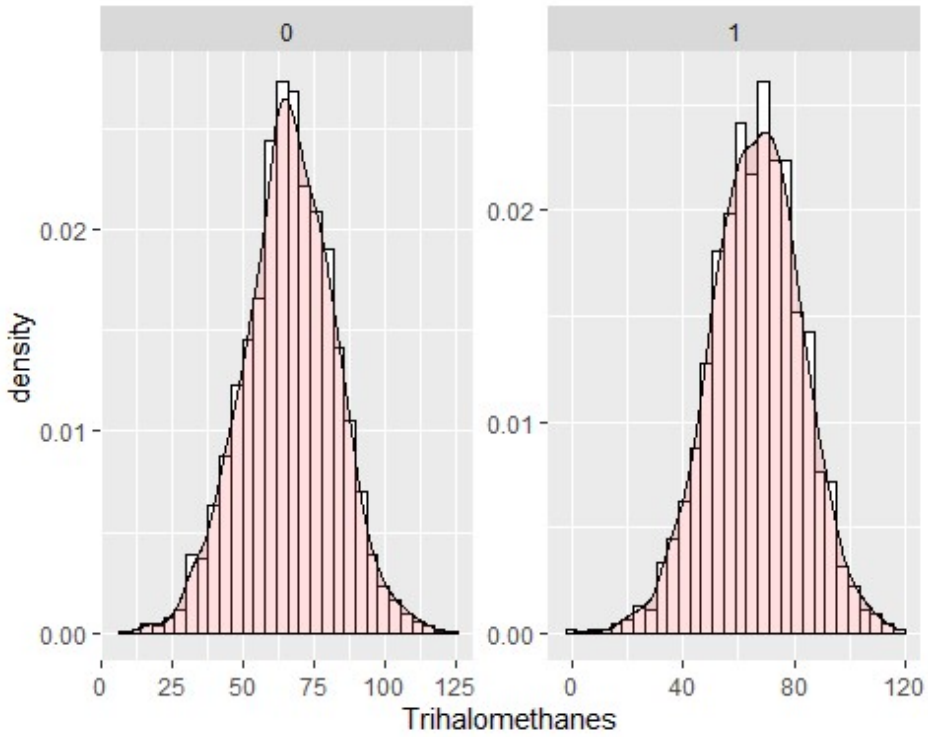

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



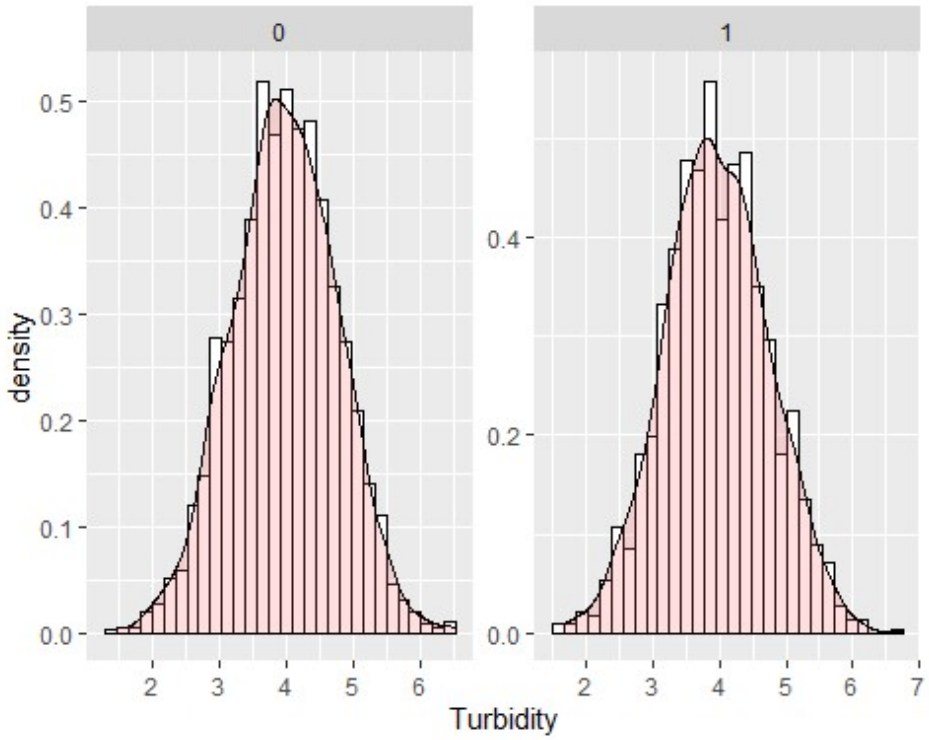

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 162 rows containing non-finite values (stat_bin).
```

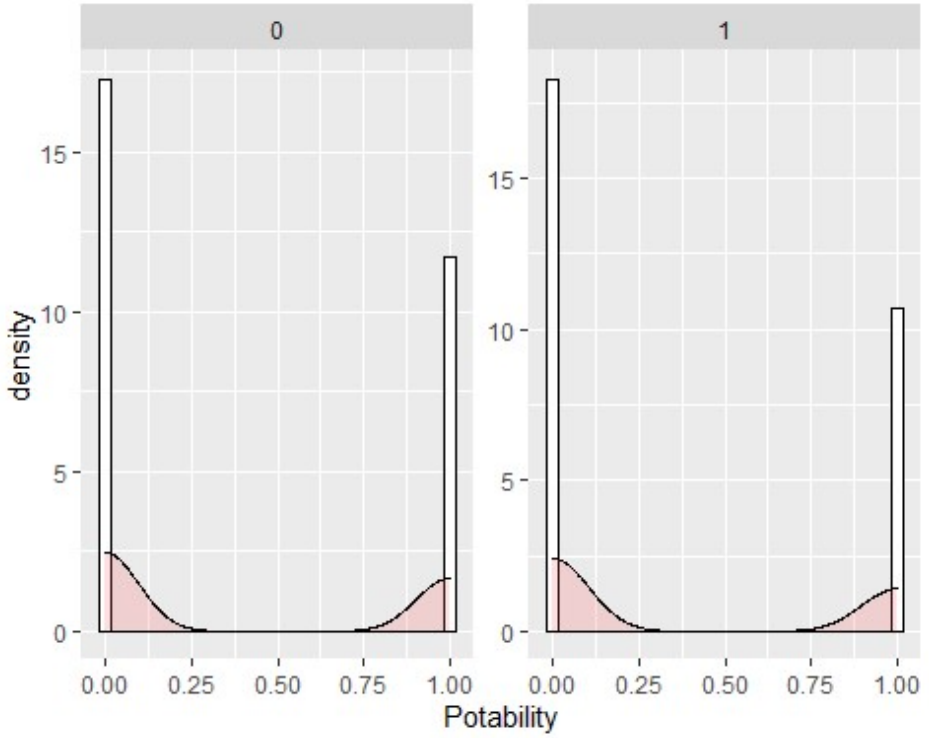
```
## Warning: Removed 162 rows containing non-finite values (stat_density).
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Visualmente parecen similares, pero compararemos las funciones summary() para estar un poco más seguros, nos interesa principalmente la similitud entre los atributos que no presentan valores nulos, ya que muchas de las imputaciones se harán en función de estos.

```
summary(data_test[data_test$na_count==0,])
```

```
##        ph              Hardness          Solids         Chloramines
##  Min.   : 0.2275   Min.   : 73.49   Min.   :  320.9   Min.   : 1.391
##  1st Qu.: 6.0897   1st Qu.:176.74   1st Qu.:15615.7   1st Qu.: 6.139
##  Median : 7.0273   Median :197.19   Median :20933.5   Median : 7.144
##  Mean   : 7.0860   Mean   :195.97   Mean   :21917.4   Mean   : 7.134
##  3rd Qu.: 8.0530   3rd Qu.:216.44   3rd Qu.:27182.6   3rd Qu.: 8.110
##  Max.   :14.0000   Max.   :317.34   Max.   :56488.7   Max.   :13.127
##     Sulfate       Conductivity   Organic_carbon  Trihalomethanes
##  Min.   :129.0   Min.   :201.6   Min.   : 2.20   Min.   :  8.577
##  1st Qu.:307.6   1st Qu.:366.7   1st Qu.:12.12   1st Qu.: 55.953
##  Median :332.2   Median :423.5   Median :14.32   Median : 66.542
##  Mean   :333.2   Mean   :426.5   Mean   :14.36   Mean   : 66.401
##  3rd Qu.:359.3   3rd Qu.:482.4   3rd Qu.:16.68   3rd Qu.: 77.292
##  Max.   :481.0   Max.   :753.3   Max.   :27.01   Max.   :124.000
##    Turbidity       Potability        na_count
##  Min.   :1.450   Min.   :0.0000   Min.   :0
##  1st Qu.:3.443   1st Qu.:0.0000   1st Qu.:0
##  Median :3.968   Median :0.0000   Median :0
##  Mean   :3.970   Mean   :0.4033   Mean   :0
##  3rd Qu.:4.514   3rd Qu.:1.0000   3rd Qu.:0
##  Max.   :6.495   Max.   :1.0000   Max.   :0
```

```
summary(data_test[data_test$na_count==1,])
```

```
##        ph              Hardness          Solids         Chloramines
##  Min.   : 0.000   Min.   : 47.43   Min.   :  728.8   Min.   : 0.352
##  1st Qu.: 6.097   1st Qu.:177.20   1st Qu.:15736.9   1st Qu.: 6.113
##  Median : 7.064   Median :196.53   Median :20922.2   Median : 7.105
##  Mean   : 7.067   Mean   :197.01   Mean   :22167.7   Mean   : 7.103
##  3rd Qu.: 8.085   3rd Qu.:217.94   3rd Qu.:27420.2   3rd Qu.: 8.121
##  Max.   :13.541   Max.   :323.12   Max.   :61227.2   Max.   :12.912
##  NA's   :491
##     Sulfate       Conductivity   Organic_carbon   Trihalomethanes
##  Min.   :211.9   Min.   :181.5   Min.   : 4.473   Min.   :  0.738
##  1st Qu.:309.2   1st Qu.:365.1   1st Qu.:11.877   1st Qu.: 55.641
##  Median :336.0   Median :420.1   Median :14.066   Median : 66.912
##  Mean   :336.1   Mean   :425.7   Mean   :14.169   Mean   : 66.388
##  3rd Qu.:360.8   3rd Qu.:480.3   3rd Qu.:16.335   3rd Qu.: 77.506
##  Max.   :462.5   Max.   :674.4   Max.   :28.300   Max.   :118.357
##  NA's   :781                                      NA's   :162
##    Turbidity       Potability        na_count
##  Min.   :1.642   Min.   :0.0000   Min.   :1
##  1st Qu.:3.438   1st Qu.:0.0000   1st Qu.:1
##  Median :3.930   Median :0.0000   Median :1
```

```
##  Mean   :3.962   Mean   :0.3692   Mean   :1
##  3rd Qu.:4.478   3rd Qu.:1.0000   3rd Qu.:1
##  Max.   :6.739   Max.   :1.0000   Max.   :1
##
```

Una vez comprobado podemos continuar. Queremos imputar NAs en las siguientes columnas: ph, Sulfate y Trihalomethanes. Las demás las dejaremos tal y como están, y por supuesto queremos obtener una distribución de valores nulos lo más semejante al conjunto original, veamos si lo conseguimos, trataremos de conseguir los mismos porrcentajes:

```
apply(data, 2, function(col)sum(is.na(col))/length(col))
```

```
##              ph        Hardness          Solids     Chloramines
Sulfate
##      0.14987790      0.00000000      0.00000000      0.00000000
0.23840049
##    Conductivity  Organic_carbon Trihalomethanes       Turbidity
Potability
##      0.00000000      0.00000000      0.04945055      0.00000000
0.00000000
```

```
library(missForest)
```

```
## Loading required package: randomForest
```

```
## randomForest 4.6-14
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
## Loading required package: foreach
```

```
## Loading required package: itertools
```

```
## Loading required package: iterators
```

```
##
## Attaching package: 'missForest'
```

```
## The following object is masked from 'package:VIM':
##
##     nrmse
```

```r
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------
tidyverse 1.3.1 --

## v tibble  3.1.2     v purrr   0.3.4
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   1.4.0     v forcats 0.5.1

## -- Conflicts ------------------------------------------
tidyverse_conflicts() --
## x purrr::accumulate()     masks foreach::accumulate()
## x dplyr::arrange()        masks plyr::arrange()
## x randomForest::combine() masks dplyr::combine()
## x purrr::compact()        masks plyr::compact()
## x dplyr::count()          masks plyr::count()
## x dplyr::failwith()       masks plyr::failwith()
## x mice::filter()          masks dplyr::filter(), stats::filter()
## x dplyr::id()             masks plyr::id()
## x dplyr::lag()            masks stats::lag()
## x randomForest::margin()  masks ggplot2::margin()
## x dplyr::mutate()         masks plyr::mutate()
## x dplyr::rename()         masks plyr::rename()
## x MASS::select()          masks dplyr::select()
## x dplyr::src()            masks Hmisc::src()
## x dplyr::summarise()      masks plyr::summarise()
## x dplyr::summarize()      masks plyr::summarize(), Hmisc::summarize()
## x purrr::when()           masks foreach::when()
```

```r
set.seed(564165)
complete_data <- data[complete.cases(data),]
head(complete_data)
```

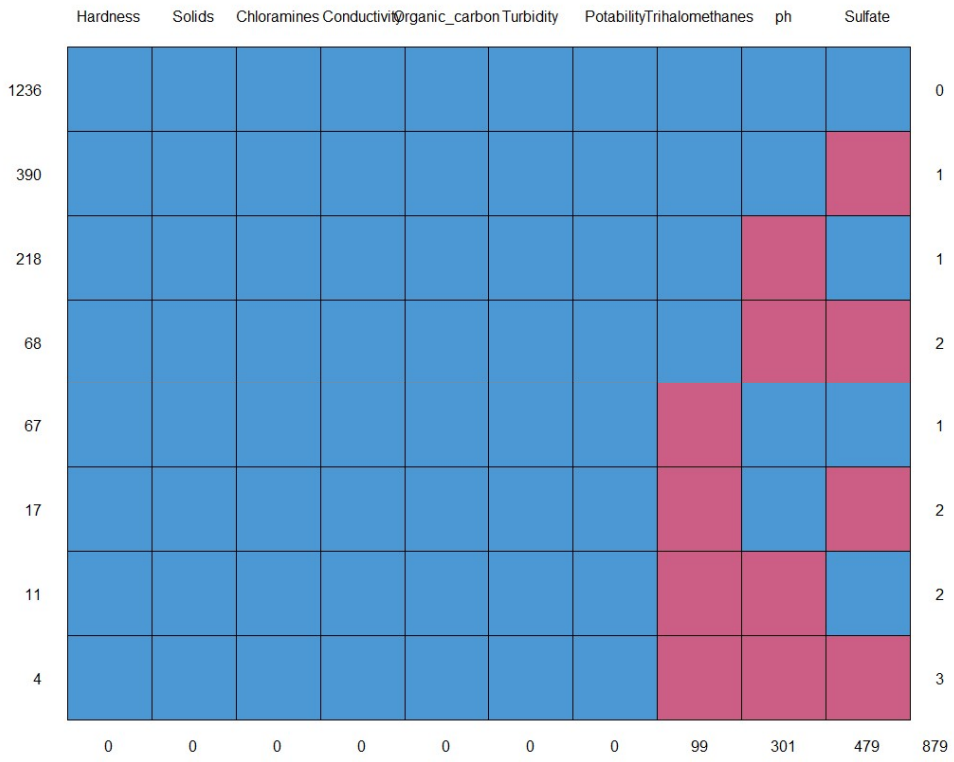
```
##           ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 4   8.316766 214.3734 22018.42    8.059332 356.8861     363.2665
18.436524
## 5   9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6   5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
## 7  10.223862 248.0717 28749.72    7.513408 393.6634     283.6516
13.789695
## 8   8.635849 203.3615 13672.09    4.563009 303.3098     474.6076
12.363817
## 10 11.180284 227.2315 25484.51    9.077200 404.0416     563.8855
17.927806
##    Trihalomethanes Turbidity Potability
## 4        100.34167  4.628771          0
## 5         31.99799  4.075075          0
## 6         54.91786  2.559708          0
```

```
## 7         84.60356  2.672989          0
## 8         62.79831  4.401425          0
## 10        71.97660  4.370562          0

complete_data_ph <- complete_data[,c("ph")]
complete_data_Sulfate <- complete_data[,c("Sulfate")]
complete_data_Triha <- complete_data[,c("Trihalomethanes")]

complete_data_noNAcols <- complete_data[,c("Hardness", "Solids",
"Chloramines", "Conductivity", "Organic_carbon", "Turbidity",
"Potability")]

complete_data_ph <- prodNA(as.data.frame(complete_data_ph), noNA =
0.14987790)
complete_data_Sulfate <- prodNA(as.data.frame(complete_data_Sulfate),
noNA = 0.23840049)
complete_data_Triha <- prodNA(as.data.frame(complete_data_Triha), noNA =
0.04945055)

complete_data_NAcols <- cbind(complete_data_ph, complete_data_Sulfate)
complete_data_NAcols <- cbind(complete_data_NAcols, complete_data_Triha)

complete_data_NAs <- cbind(complete_data_NAcols, complete_data_noNAcols)
complete_data_NAs <- complete_data_NAs[, c(1, 4, 5, 6, 2, 7, 8, 3, 9,
10)]

names(complete_data_NAs)[names(complete_data_NAs) == "complete_data_ph"]
<- "ph"
names(complete_data_NAs)[names(complete_data_NAs) ==
"complete_data_Sulfate"] <- "Sulfate"
names(complete_data_NAs)[names(complete_data_NAs) ==
"complete_data_Triha"] <- "Trihalomethanes"

head(complete_data_NAs)

##           ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 4         NA 214.3734 22018.42    8.059332       NA     363.2665
18.436524
## 5   9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6   5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
## 7  10.223862 248.0717 28749.72    7.513408 393.6634     283.6516
13.789695
## 8   8.635849 203.3615 13672.09    4.563009       NA     474.6076
12.363817
## 10        NA 227.2315 25484.51    9.077200 404.0416     563.8855
17.927806
##    Trihalomethanes Turbidity Potability
```

```
## 4         100.34167  4.628771          0
## 5          31.99799  4.075075          0
## 6          54.91786  2.559708          0
## 7          84.60356  2.672989          0
## 8          62.79831  4.401425          0
## 10         71.97660  4.370562          0
```

Utilizaremos la función md.pattern, visualmente podremos comprobar si el patrón es similar al del conjunto original de datos.

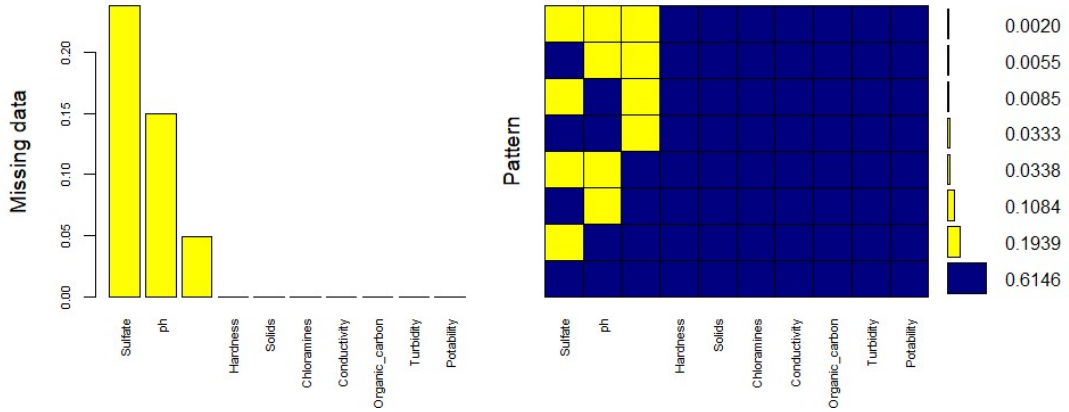
```
md.pattern(complete_data_NAs)
```




```
##      Hardness Solids Chloramines Conductivity Organic_carbon Turbidity
## 1236        1      1           1            1              1         1
## 390         1      1           1            1              1         1
## 218         1      1           1            1              1         1
## 68          1      1           1            1              1         1
```

```
## 67           1      1            1             1                1          1
## 17           1      1            1             1                1          1
## 11           1      1            1             1                1          1
## 4            1      1            1             1                1          1
##              0      0            0             0                0          0
##      Potability Trihalomethanes  ph Sulfate
## 1236          1                1   1       1   0
## 390           1                1   1       0   1
## 218           1                1   0       1   1
## 68            1                1   0       0   2
## 67            1                0   1       1   1
## 17            1                0   1       0   2
## 11            1                0   0       1   2
## 4             1                0   0       0   3
##               0               99 301     479 879
```

```
library(VIM)
library(mice)
set.seed(100)
mice_plot <- aggr(complete_data_NAs, col=c('navyblue','yellow'),
             numbers=TRUE, sortVars=TRUE,
             labels=names(complete_data_NAs), cex.axis=.7,
             gap=3, ylab=c("Missing data","Pattern"))
```




```
##
##  Variables sorted by number of missings:
##         Variable      Count
##          Sulfate 0.23818996
##               ph 0.14967678
##  Trihalomethanes 0.04922924
##         Hardness 0.00000000
##           Solids 0.00000000
##      Chloramines 0.00000000
##     Conductivity 0.00000000
##   Organic_carbon 0.00000000
##        Turbidity 0.00000000
##       Potability 0.00000000
```

De esta forma podemos introducir NAs en la misma proporción que en el conjunto original.

Aplicamos MICE

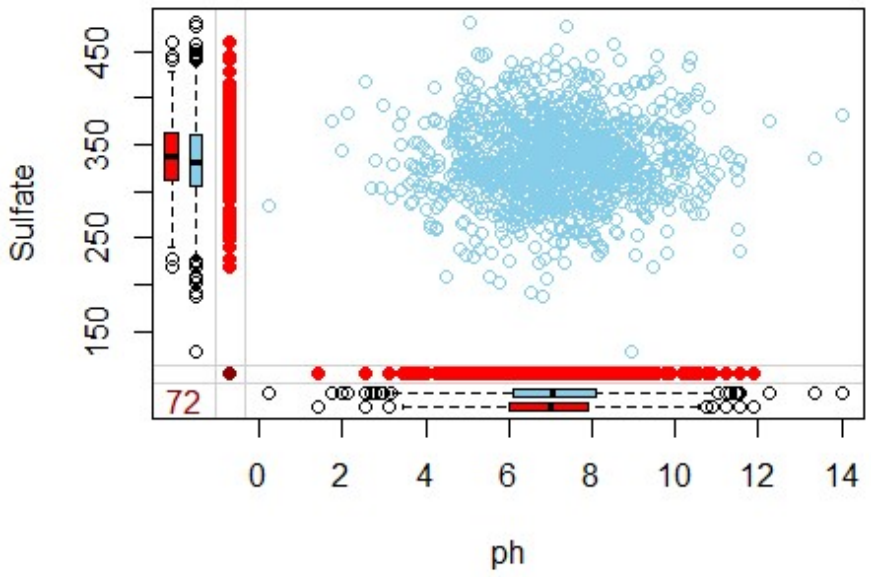
Empezaremos con la librería "MICE" para multivariate imputation. Emplearemos el método predictive mean matching, un método de imputación semiparamétrico. Sus principales ventajas son que los valores imputados coinciden con alguno de los valores observados en la misma variable y que puede preservar relaciones no lineales incluso si la parte estructural del modelo de imputación es incorrecta.

Es un buen método de imputación en general. Las funciones mice.impute.norm() y mice.impute.norm.nob() no se adaptan a nuestros datos debido a la distribución. La función mice.impute.norm.predict() aplica una regresión lineal entre las variables, puede ser útil.

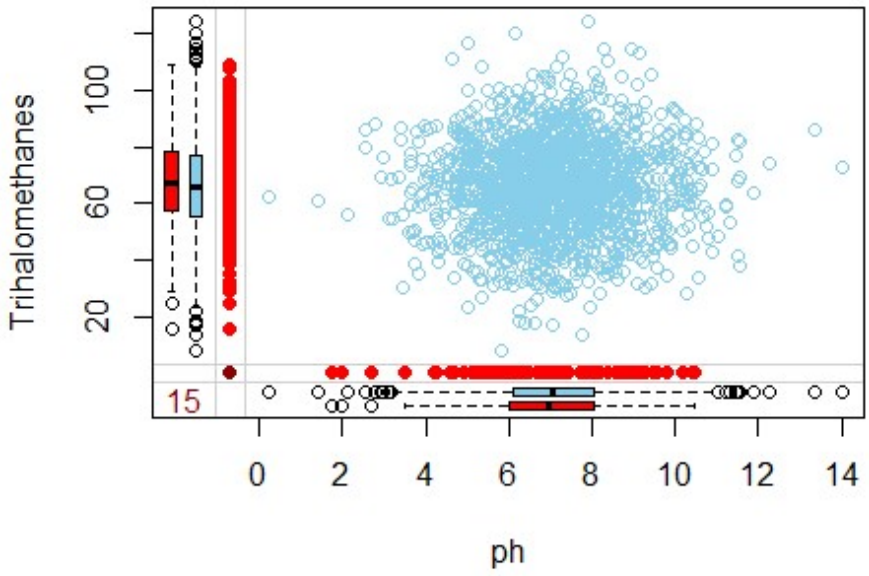
Para fracciones de información perdida γ=(0.1, 0.3, 0.5, 0.7, 0.9) necesitamos establecer m=(20, 20, 40, 100, >100) imputaciones, respectivamente.

Analizaremos la co-ocurrencoia de valores nulos, ha podido cambiar al ser aleatoria la asignación:

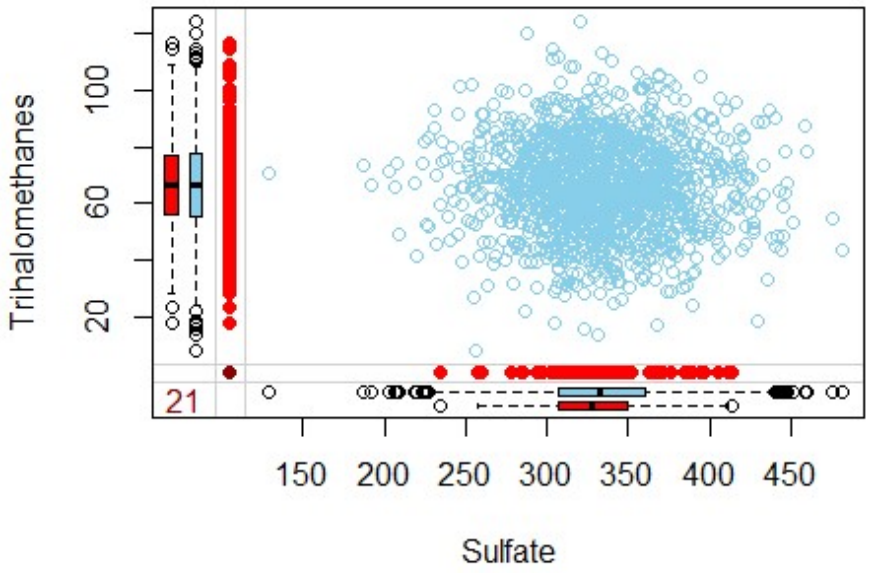
```
library(VIM)
marginplot(complete_data_NAs[c(1,5)])
```




```
marginplot(complete_data_NAs[c(1,8)])
```




```
marginplot(complete_data_NAs[c(5,8)])
```

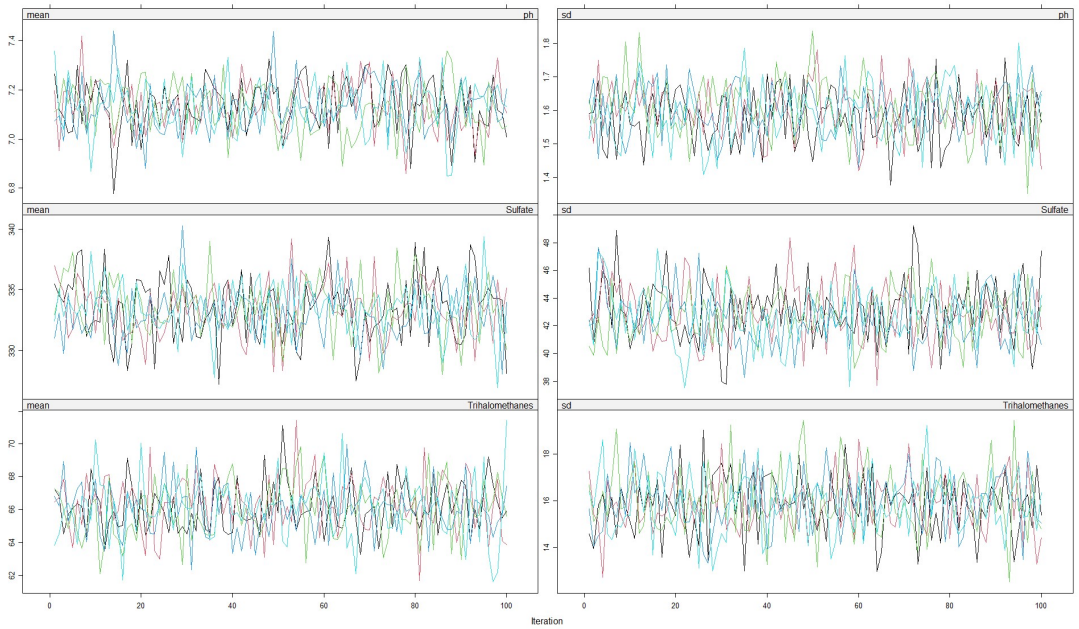


Vemos que es ligeramente distinta, pero al asignar los valores nulos de forma aleatoria es dificil de conseguir, e imputar los valores nulos manualmente sería un poco tedioso.

Lanzamos el método MICE de imputación de valores. Le asignaremos un valor de 100 iteraciones y 5 conjuntos de salida, se recomiendan valores de m mucho más elevados, sobretodo para el porcentaje de datos nulos presente, pero lo haremos así para reducir los tiempos de ejecución y presentar un trabajo didáctico, no tan orientado al rendimiento.

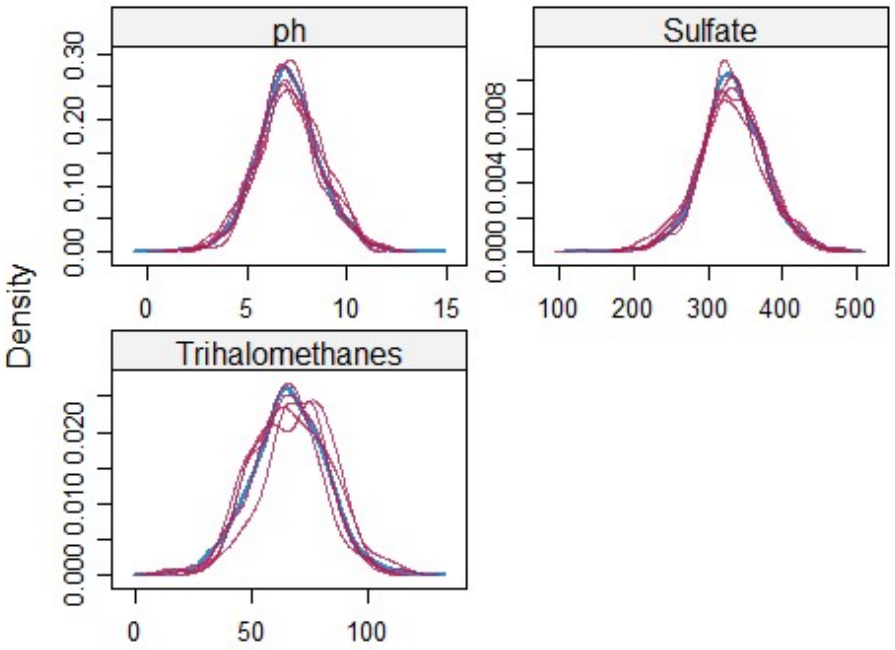
Mediante la función plot() podemos visualizar como varia la asignación por iteración:

```
plot(complete_data_NAs_MICE)
```



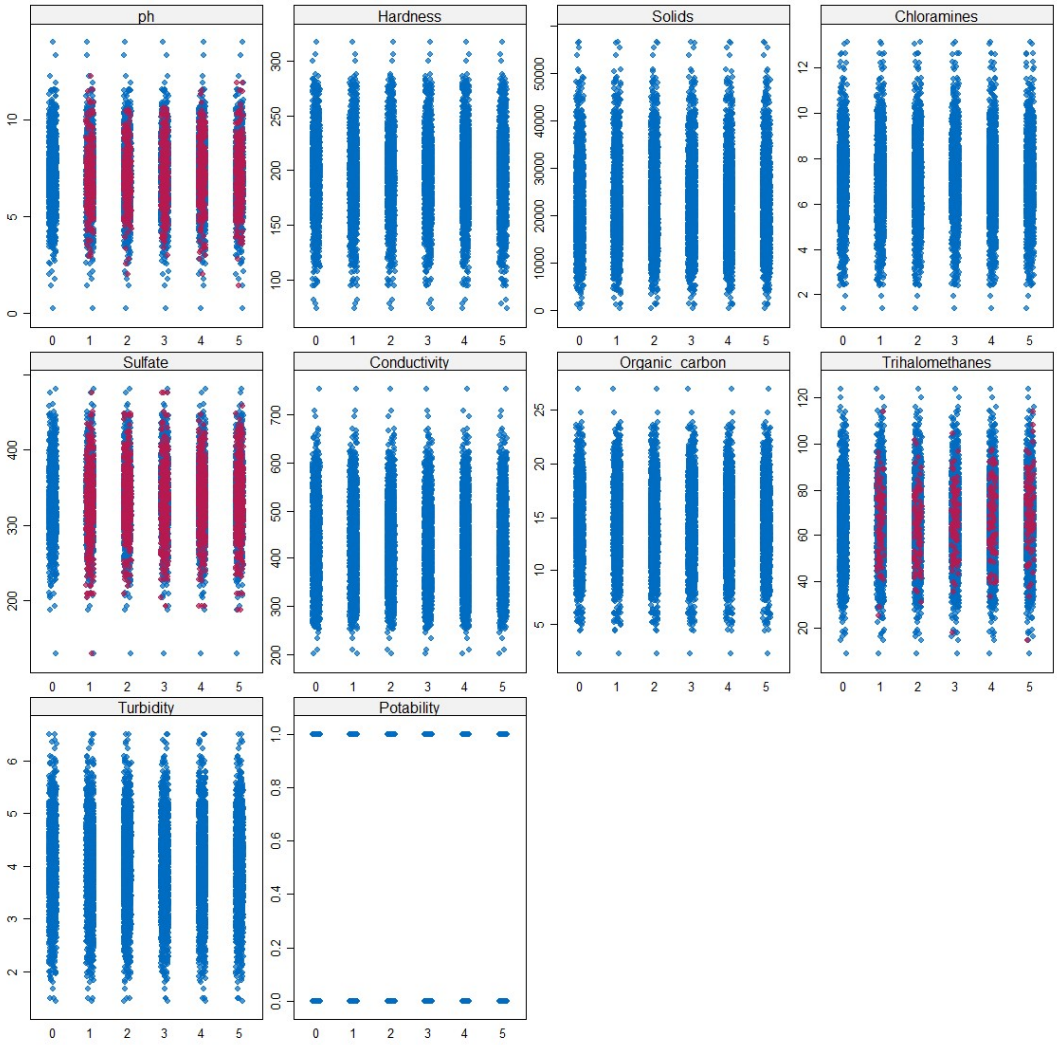
No son las fluctuaciones ideales, veamos los gráficos de densidad:

```
densityplot(complete_data_NAs_MICE)
```



Y la diferencia entre originales y asignados:

```
stripplot(complete_data_NAs_MICE, pch = 20, cex = 1.2)
```



También veremos la comparativa en el resultado de la función summary() entre los datos originales y todos los conjuntos generados(5):

```
summary(data)
```

```
##        ph              Hardness        Solids         Chloramines    
##  Min.   : 0.000   Min.   : 47.43   Min.   :  320.9   Min.   : 0.352  
##  1st Qu.: 6.093   1st Qu.:176.85   1st Qu.:15666.7   1st Qu.: 6.127  
##  Median : 7.037   Median :196.97   Median :20927.8   Median : 7.130  
##  Mean   : 7.081   Mean   :196.37   Mean   :22014.1   Mean   : 7.122  
##  3rd Qu.: 8.062   3rd Qu.:216.67   3rd Qu.:27332.8   3rd Qu.: 8.115  
##  Max.   :14.000   Max.   :323.12   Max.   :61227.2   Max.   :13.127  
##  NA's   :491                                                        
##     Sulfate       Conductivity   Organic_carbon  Trihalomethanes  
##  Min.   :129.0   Min.   :181.5   Min.   : 2.20   Min.   :  0.738  
##  1st Qu.:307.7   1st Qu.:365.7   1st Qu.:12.07   1st Qu.: 55.845  
##  Median :333.1   Median :421.9   Median :14.22   Median : 66.622
```

```
##  Mean   :333.8   Mean   :426.2   Mean   :14.28   Mean   : 66.396
##  3rd Qu.:360.0   3rd Qu.:481.8   3rd Qu.:16.56   3rd Qu.: 77.337
##  Max.   :481.0   Max.   :753.3   Max.   :28.30   Max.   :124.000
##  NA's   :781                                     NA's   :162
##    Turbidity       Potability
##  Min.   :1.450   Min.   :0.0000
##  1st Qu.:3.440   1st Qu.:0.0000
##  Median :3.955   Median :0.0000
##  Mean   :3.967   Mean   :0.3901
##  3rd Qu.:4.500   3rd Qu.:1.0000
##  Max.   :6.739   Max.   :1.0000
##
```

```
summary(mice::complete(complete_data_NAs_MICE, "long"))
```

```
##       .imp        .id             ph             Hardness
##  Min.   :1   Min.   :   1   Min.   : 0.2275   Min.   : 73.49
##  1st Qu.:2   1st Qu.: 503   1st Qu.: 6.1025   1st Qu.:176.74
##  Median :3   Median :1006   Median : 7.0381   Median :197.19
##  Mean   :3   Mean   :1006   Mean   : 7.0900   Mean   :195.97
##  3rd Qu.:4   3rd Qu.:1509   3rd Qu.: 8.0510   3rd Qu.:216.45
##  Max.   :5   Max.   :2011   Max.   :14.0000   Max.   :317.34
##      Solids         Chloramines        Sulfate       Conductivity
##  Min.   :  320.9   Min.   : 1.391   Min.   :129.0   Min.   :201.6
##  1st Qu.:15613.2   1st Qu.: 6.138   1st Qu.:307.1   1st Qu.:366.6
##  Median :20933.5   Median : 7.144   Median :332.7   Median :423.5
##  Mean   :21917.4   Mean   : 7.134   Mean   :333.0   Mean   :426.5
##  3rd Qu.:27192.3   3rd Qu.: 8.110   3rd Qu.:360.4   3rd Qu.:482.5
##  Max.   :56488.7   Max.   :13.127   Max.   :481.0   Max.   :753.3
##  Organic_carbon  Trihalomethanes     Turbidity       Potability
##  Min.   : 2.20   Min.   :  8.577   Min.   :1.450   Min.   :0.0000
##  1st Qu.:12.12   1st Qu.: 55.894   1st Qu.:3.443   1st Qu.:0.0000
##  Median :14.32   Median : 66.542   Median :3.968   Median :0.0000
##  Mean   :14.36   Mean   : 66.343   Mean   :3.970   Mean   :0.4033
##  3rd Qu.:16.68   3rd Qu.: 77.287   3rd Qu.:4.515   3rd Qu.:1.0000
##  Max.   :27.01   Max.   :124.000   Max.   :6.495   Max.   :1.0000
```

Una vez comprobada cierta similitud podemos pasar a tratar de mejorar un poco la imputación, para esto usaremos las funciones with() y pool() para hacer una regresión sobre una de las variables calculadas, en este caso la que más valores nulos tiene con diferencia, Sulfate, en la primera iteración incluiremos Trihalomethanes, ya que presenta un porcentaje pequeño de valores nulos, pero no ph:

```
MICE_test <- mice::complete(complete_data_NAs_MICE, "long")
fit <- with(complete_data_NAs_MICE, lm(Sulfate~
Hardness+Solids+Chloramines+Conductivity+Organic_carbon+Turbidity+Trihalo
methanes+Potability))
summary(mice::pool(fit))
```

```
##              term      estimate    std.error   statistic        df
p.value
```

```
## 1     (Intercept)  3.803690e+02 1.388601e+01 27.39224270  49.02522
0.000000e+00
## 2        Hardness -1.370754e-01 3.969809e-02 -3.45294736  16.60291
3.127261e-03
## 3          Solids -7.341511e-04 1.372353e-04 -5.34957708  26.28253
1.294978e-05
## 4     Chloramines  3.521150e-01 7.601914e-01  0.46319262  23.72700
6.474469e-01
## 5    Conductivity -4.566202e-03 1.418302e-02 -0.32194855  32.75251
7.495376e-01
## 6  Organic_carbon  4.486006e-01 3.073190e-01  1.45972284 117.06794
1.470445e-01
## 7       Turbidity -4.413580e-01 1.275387e+00 -0.34605819 201.91712
7.296593e-01
## 8 Trihalomethanes -1.457387e-01 6.185209e-02 -2.35624531 229.10427
1.930411e-02
## 9      Potability  1.215835e-01 2.313756e+00  0.05254813  35.09533
9.583899e-01
```

Tratamos de buscar los atributos con menor p-value, eliminamos los 3 con mayor valor e iteramos un par de veces:

```
imp_2 <- mice(complete_data_NAs[,-c(4,6,9,10)],m=5, maxit = 100,
method="pmm",seed=245435, print=FALSE)
fit_2 <- with(imp_2, lm(Sulfate~
Hardness+Solids+Organic_carbon+Trihalomethanes))
summary(mice::pool(fit_2))
```

```
##              term      estimate    std.error statistic       df
p.value
## 1     (Intercept)  3.792502e+02 1.098304e+01 34.530531 22.62327
0.000000e+00
## 2        Hardness -1.474752e-01 3.184427e-02 -4.631137 92.41540
1.186468e-05
## 3          Solids -7.304476e-04 1.209384e-04 -6.039833 84.85432
3.954112e-08
## 4  Organic_carbon  4.779421e-01 3.341791e-01  1.430197 41.98074
1.600614e-01
## 5 Trihalomethanes -1.190829e-01 8.039689e-02 -1.481187 16.98986
1.568645e-01
```

```
imp_3 <- mice(complete_data_NAs[,-c(4,6,8,9,10)],m=5, maxit = 100,
method="pmm",seed=245435, print=FALSE)
fit_3 <- with(imp_3, lm(Sulfate~ Hardness+Solids+Organic_carbon))
summary(mice::pool(fit_3))
```

```
##             term      estimate    std.error statistic        df
p.value
## 1    (Intercept)  3.707125e+02 8.0727160741 45.921656 119.78584
0.000000e+00
## 2       Hardness -1.305495e-01 0.0300160282 -4.349325 263.97454
```

```
1.952373e-05
## 3         Solids -7.625101e-04 0.0001212029 -6.291189  75.26736
1.893303e-08
## 4 Organic_carbon  3.537520e-01 0.3009091725  1.175611 155.84847
2.415434e-01
```

Vemos los valores de lambda y fmi que representan “Fraction of Missing Information” como vemos en la documentación.

```
mice::pool(fit)

## Class: mipo    m = 5
##              term m      estimate         ubar            b
t dfcom
## 1     (Intercept) 5  3.803690e+02 1.386904e+02 4.510916e+01
1.928214e+02  2002
## 2        Hardness 5 -1.370754e-01 8.086907e-04 6.393730e-04 1.575938e-
03  2002
## 3          Solids 5 -7.341511e-04 1.156527e-08 6.056891e-09 1.883354e-
08  2002
## 4     Chloramines 5  3.521150e-01 3.429979e-01 1.957442e-01 5.778909e-
01  2002
## 5    Conductivity 5 -4.566202e-03 1.317441e-04 5.784490e-05 2.011580e-
04  2002
## 6  Organic_carbon 5  4.486006e-01 7.762034e-02 1.402054e-02 9.444499e-
02  2002
## 7       Turbidity 5 -4.413580e-01 1.411399e+00 1.793434e-01
1.626611e+00  2002
## 8 Trihalomethanes 5 -1.457387e-01 3.354357e-03 3.927705e-04 3.825681e-
03  2002
## 9      Potability 5  1.215835e-01 3.570060e+00 1.486171e+00
5.353465e+00  2002
##          df       riv    lambda       fmi
## 1  49.02522 0.3903010 0.2807313 0.3083821
## 2  16.60291 0.9487528 0.4868513 0.5392056
## 3  26.28253 0.6284564 0.3859216 0.4278632
## 4  23.72700 0.6848236 0.4064661 0.4508806
## 5  32.75251 0.5268840 0.3450714 0.3817082
## 6 117.06794 0.2167557 0.1781423 0.1918322
## 7 201.91712 0.1524814 0.1323070 0.1407757
## 8 229.10427 0.1405112 0.1232002 0.1307554
## 9  35.09533 0.4995447 0.3331309 0.3681415

mice::pool(fit_2)

## Class: mipo    m = 5
##              term m      estimate         ubar            b
t dfcom
## 1     (Intercept) 5  3.792502e+02 7.039802e+01 4.185763e+01
1.206272e+02  2006
## 2        Hardness 5 -1.474752e-01 8.092740e-04 1.706532e-04 1.014058e-
```

```
03  2006
## 3          Solids 5 -7.304476e-04 1.153693e-08 2.574299e-09 1.462609e-
08  2006
## 4  Organic_carbon 5  4.779421e-01 7.772664e-02 2.829086e-02 1.116757e-
01  2006
## 5 Trihalomethanes 5 -1.190829e-01 3.353122e-03 2.592115e-03 6.463661e-
03  2006
##         df       riv    lambda       fmi
## 1 22.62327 0.7135024 0.4164000 0.4619523
## 2 92.41540 0.2530463 0.2019449 0.2186729
## 3 84.85432 0.2677626 0.2112088 0.2291656
## 4 41.98074 0.4367748 0.3039967 0.3349434
## 5 16.98986 0.9276544 0.4812348 0.5331376
```

```
mice::pool(fit_3)
```

```
## Class: mipo    m = 5
##             term m      estimate         ubar            b
t dfcom
## 1    (Intercept) 5  3.707125e+02 5.369982e+01 9.557440e+00
6.516874e+01  2007
## 2       Hardness 5 -1.305495e-01 7.986231e-04 8.528233e-05 9.009619e-
04  2007
## 3         Solids 5 -7.625101e-04 1.138665e-08 2.752906e-09 1.469013e-
08  2007
## 4 Organic_carbon 5  3.537520e-01 7.672163e-02 1.152058e-02 9.054633e-
02  2007
##          df       riv    lambda       fmi
## 1 119.78584 0.2135748 0.1759882 0.1894101
## 2 263.97454 0.1281440 0.1135884 0.1202288
## 3  75.26736 0.2901194 0.2248779 0.2446850
## 4 155.84847 0.1801929 0.1526809 0.1633492
```

Y como tenemos los valores reales podemos calcular el error medio, de cada uno de los 5 modelos propuestos y del conjunto:

```
library(Metrics)
a=0
for (i in 1:5) {
  predicted <- mice::complete(complete_data_NAs_MICE,
i)[is.na(complete_data_NAs$Sulfate),]$Sulfate
  actual <- complete_data[is.na(complete_data_NAs$Sulfate),]$Sulfate

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 60.4991
## 56.57836
## 58.91149
```

```
## 56.29042
## 58.98215
```

```
cat("mean RMSE: ",a/5)
```

```
## mean RMSE:  58.2523
```

```
a=0
for (i in 1:5) {
  predicted <- mice::complete(imp_2,
i)[is.na(complete_data_NAs$Sulfate),]$Sulfate
  actual <- complete_data[is.na(complete_data_NAs$Sulfate),]$Sulfate

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 55.98593
## 56.56648
## 56.08869
## 57.59553
## 58.41985
```

```
cat("mean RMSE: ",a/5)
```

```
## mean RMSE:  56.9313
```

```
a=0
for (i in 1:5) {
  predicted <- mice::complete(imp_3,
i)[is.na(complete_data_NAs$Sulfate),]$Sulfate
  actual <- complete_data[is.na(complete_data_NAs$Sulfate),]$Sulfate

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 55.46133
## 56.73941
## 54.0202
## 58.2194
## 54.70953
```

```
cat("mean RMSE: ",a/5)
```

```
## mean RMSE:  55.82997
```

Podemos comprobar para el atributo ph también:

```r
library(Metrics)
a=0
for (i in 1:5) {
  predicted <- mice::complete(complete_data_NAs_MICE,
i)[is.na(complete_data_NAs$ph),]$ph
  actual <- complete_data[is.na(complete_data_NAs$ph),]$ph

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 2.200999
## 2.166901
## 2.196755
## 2.246637
## 2.10456
```

```r
cat("mean RMSE: ",a/5)
```

```
## mean RMSE:  2.183171
```

```r
a=0
for (i in 1:5) {
  predicted <- mice::complete(imp_2, i)[is.na(complete_data_NAs$ph),]$ph
  actual <- complete_data[is.na(complete_data_NAs$ph),]$ph

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 2.134188
## 2.388057
## 2.111239
## 2.292533
## 2.128205
```

```r
cat("mean RMSE: ",a/5)
```

```
## mean RMSE:  2.210844
```

```r
a=0
for (i in 1:5) {
  predicted <- mice::complete(imp_3, i)[is.na(complete_data_NAs$ph),]$ph
  actual <- complete_data[is.na(complete_data_NAs$ph),]$ph

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 2.246288
## 2.343491
## 2.257015
## 2.299264
## 2.009479

cat("mean RMSE: ",a/5)

## mean RMSE:  2.231108
```

Vemos que las mejoras no son muy significativas dependiendo de los atributos que utilicemos para imputar los valores, en un estudio en profundidad podríamos hacerlo uno a uno, es decir, introduciendo los atributos a predecir uno a uno hasta conseguir un valor aceptable y luego añadir ese valor predicho en la predicción del siguiente atributo para ver si mejora, pero es un proceso iterativo que puede llevar bastante tiempo y para la práctica hemos decidido estudiar otros tipos de imputaciones también.

De momento nos quedamos con los errores medios para compararlos con otros métodos. Necesitamos el de Trihalometanes:

```
library(Metrics)
a=0
for (i in 1:5) {
  predicted <- mice::complete(complete_data_NAs_MICE,
i)[is.na(complete_data_NAs$Trihalomethanes),]$Trihalomethanes
  actual <-
complete_data[is.na(complete_data_NAs$Trihalomethanes),]$Trihalomethanes

  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}

## 22.73853
## 21.94045
## 21.7571
## 22.70659
## 24.35777

cat("mean RMSE: ",a/5,"\n")

## mean RMSE:  22.70009

a=0
for (i in 1:5) {
  predicted <- mice::complete(imp_2,
i)[is.na(complete_data_NAs$Trihalomethanes),]$Trihalomethanes
  actual <-
complete_data[is.na(complete_data_NAs$Trihalomethanes),]$Trihalomethanes
```

```
  cat(Metrics::rmse(actual, predicted), "\n")

  a= a + (Metrics::rmse(actual, predicted))
}
```

```
## 21.55522
## 22.91399
## 24.6577
## 21.54778
## 21.02015
```

```
cat("mean RMSE: ",a/5, "\n")
```

```
## mean RMSE:  22.33897
```

Creamos una función para ir guardando los resultados de RMSE de cada modelo para cada variable.

```
res <- "Resultado"
name <- "Model"
add_value <- function(result, new_name) {
  res <<- c(res, result)
  name <<- c(name, new_name)
}
add_value(2.183171, "result_MICE_ph")
add_value(55.8299, "result_MICE_Sulfate")
add_value(22.33897 , "result_MICE_Trihalomethanes")
```

MICE también permite crear nuestras propias funciones de imputación, puede ser una herramienta muy útil en algunos casos, teniendo un conocimiento previo de los datos y sabiendo como quieres imputar dichos valores, pero en este caso no hemos visto ninguna relación entre datos, ni conocemos en profundidad el tema como para definir un comportamiento.

Aplicamos missForest

El siguiente método será mediante la librería missForest, Como sugiere el nombre, es una implementación del algoritmo de bosque aleatorio. Es un método de imputación no paramétrico aplicable a varios tipos de variables.

Lanzamos el método missForest de imputación de valores. Destacar que estamos empleando métodos no paramétricos debido al estudio previo de distribución de nuestros datos. Un método no paramétrico no hace suposiciones explícitas sobre la forma funcional de f (cualquier función arbitraria). En su lugar, intenta estimar f de modo que pueda estar lo más cerca posible de los puntos de datos sin que parezca poco práctico.

El funcionamiento, en palabras simples, crea un modelo de bosque aleatorio para cada variable. Luego, usa el modelo para predecir los valores perdidos en la variable con la ayuda de los valores observados. Produce una estimación del error de imputación

OOB (fuera de bolsa). Además, proporciona un alto nivel de control sobre el proceso de imputación. Tiene opciones para devolver OOB por separado (para cada variable) en lugar de agregar toda la matriz de datos. Esto ayuda a observar más de cerca la precisión con la que el modelo ha imputado valores para cada variable. También es válido para variables categóricas, pero no es nuestro caso.

```
library(missForest)
set.seed(100)
complete_data_NAs_MISSFOREST_1 <- missForest(complete_data_NAs,
verbose=TRUE)
```

```
##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.001914282
##     difference(s): 6.555447e-08
##     time: 2.49 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.001912215
##     difference(s): 1.412998e-08
##     time: 2.48 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.001925875
##     difference(s): 1.547545e-08
##     time: 2.46 seconds
```

```
complete_data_NAs_MISSFOREST <- complete_data_NAs_MISSFOREST_1$ximp
head(complete_data_NAs_MISSFOREST)
```

```
##           ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 4   7.281393 214.3734 22018.42    8.059332 326.4310     363.2665
18.436524
## 5   9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6   5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
## 7  10.223862 248.0717 28749.72    7.513408 393.6634     283.6516
13.789695
## 8   8.635849 203.3615 13672.09    4.563009 330.3556     474.6076
12.363817
## 10  7.453271 227.2315 25484.51    9.077200 404.0416     563.8855
17.927806
##    Trihalomethanes Turbidity Potability
## 4        100.34167  4.628771          0
## 5         31.99799  4.075075          0
## 6         54.91786  2.559708          0
## 7         84.60356  2.672989          0
## 8         62.79831  4.401425          0
## 10        71.97660  4.370562          0
```

Utilizando la opción verbose=TRUE podemos ver el error medio en cada iteración y la diferencia, un indicador de la mejora en cada iteración, vemos que aunque aumentemos las iteraciones no mejoraremos mucho el resultado, por defecto no siempre utiliza el mismo número de iteraciones, se basa en estos valores para el número, podemos modificar el valor ntree, que indica el número de árboles a generar, como el conjunto de datos es pequeño, pongamos un valor elevado y probemos:

```
set.seed(100)
complete_data_NAs_MISSFOREST_2 <- missForest(complete_data_NAs,
verbose=TRUE, ntree=500)
```

```
##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.001889575
##     difference(s): 5.642094e-08
##     time: 13.25 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.001877447
##     difference(s): 3.538039e-09
##     time: 12.99 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.001880004
##     difference(s): 3.247985e-09
##     time: 13.18 seconds
##
##   missForest iteration 4 in progress...done!
##     estimated error(s): 0.001883395
##     difference(s): 3.156605e-09
##     time: 13.11 seconds
##
##   missForest iteration 5 in progress...done!
##     estimated error(s): 0.001883695
##     difference(s): 2.880213e-09
##     time: 12.66 seconds
##
##   missForest iteration 6 in progress...done!
##     estimated error(s): 0.001882746
##     difference(s): 2.984796e-09
##     time: 12.64 seconds
```

```
complete_data_NAs_MISSFOREST_test <- complete_data_NAs_MISSFOREST_2$ximp
head(complete_data_NAs_MISSFOREST)
```

```
##           ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 4   7.281393 214.3734 22018.42    8.059332 326.4310     363.2665
18.436524
## 5   9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
```

```
## 6   5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
## 7  10.223862 248.0717 28749.72    7.513408 393.6634     283.6516
13.789695
## 8   8.635849 203.3615 13672.09    4.563009 330.3556     474.6076
12.363817
## 10  7.453271 227.2315 25484.51    9.077200 404.0416     563.8855
17.927806
##    Trihalomethanes Turbidity Potability
## 4        100.34167  4.628771          0
## 5         31.99799  4.075075          0
## 6         54.91786  2.559708          0
## 7         84.60356  2.672989          0
## 8         62.79831  4.401425          0
## 10        71.97660  4.370562          0
```

Calculamos el error cuadrático medio de las imputaciones en cada variable en ambos casos y veamos la diferencia:

```
# Realizamos los mismos pasos que con el modelo de imputación MICE para
comparar los vectores de cada variable.
actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted =
complete_data_NAs_MISSFOREST_test[is.na(complete_data_NAs$ph), ]$ph
result_MISSFOREST_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted =
complete_data_NAs_MISSFOREST_test[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_MISSFOREST_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_MISSFOREST_test[is.na(complete_data_NAs$Trihalomethanes
), ]$Trihalomethanes
result_MISSFOREST_Trihalomethanes = Metrics::rmse(actual, predicted)

result_MISSFOREST_ph

## [1] 1.510945

result_MISSFOREST_Sulfate

## [1] 37.01338

result_MISSFOREST_Trihalomethanes

## [1] 17.60136
```

```
# Realizamos los mismos pasos que con el modelo de imputación MICE para
comparar los vectores de cada variable.
actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted = complete_data_NAs_MISSFOREST[is.na(complete_data_NAs$ph),
]$ph
result_MISSFOREST_2_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted =
complete_data_NAs_MISSFOREST[is.na(complete_data_NAs$Sulfate), ]$Sulfate
result_MISSFOREST_2_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_MISSFOREST[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_MISSFOREST_2_Trihalomethanes = Metrics::rmse(actual, predicted)

result_MISSFOREST_2_ph

## [1] 1.517278

result_MISSFOREST_2_Sulfate

## [1] 37.36304

result_MISSFOREST_2_Trihalomethanes

## [1] 17.47179
```

La diferencia no justifica el mayor tiempo de ejecución por tanto, optamos por la primera opción. Como suele ser habitual, los métodos basados en random forest ofrecen resultados muy buenos en comparación con otros. Guardamos los resultados de RMSE del modelo para cada variable.

```
add_value(result_MISSFOREST_ph, "result_MISSFOREST_ph")
add_value(result_MISSFOREST_Sulfate, "result_MISSFOREST_Sulfate")
add_value(result_MISSFOREST_Trihalomethanes,
"result_MISSFOREST_Trihalomethanes")
res

## [1] "Resultado"        "2.183171"         "55.8299"
"22.33897"
## [5] "1.51094525896161" "37.0133765429725" "17.6013623569472"

name

## [1] "Model"                              "result_MICE_ph"
## [3] "result_MICE_Sulfate"                "result_MICE_Trihalomethanes"
## [5] "result_MISSFOREST_ph"               "result_MISSFOREST_Sulfate"
## [7] "result_MISSFOREST_Trihalomethanes"
```

Aplicamos Hmisc

Hmisc es un paquete de usos múltiples útil para análisis de datos, gráficos de alto nivel, imputación de valores perdidos, creación avanzada de tablas, ajuste y diagnóstico de modelos (regresión lineal, regresión logística y regresión de cox), etc. También ofrece 2 potentes funciones para imputar valores perdidos, estas son impute () y aregImpute ().

La función impute () simplemente imputa el valor perdido utilizando el método estadístico definido por el usuario (media, máxima, media), su valor predeterminado es la mediana. Por otro lado, aregImpute () permite la imputación de valores mediante regresión aditiva, bootstrapping y coincidencia de medias predictiva. Luego, un modelo aditivo flexible (método de regresión no paramétrico) se ajusta a las muestras tomadas con reemplazos de los datos originales y los valores perdidos (actúa como variable dependiente) se predicen usando valores no perdidos (variable independiente). Luego, utiliza la coincidencia de medias predictiva (predeterminada) para imputar los valores perdidos. La coincidencia de medias predictiva funciona bien para continuos y categóricos (binarios y multinivel) sin la necesidad de calcular residuos y ajuste de máxima probabilidad. En el bootstrap, se utilizan diferentes remuestreos de bootstrap para cada una de las múltiples imputaciones.

Hmisc asume linealidad en las variables que se predicen, puede presenmtar un problema en nuestro caso, pero veremos los resultados. Estableceremos “n.impute” en 5 para no generrar muchos subconjuntos, misma lógica que con MICE.

```
library(Hmisc)
set.seed(100)

complete_data_NAs_HMISC_areg <- aregImpute(~ Sulfate + Hardness + Solids
+ Chloramines + ph + Conductivity + Organic_carbon + Trihalomethanes +
Turbidity + Potability, data = complete_data_NAs, n.impute = 5, type=
"pmm", nk=3, burnin=10)

## Iteration 1 Iteration 2 Iteration 3 Iteration 4 Iteration 5 Iteration
6 Iteration 7 Iteration 8 Iteration 9 Iteration 10 Iteration 11 Iteration
12 Iteration 13 Iteration 14 Iteration 15

print(complete_data_NAs_HMISC_areg)

##
## Multiple Imputation using Bootstrap and PMM
##
## aregImpute(formula = ~Sulfate + Hardness + Solids + Chloramines +
##     ph + Conductivity + Organic_carbon + Trihalomethanes + Turbidity +
##     Potability, data = complete_data_NAs, n.impute = 5, nk = 3,
##     type = "pmm", burnin = 10)
##
## n: 2011  p: 10   Imputations: 5      nk: 3
##
## Number of NAs:
```

```
##         Sulfate        Hardness          Solids     Chloramines
ph
##             479               0               0               0
301
##    Conductivity  Organic_carbon Trihalomethanes       Turbidity
Potability
##               0               0              99               0
0
##
##                  type d.f.
## Sulfate             s    2
## Hardness            s    2
## Solids              s    2
## Chloramines         s    2
## ph                  s    2
## Conductivity        s    2
## Organic_carbon      s    2
## Trihalomethanes     s    1
## Turbidity           s    2
## Potability          l    1
##
## Transformation of Target Variables Forced to be Linear
##
## R-squares for Predicting Non-Missing Values for Each Variable
## Using Last Imputations of Predictors
##         Sulfate              ph Trihalomethanes
##           0.065           0.051           0.013
```

Podemos probar con diferentes valores nk, con la opción nk=c(0,3:5) obtenemos los resultados de R^2 para distintos valores de nk, podemos valorar que opción usar:

```
library(Hmisc)
set.seed(100)

complete_data_NAs_HMISC_areg_test <- aregImpute(~ Sulfate + Hardness +
Solids + Chloramines + ph + Organic_carbon + Trihalomethanes + Turbidity
+ Potability, data = complete_data_NAs, n.impute = 10, type= "pmm",
nk=c(0,3:5), tlinear = FALSE)

## Iteration 1 Iteration 2 Iteration 3 Iteration 4 Iteration 5 Iteration
6 Iteration 7 Iteration 8 Iteration 9 Iteration 10 Iteration 11 Iteration
12 Iteration 13

print(complete_data_NAs_HMISC_areg_test)

##
## Multiple Imputation using Bootstrap and PMM
##
## aregImpute(formula = ~Sulfate + Hardness + Solids + Chloramines +
##     ph + Organic_carbon + Trihalomethanes + Turbidity + Potability,
##     data = complete_data_NAs, n.impute = 10, nk = c(0, 3:5),
```

```
##     tlinear = FALSE, type = "pmm")
##
## n: 2011  p: 9    Imputations: 10     nk: 0
##
## Number of NAs:
##         Sulfate        Hardness          Solids     Chloramines
ph
##             479               0               0               0
301
##  Organic_carbon Trihalomethanes       Turbidity      Potability
##               0              99               0               0
##
##                 type d.f.
## Sulfate            s    1
## Hardness           s    1
## Solids             s    1
## Chloramines        s    1
## ph                 s    1
## Organic_carbon     s    1
## Trihalomethanes    s    1
## Turbidity          s    1
## Potability         l    1
##
## R-squares for Predicting Non-Missing Values for Each Variable
## Using Last Imputations of Predictors
##         Sulfate              ph Trihalomethanes
##           0.051           0.022           0.006
##
## Resampling results for determining the complexity of imputation models
##
## Variable being imputed: Sulfate
##                                            nk=0    nk=3    nk=4
nk=5
## Bootstrap bias-corrected R^2              0.0243  0.0625  0.0574
0.0502
## 10-fold cross-validated  R^2              0.0288  0.0689  0.0559
0.0617
## Bootstrap bias-corrected mean   |error|  32.2440 46.0083 46.6924
46.0407
## 10-fold cross-validated  mean   |error| 334.7467 45.5396 46.7290
46.6970
## Bootstrap bias-corrected median |error|  26.2257 36.7337 36.4072
35.7148
## 10-fold cross-validated  median |error| 333.1827 35.2310 36.6979
37.1386
##
## Variable being imputed: ph
##                                          nk=0   nk=3   nk=4   nk=5
## Bootstrap bias-corrected R^2            0.0111 0.0456 0.0414 0.0347
## 10-fold cross-validated  R^2            0.0287 0.0488 0.0461 0.0531
```

```
## Bootstrap bias-corrected mean   |error| 1.2227 1.7429 1.7644 1.7271
## 10-fold cross-validated  mean   |error| 7.0978 1.7193 1.7730 1.7835
## Bootstrap bias-corrected median |error| 0.9699 1.3269 1.3975 1.3884
## 10-fold cross-validated  median |error| 7.0361 1.3942 1.4357 1.4086
##
## Variable being imputed: Trihalomethanes
##                                          nk=0     nk=3     nk=4
nk=5
## Bootstrap bias-corrected R^2            -0.00306 -0.00326 -0.00757 -
0.01319
## 10-fold cross-validated  R^2             0.00748  0.00461  0.00361
0.00551
## Bootstrap bias-corrected mean   |error| 12.69502 18.77391 19.10445
20.34643
## 10-fold cross-validated  mean   |error| 66.31036 18.08762 20.04604
19.65292
## Bootstrap bias-corrected median |error| 10.54825 16.17004 16.51517
16.95737
## 10-fold cross-validated  median |error| 66.47837 15.65838 17.83833
16.64646
```

Muestra como resultado los valores R^2 para los valores perdidos predichos. Cuanto mayor sean éstos, mejores son los valores predichos. En nuestro caso hemos obtenido valores de R^2 muy bajos, por lo que podemos concluir que los valores predichos con hmisc en nuestro caso son muy malos. Podemos comprobar el error con los valores reales en cada iteración:

```
for (i in 1:10){
  temp_data <- impute.transcan(complete_data_NAs_HMISC_areg_test,
imputation = i, data = complete_data_NAs, list.out = TRUE,
                           pr = FALSE, check = FALSE)
  temp_data <- data.frame(temp_data)
  actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
  predicted = temp_data[is.na(complete_data_NAs$ph), ]$ph
  result_HMISC_ph = Metrics::rmse(actual, predicted)

  actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
  predicted = temp_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
  result_HMISC_Sulfate = Metrics::rmse(actual, predicted)

  actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
  predicted = temp_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
  result_HMISC_Trihalomethanes = Metrics::rmse(actual, predicted)

  cat("ph RMSE:\t\t", result_HMISC_ph, "\n")
  cat("Sulfate RMSE:\t\t", result_HMISC_Sulfate, "\n")
  cat("Trihalomethanes RMSE:   ", result_HMISC_Trihalomethanes, "\n")
```

```
  cat("---------------------------------\n")
}
```

```
## ph RMSE:      2.305943
## Sulfate RMSE:         53.42825
## Trihalomethanes RMSE:    22.45553
## ---------------------------------
## ph RMSE:      2.145187
## Sulfate RMSE:         53.65322
## Trihalomethanes RMSE:    22.7522
## ---------------------------------
## ph RMSE:      2.263339
## Sulfate RMSE:         57.32731
## Trihalomethanes RMSE:    23.05537
## ---------------------------------
## ph RMSE:      2.224341
## Sulfate RMSE:         61.15955
## Trihalomethanes RMSE:    22.50458
## ---------------------------------
## ph RMSE:      2.037616
## Sulfate RMSE:         52.695
## Trihalomethanes RMSE:    23.58033
## ---------------------------------
## ph RMSE:      2.178177
## Sulfate RMSE:         58.02493
## Trihalomethanes RMSE:    24.70198
## ---------------------------------
## ph RMSE:      2.179307
## Sulfate RMSE:         56.44421
## Trihalomethanes RMSE:    22.9452
## ---------------------------------
## ph RMSE:      2.104773
## Sulfate RMSE:         55.93981
## Trihalomethanes RMSE:    24.62824
## ---------------------------------
## ph RMSE:      2.346371
## Sulfate RMSE:         55.24633
## Trihalomethanes RMSE:    24.22919
## ---------------------------------
## ph RMSE:      2.098786
## Sulfate RMSE:         58.13226
## Trihalomethanes RMSE:    19.29753
## ---------------------------------
```

Comparemos la imputación 10 con los valores reales, es una opción muy optimista (nos ponemos en el mejor caso) pero viendo los resultados será dificil incluirlo en los métodos finales.

```
# Realizamos los mismos pasos que con los modelos de imputación MICE y
MISSFOREST para comparar los vectores de cada variable.
```

```r
complete_data_NAs_HMISC <-
impute.transcan(complete_data_NAs_HMISC_areg_test, imputation = 10, data
= complete_data_NAs, list.out = TRUE,
                            pr = FALSE, check = FALSE)
complete_data_NAs_HMISC <- data.frame(complete_data_NAs_HMISC)


actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted = complete_data_NAs_HMISC[is.na(complete_data_NAs$ph), ]$ph
result_HMISC_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted = complete_data_NAs_HMISC[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_HMISC_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_HMISC[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_HMISC_Trihalomethanes = Metrics::rmse(actual, predicted)

result_HMISC_ph

## [1] 2.098786

result_HMISC_Sulfate

## [1] 58.13226

result_HMISC_Trihalomethanes

## [1] 19.29753

add_value(result_HMISC_ph, "result_HMISC_ph")
add_value(result_HMISC_Sulfate, "result_HMISC_Sulfate")
add_value(result_HMISC_Trihalomethanes, "result_HMISC_Trihalomethanes")
res

##  [1] "Resultado"        "2.183171"         "55.8299"
"22.33897"
##  [5] "1.51094525896161" "37.0133765429725" "17.6013623569472"
"2.09878631291901"
##  [9] "58.1322568883055" "19.2975277754206"

name

##  [1] "Model"                              "result_MICE_ph"
##  [3] "result_MICE_Sulfate"                "result_MICE_Trihalomethanes"
##  [5] "result_MISSFOREST_ph"               "result_MISSFOREST_Sulfate"
##  [7] "result_MISSFOREST_Trihalomethanes" "result_HMISC_ph"
```

```
##  [9] "result_HMISC_Sulfate"
"result_HMISC_Trihalomethanes"
```

Aplicamos mi

El paquete mi (imputación múltiple con diagnósticos) proporciona varias funciones para tratar los valores perdidos. Al igual que otros paquetes, también crea varios modelos de imputación para aproximar los valores perdidos. Y utiliza el método predictivo de coincidencia de medias (igual que mice: para cada observación en una variable con un valor perdido, encontramos la observación de los valores disponibles) con la más cercana media predictiva para esa variable. El valor observado de esta "coincidencia" se utiliza luego como valor imputado), como novedad, agrega ruido al proceso de imputación para resolver el problema de las restricciones aditivas. Los argumentos por defecto son: rand.imp.method "bootstrap", n.imp 3 y n.iter 30.

La función summary() nos ofece información de los valores imputados. La imputación múltiple es un proceso iterativo como ya hemos comentado anteriormente, en el que ir evaluando los resultados para optimizar dicha imputación, pero no nos extenderemos mucho en el proceso, dado que los resultados en distintas pruebas no han sido óptimos.

```
library(mi)
```

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

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

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loading required package: stats4
```

```
## Registered S3 methods overwritten by 'lme4':
##   method                          from
##   cooks.distance.influence.merMod car
##   influence.merMod                car
##   dfbeta.influence.merMod         car
##   dfbetas.influence.merMod        car
```

```
## mi (Version 1.0, packaged: 2015-04-16 14:03:10 UTC; goodrich)
```

```
## mi  Copyright (C) 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015
Trustees of Columbia University
```

```
## This program comes with ABSOLUTELY NO WARRANTY.
```

```
## This is free software, and you are welcome to redistribute it
```

```
## under the General Public License version 2 or later.
```

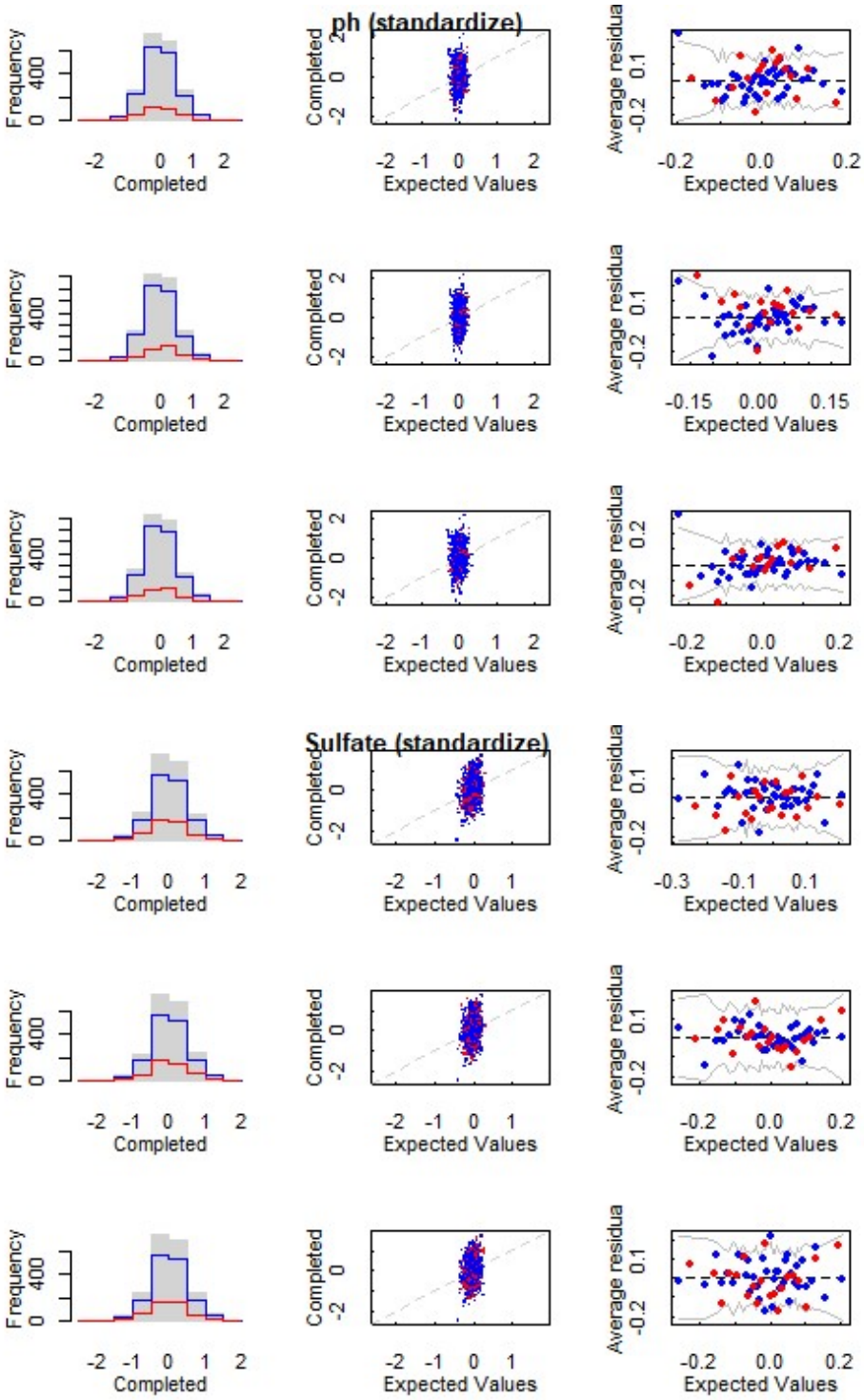
```
## Execute RShowDoc('COPYING') for details.

##
## Attaching package: 'mi'

## The following object is masked from 'package:tidyr':
##
##     complete

## The following objects are masked from 'package:mice':
##
##     complete, pool

set.seed(100)
complete_data_NAs_MI_mi <- mi(complete_data_NAs)
summary(complete_data_NAs_MI_mi)

## $ph
## $ph$is_missing
## missing
## FALSE  TRUE
##  1710   301
##
## $ph$imputed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.705863 -0.346296 -0.002425  0.009297  0.348529  1.522046
##
## $ph$observed
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.1801 -0.3146 -0.0161  0.0000  0.3026  2.1964
##
##
## $Hardness
## $Hardness$is_missing
## [1] "all values observed"
##
## $Hardness$observed
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.87644 -0.29452  0.01875  0.00000  0.31367  1.85950
##
##
## $Solids
## $Solids$is_missing
## [1] "all values observed"
##
## $Solids$observed
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.24947 -0.36459 -0.05693  0.00000  0.30462  2.00013
##
##
## $Chloramines
```

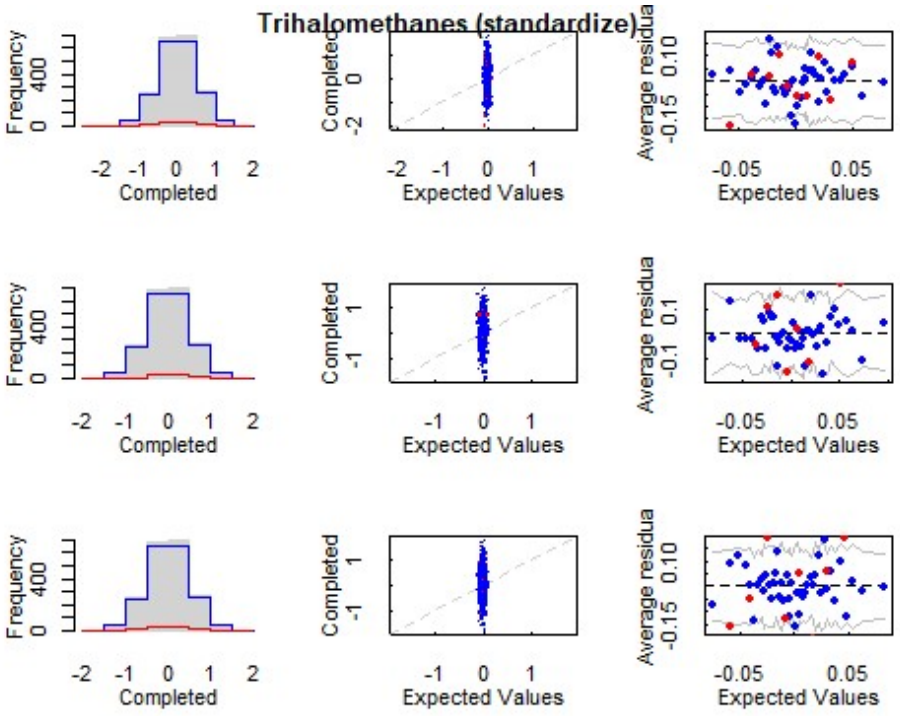
```
## $Chloramines$is_missing
## [1] "all values observed"
##
## $Chloramines$observed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.812025 -0.314056  0.003019  0.000000  0.307728  1.890644
##
##
## $Sulfate
## $Sulfate$is_missing
## missing
## FALSE  TRUE
##  1532   479
##
## $Sulfate$imputed
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.55094 -0.35113 -0.03048 -0.01974  0.30962  1.61649
##
## $Sulfate$observed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.446632 -0.308766 -0.007273  0.000000  0.322843  1.765908
##
##
## $Conductivity
## $Conductivity$is_missing
## [1] "all values observed"
##
## $Conductivity$observed
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.39326 -0.37074 -0.01902  0.00000  0.34596  2.02457
##
##
## $Organic_carbon
## $Organic_carbon$is_missing
## [1] "all values observed"
##
## $Organic_carbon$observed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.828250 -0.335885 -0.005367  0.000000  0.349679  1.902129
##
##
## $Trihalomethanes
## $Trihalomethanes$is_missing
## missing
## FALSE  TRUE
##  1912    99
##
## $Trihalomethanes$imputed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -2.008769 -0.341593  0.029971  0.005042  0.365451  1.241456
```

```
##
## $Trihalomethanes$observed
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.801728 -0.327209  0.005143  0.000000  0.340353  1.799936
##
##
## $Turbidity
## $Turbidity$is_missing
## [1] "all values observed"
##
## $Turbidity$observed
##       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.
## -1.6144943 -0.3375511 -0.0009942  0.0000000  0.3488494  1.6178844
##
##
## $Potability
## $Potability$is_missing
## [1] "all values observed"
##
## $Potability$observed
##
##    1    2
## 1200  811
```

Mi() también nos ofrece herramientas para la visualización de datos. Veremos en azul los valores observados, los imputados en rojo y en gris completos (observados e imputados).

```
plot(complete_data_NAs_MI_mi, ask=FALSE)
```


ph (standardize)
Sulfate (standardize)
Frequency
Completed
Expected Values
Average residua




```
complete_data_NAs_MI <- mi::complete(complete_data_NAs_MI_mi, m=1)

# Realizamos los mismos pasos que con los modelos anteriores para
comparar los vectores de cada variable.
actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted = complete_data_NAs_MI[is.na(complete_data_NAs$ph), ]$ph
result_MI_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted = complete_data_NAs_MI[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_MI_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_MI[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_MI_Trihalomethanes = Metrics::rmse(actual, predicted)

result_MI_ph

## [1] 2.227274

result_MI_Sulfate

## [1] 56.83972
```

```
result_MI_Trihalomethanes
```

```
## [1] 26.29246
```

```
add_value(result_MI_ph, "result_MI_ph")
add_value(result_MI_Sulfate, "result_MI_Sulfate")
add_value(result_MI_Trihalomethanes, "result_MI_Trihalomethanes")
```

Aplicamos kNN

En este método de imputación, los valores perdidos de un atributo se imputan utilizando el número dado de atributos que son más similares al atributo cuyos valores faltan. La similitud de dos atributos se determina mediante una función de distancia. Las principales características que podríamos decir son: No se requiere la creación de un modelo predictivo para cada atributo con datos faltantes, los atributos con múltiples valores perdidos se pueden tratar fácilmente y se tiene en cuenta la estructura de correlación de los datos, en nuestro caso hemos visto que la correlación es prácticamente 0, pero veamos que resultados nos ofrece.

Como vemos en los apuntes de la asignatura, es un método muy sensible al valor de k que le demos, en principio se recomienda probar con la raíz cuadrada del número de variables.

```
set.seed(100)
complete_data_NAs_kNN <- kNN(complete_data_NAs, k=3)
complete_data_NAs_kNN <- complete_data_NAs_kNN[, 1:10]
```

Comparemos los resultados con los valores reales:

```
# Realizamos los mismos pasos que con los modelos anteriores para
comparar los vectores de cada variable.
actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$ph), ]$ph
result_kNN_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_kNN_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_kNN[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_kNN_Trihalomethanes = Metrics::rmse(actual, predicted)

result_kNN_ph
```

```
## [1] 1.786831
```

```
result_kNN_Sulfate
```

```
## [1] 43.6915
```

```
result_kNN_Trihalomethanes
```

```
## [1] 20.25328
```

Tras varias pruebas con distintos valores de k, hemos decidido k=100 como valor con menor error medio.

```
set.seed(100)
complete_data_NAs_kNN <- kNN(complete_data_NAs, k=100)
complete_data_NAs_kNN <- complete_data_NAs_kNN
head(complete_data_NAs_kNN)
```

```
##          ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1  7.005555 214.3734 22018.42    8.059332 330.8700     363.2665
18.436524
## 2  9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 3  5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
## 4 10.223862 248.0717 28749.72    7.513408 393.6634     283.6516
13.789695
## 5  8.635849 203.3615 13672.09    4.563009 333.2864     474.6076
12.363817
## 6  7.176370 227.2315 25484.51    9.077200 404.0416     563.8855
17.927806
##   Trihalomethanes Turbidity Potability ph_imp Hardness_imp Solids_imp
## 1       100.34167  4.628771          0   TRUE        FALSE      FALSE
## 2        31.99799  4.075075          0  FALSE        FALSE      FALSE
## 3        54.91786  2.559708          0  FALSE        FALSE      FALSE
## 4        84.60356  2.672989          0  FALSE        FALSE      FALSE
## 5        62.79831  4.401425          0  FALSE        FALSE      FALSE
## 6        71.97660  4.370562          0   TRUE        FALSE      FALSE
##   Chloramines_imp Sulfate_imp Conductivity_imp Organic_carbon_imp
## 1           FALSE        TRUE            FALSE              FALSE
## 2           FALSE       FALSE            FALSE              FALSE
## 3           FALSE       FALSE            FALSE              FALSE
## 4           FALSE       FALSE            FALSE              FALSE
## 5           FALSE        TRUE            FALSE              FALSE
## 6           FALSE       FALSE            FALSE              FALSE
##   Trihalomethanes_imp Turbidity_imp Potability_imp
## 1               FALSE         FALSE          FALSE
## 2               FALSE         FALSE          FALSE
## 3               FALSE         FALSE          FALSE
## 4               FALSE         FALSE          FALSE
## 5               FALSE         FALSE          FALSE
## 6               FALSE         FALSE          FALSE
```

```r
# Realizamos los mismos pasos que con los modelos anteriores para
comparar los vectores de cada variable.
actual = complete_data[is.na(complete_data_NAs$ph), ]$ph
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$ph), ]$ph
result_kNN_ph = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_kNN_Sulfate = Metrics::rmse(actual, predicted)

actual = complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
predicted =
complete_data_NAs_kNN[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_kNN_Trihalomethanes = Metrics::rmse(actual, predicted)

result_kNN_ph

## [1] 1.509153

result_kNN_Sulfate

## [1] 37.55507

result_kNN_Trihalomethanes

## [1] 16.93846
```

No estandarizar los datos antes de aplicar un método basado en distancias es un error, pero no tenemos una referencia previa con la que comparar, por tanto, repetiremos el proceso con el método hasta ahora mas prometedor, missForest escalado y compararemos los errores:

```r
library(caret)

##
## Attaching package: 'caret'

## The following objects are masked from 'package:Metrics':
##
##     precision, recall

## The following object is masked from 'package:purrr':
##
##     lift

## The following object is masked from 'package:survival':
##
##     cluster
```

```
scaled_test <- scale(complete_data_NAs)
```

Aplicamos directamente con valor óptimo tras varias iteraciones:

```
set.seed(100)
complete_data_NAs_kNN <- kNN(scaled_test, k=100)
complete_data_NAs_kNN <- as.data.frame(complete_data_NAs_kNN[, 1:10])

# Realizamos los mismos pasos que con los modelos anteriores para
comparar los vectores de cada variable.
actual = scale(complete_data[is.na(complete_data_NAs$ph), ]$ph)
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$ph), ]$ph
result_kNN_ph_scale = Metrics::rmse(actual, predicted)

actual = scale(complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate)
predicted = complete_data_NAs_kNN[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_kNN_Sulfate_scale = Metrics::rmse(actual, predicted)

actual = scale(complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes)
predicted =
complete_data_NAs_kNN[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes
result_kNN_Trihalomethanes_scale = Metrics::rmse(actual, predicted)

result_kNN_ph_scale

## [1] 0.9579691

result_kNN_Sulfate_scale

## [1] 0.9572322

result_kNN_Trihalomethanes_scale

## [1] 0.9839852
```

Usaremos de referencia missForest, el mejor método hasta ahora:

```
library(missForest)
set.seed(100)
complete_data_NAs_MISSFOREST_scale <-
missForest(scale(complete_data_NAs), verbose=TRUE)

##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.5421674
##     difference(s): 0.003738684
##     time: 2.56 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.5383293
```

```
##     difference(s): 0.0009312421
##     time: 2.69 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.5438197
##     difference(s): 0.0009318232
##     time: 2.63 seconds

complete_data_NAs_MISSFOREST_scaled <-
as.data.frame(complete_data_NAs_MISSFOREST_scale$ximp)

# Realizamos los mismos pasos que con el modelo de imputación MICE para
comparar los vectores de cada variable.
actual = scale(complete_data[is.na(complete_data_NAs$ph), ]$ph)
predicted =
complete_data_NAs_MISSFOREST_scaled[is.na(complete_data_NAs$ph), ]$ph
result_MISSFOREST_ph_scale = Metrics::rmse(actual, predicted)

actual = scale(complete_data[is.na(complete_data_NAs$Sulfate), ]$Sulfate)
predicted =
complete_data_NAs_MISSFOREST_scaled[is.na(complete_data_NAs$Sulfate),
]$Sulfate
result_MISSFOREST_Sulfate_scale = Metrics::rmse(actual, predicted)

actual = scale(complete_data[is.na(complete_data_NAs$Trihalomethanes),
]$Trihalomethanes)
predicted =
complete_data_NAs_MISSFOREST_scaled[is.na(complete_data_NAs$Trihalomethan
es), ]$Trihalomethanes
result_MISSFOREST_Trihalomethanes_scale = Metrics::rmse(actual,
predicted)

result_MISSFOREST_ph_scale

## [1] 0.9548765

result_MISSFOREST_Sulfate_scale

## [1] 0.9467341

result_MISSFOREST_Trihalomethanes_scale

## [1] 1.027493
```

Añadimos todos los valores a nuestros resultados.

```
add_value(result_kNN_ph, "result_kNN_ph")
add_value(result_kNN_Sulfate, "result_kNN_Sulfate")
add_value(result_kNN_Trihalomethanes, "result_kNN_Trihalomethanes")

add_value(result_kNN_ph_scale, "result_kNN_ph_scale")
```

```
add_value(result_kNN_Sulfate_scale, "result_kNN_Sulfate_scale")
add_value(result_kNN_Trihalomethanes_scale,
"result_kNN_Trihalomethanes_scale")

add_value(result_MISSFOREST_ph_scale, "result_MISSFOREST_ph_scale")
add_value(result_MISSFOREST_Sulfate_scale,
"result_MISSFOREST_Sulfate_scale")
add_value(result_MISSFOREST_Trihalomethanes_scale,
"result_MISSFOREST_Trihalomethanes_scale")
```

Resumen y comparación de métodos de imputación

Vamos a imprimir los valores obtenidos hasta ahora de los errores respecto a los valores reales en todos los métodos de imputación:

```
cat(paste0(data.frame(cbind(name, res))[1,1], "\t\t\t\t"))

## Model

cat(paste0(data.frame(cbind(name, res))[1,2], "\n"))

## Resultado

cat("-----------------------------------------------------------------\n")

## -----------------------------------------------------------------

for (i in c(2,5,8,11,14,17,20)){
  cat(paste0(data.frame(cbind(name, res))[i,1], ":  "))
  cat(paste0(data.frame(cbind(name, res))[i,2], "\n"))
  cat(paste0(data.frame(cbind(name, res))[i+1,1], ":  "))
  cat(paste0(data.frame(cbind(name, res))[i+1,2], "\n"))
  cat(paste0(data.frame(cbind(name, res))[i+2,1], ":  "))
  cat(paste0(data.frame(cbind(name, res))[i+2,2], "\n"))
  cat("-----------------------------------------------------------------\n")
}

## result_MICE_ph:  2.183171
## result_MICE_Sulfate:  55.8299
## result_MICE_Trihalomethanes:  22.33897
## -----------------------------------------------------------------
## result_MISSFOREST_ph:  1.51094525896161
## result_MISSFOREST_Sulfate:  37.0133765429725
## result_MISSFOREST_Trihalomethanes:  17.6013623569472
## -----------------------------------------------------------------
## result_HMISC_ph:  2.09878631291901
## result_HMISC_Sulfate:  58.1322568883055
## result_HMISC_Trihalomethanes:  19.2975277754206
## -----------------------------------------------------------------
## result_MI_ph:  2.22727404135808
## result_MI_Sulfate:  56.8397207598641
## result_MI_Trihalomethanes:  26.2924611944089
```

```
## ------------------------------------------------------------
## result_kNN_ph:  1.50915281814896
## result_kNN_Sulfate:  37.5550658167241
## result_kNN_Trihalomethanes:  16.9384633022163
## ------------------------------------------------------------
## result_kNN_ph_scale:  0.957969131805744
## result_kNN_Sulfate_scale:  0.957232241571221
## result_kNN_Trihalomethanes_scale:  0.983985228121779
## ------------------------------------------------------------
## result_MISSFOREST_ph_scale:  0.954876539275608
## result_MISSFOREST_Sulfate_scale:  0.946734074691805
## result_MISSFOREST_Trihalomethanes_scale:  1.0274929424836
## ------------------------------------------------------------
```

Seleccionaremos los métodos que mejores valores nos han dado, missForest y KNN.

Identificación y tratamiento de valores extremos.

A continuación debemos estudiar los valores atípicos en nuestro conjunto de datos, pero ya hemos hecho el estudio en el apartado anterior en el estudio de la distribución de nuestros datos para la selección de métodos de imputación, por tanto, lo que haremos será generar distintos conjuntos de datos, si recordamos, hemos generado dos conjuntos de datos eliminando outliers del conjunto de los datos que no tienen valores nulos:

data_Tukey_outliers <- remove_outliers_Tukey(complete_data) data_sd_outliers <- remove_outliers_sd(complete_data)

Lo ideal sería tener varios conjuntos antes de imputar valores nulos, hemos decidido crear 3 conjuntos: Uno con todos los datos, sobre el que imputaremos valores nulos y luego eliminaremos outliers. Pero como estos outliers pueden afectar en los métodos de imputación de valores, crearemos dos conjuntos más eliminando outliers previamente, y después, un conjunto por cada función definida para eliminar outliers. Para lidiar con los valores nulos a la hora de eliminar outliers hemos decidido una opción quizás un poco rudimentaria, pero queríamos conservar los valores nulos originales y eliminar las filas en caso de que los valores no nulos fueran outliers. Para esto modificamos la función que ofrece los límites para cada columna de lo que definiremos como outliers para mostrarlos por pantalla:

```
outliers_Tukey_na <- function(x) {

  Q1 <- quantile(x, probs=.25)
  Q3 <- quantile(x, probs=.75)
  iqr = Q3-Q1

 upper_limit = Q3 + (iqr*1.5)
 lower_limit = Q1 - (iqr*1.5)
 cat(upper_limit,"\t",lower_limit,"\n")
}
```

```
outliers_sd_na <- function(x) {

    upper_limit = mean(x) + 3*sd(x)
    lower_limit = mean(x) - 3*sd(x)

    cat(upper_limit,"\t",lower_limit,"\n")
}

outliers_Tukey_na(data[complete.cases(data$ph),]$ph)

## 11.01553      3.139631

outliers_Tukey_na(data[complete.cases(data$Hardness),]$Hardness)

## 276.3928      117.1252

outliers_Tukey_na(data[complete.cases(data$Solids),]$Solids)

## 44831.87      -1832.417

outliers_Tukey_na(data[complete.cases(data$Chloramines),]$Chloramines)

## 11.09609      3.146221

outliers_Tukey_na(data[complete.cases(data$Sulfate),]$Sulfate)

## 438.3262      229.3235

outliers_Tukey_na(data[complete.cases(data$Conductivity),]$Conductivity)

## 655.8791      191.6476

outliers_Tukey_na(data[complete.cases(data$Organic_carbon),]$Organic_carb
on)

## 23.29543      5.328026

outliers_Tukey_na(data[complete.cases(data$Trihalomethanes),]$Trihalometh
anes)

## 109.5769      23.60513

outliers_Tukey_na(data[complete.cases(data$Turbidity),]$Turbidity)

## 6.091233      1.848797
```

En base a estos valores eliminaremos columna a columna los queno se encuentren en los rangos aceptados:

```
test <- data
head(test)
```

```
##         ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1       NA 204.8905 20791.32    7.300212 368.5164     564.3087
10.379783
## 2 3.716080 129.4229 18630.06    6.635246       NA     592.8854
15.180013
## 3 8.099124 224.2363 19909.54    9.275884       NA     418.6062
16.868637
## 4 8.316766 214.3734 22018.42    8.059332 356.8861     363.2665
18.436524
## 5 9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6 5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
##   Trihalomethanes Turbidity Potability
## 1        86.99097  2.963135          0
## 2        56.32908  4.500656          0
## 3        66.42009  3.055934          0
## 4       100.34167  4.628771          0
## 5        31.99799  4.075075          0
## 6        54.91786  2.559708          0
```

Vemos la función summary para comprobar que después de la transformación los máximos y los mínimos han cambiado dentro de los parámetros definidos:

```
summary(test)
```

```
##        ph            Hardness          Solids         Chloramines
##  Min.   : 0.000   Min.   : 47.43   Min.   :  320.9   Min.   : 0.352
##  1st Qu.: 6.093   1st Qu.:176.85   1st Qu.:15666.7   1st Qu.: 6.127
##  Median : 7.037   Median :196.97   Median :20927.8   Median : 7.130
##  Mean   : 7.081   Mean   :196.37   Mean   :22014.1   Mean   : 7.122
##  3rd Qu.: 8.062   3rd Qu.:216.67   3rd Qu.:27332.8   3rd Qu.: 8.115
##  Max.   :14.000   Max.   :323.12   Max.   :61227.2   Max.   :13.127
##  NA's   :491
##     Sulfate       Conductivity   Organic_carbon  Trihalomethanes
##  Min.   :129.0   Min.   :181.5   Min.   : 2.20   Min.   :  0.738
##  1st Qu.:307.7   1st Qu.:365.7   1st Qu.:12.07   1st Qu.: 55.845
##  Median :333.1   Median :421.9   Median :14.22   Median : 66.622
##  Mean   :333.8   Mean   :426.2   Mean   :14.28   Mean   : 66.396
##  3rd Qu.:360.0   3rd Qu.:481.8   3rd Qu.:16.56   3rd Qu.: 77.337
##  Max.   :481.0   Max.   :753.3   Max.   :28.30   Max.   :124.000
##  NA's   :781                                     NA's   :162
##    Turbidity       Potability
##  Min.   :1.450   Min.   :0.0000
##  1st Qu.:3.440   1st Qu.:0.0000
##  Median :3.955   Median :0.0000
##  Mean   :3.967   Mean   :0.3901
##  3rd Qu.:4.500   3rd Qu.:1.0000
##  Max.   :6.739   Max.   :1.0000
##
```

```r
test <- subset(test, (ph < 11.01553 & ph > 3.139631) | is.na(ph))
test <- subset(test, Hardness < 276.3928 & Hardness > 117.1252)
test <- subset(test, Solids < 44831.87 & Solids > -1832.417)
test <- subset(test, Chloramines < 11.09609 & Chloramines > 3.146221)
test <- subset(test, (Sulfate < 438.3262 & Sulfate > 229.3235) |
is.na(Sulfate))
test <- subset(test, Conductivity < 655.8791 & Conductivity > 191.6476)
test <- subset(test, Organic_carbon < 23.29543 & Organic_carbon >
5.328026)
test <- subset(test, (Trihalomethanes < 109.5769 & Trihalomethanes >
23.60513) | is.na(Trihalomethanes))
test <- subset(test, Turbidity < 6.091233 & Turbidity > 1.848797)
no_ouliers_incomplete_Tukey <- test
head(no_ouliers_incomplete_Tukey)
```

```
##         ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1       NA 204.8905 20791.32    7.300212 368.5164     564.3087
10.379783
## 2 3.716080 129.4229 18630.06    6.635246       NA     592.8854
15.180013
## 3 8.099124 224.2363 19909.54    9.275884       NA     418.6062
16.868637
## 4 8.316766 214.3734 22018.42    8.059332 356.8861     363.2665
18.436524
## 5 9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6 5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
##   Trihalomethanes Turbidity Potability
## 1        86.99097  2.963135          0
## 2        56.32908  4.500656          0
## 3        66.42009  3.055934          0
## 4       100.34167  4.628771          0
## 5        31.99799  4.075075          0
## 6        54.91786  2.559708          0
```

```r
nrow(no_ouliers_incomplete_Tukey)
```

```
## [1] 2951
```

Los valores son correctos en todas las columnas:

```r
summary(no_ouliers_incomplete_Tukey)
```

```
##        ph            Hardness         Solids         Chloramines
##  Min.   : 3.149   Min.   :117.8   Min.   :  320.9   Min.   : 3.181
##  1st Qu.: 6.115   1st Qu.:177.8   1st Qu.:15521.4   1st Qu.: 6.175
##  Median : 7.036   Median :197.2   Median :20743.3   Median : 7.121
##  Mean   : 7.074   Mean   :196.5   Mean   :21598.0   Mean   : 7.119
##  3rd Qu.: 8.001   3rd Qu.:215.7   3rd Qu.:26981.8   3rd Qu.: 8.076
```

```
##  Max.   :10.947   Max.   :275.9   Max.   :44652.4   Max.   :11.087
##  NA's   :451
##     Sulfate       Conductivity   Organic_carbon   Trihalomethanes
##  Min.   :229.6   Min.   :201.6   Min.   : 5.362   Min.   : 23.79
##  1st Qu.:309.1   1st Qu.:365.2   1st Qu.:12.128   1st Qu.: 55.94
##  Median :333.3   Median :421.2   Median :14.243   Median : 66.30
##  Mean   :334.1   Mean   :425.5   Mean   :14.317   Mean   : 66.42
##  3rd Qu.:359.1   3rd Qu.:481.3   3rd Qu.:16.576   3rd Qu.: 77.32
##  Max.   :437.6   Max.   :652.5   Max.   :23.234   Max.   :108.85
##  NA's   :704                                      NA's   :147
##    Turbidity       Potability
##  Min.   :1.873   Min.   :0.0000
##  1st Qu.:3.441   1st Qu.:0.0000
##  Median :3.955   Median :0.0000
##  Mean   :3.969   Mean   :0.3819
##  3rd Qu.:4.498   3rd Qu.:1.0000
##  Max.   :6.084   Max.   :1.0000
##
```

Y ahora para la segunda función:

```
outliers_sd_na(data[complete.cases(data$ph),]$ph)

## 11.86375      2.297836

outliers_sd_na(data[complete.cases(data$Hardness),]$Hardness)

## 295.0088      97.73021

outliers_sd_na(data[complete.cases(data$Solids),]$Solids)

## 48319.81      -4291.62

outliers_sd_na(data[complete.cases(data$Chloramines),]$Chloramines)

## 11.87153      2.373022

outliers_sd_na(data[complete.cases(data$Sulfate),]$Sulfate)

## 458.0263      209.5253

outliers_sd_na(data[complete.cases(data$Conductivity),]$Conductivity)

## 668.6773      183.7329

outliers_sd_na(data[complete.cases(data$Organic_carbon),]$Organic_carbon)

## 24.20946      4.360484

outliers_sd_na(data[complete.cases(data$Trihalomethanes),]$Trihalomethane
s)

## 114.9213      17.87127
```

```r
outliers_sd_na(data[complete.cases(data$Turbidity),]$Turbidity)
```

```
## 6.307933      1.625639
```

```r
test <- data
test <- subset(test, (ph < 11.86375 & ph > 2.297836) | is.na(ph))
test <- subset(test, Hardness < 295.0088 & Hardness > 97.73021)
test <- subset(test, Solids < 48319.81 & Solids > -4291.62)
test <- subset(test, Chloramines < 11.87153 & Chloramines > 2.373022)
test <- subset(test, (Sulfate < 458.0263 & Sulfate > 209.5253) |
is.na(Sulfate))
test <- subset(test, Conductivity < 668.6773 & Conductivity > 183.7329)
test <- subset(test, Organic_carbon < 24.20946 & Organic_carbon >
4.360484)
test <- subset(test, (Trihalomethanes < 114.9213 & Trihalomethanes >
17.87127) | is.na(Trihalomethanes))
test <- subset(test, Turbidity < 6.307933 & Turbidity > 1.625639)
no_ouliers_incomplete_sd <- test
head(no_ouliers_incomplete_sd)
```

```
##         ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1       NA 204.8905 20791.32    7.300212 368.5164     564.3087
10.379783
## 2 3.716080 129.4229 18630.06    6.635246       NA     592.8854
15.180013
## 3 8.099124 224.2363 19909.54    9.275884       NA     418.6062
16.868637
## 4 8.316766 214.3734 22018.42    8.059332 356.8861     363.2665
18.436524
## 5 9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6 5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
##   Trihalomethanes Turbidity Potability
## 1        86.99097  2.963135          0
## 2        56.32908  4.500656          0
## 3        66.42009  3.055934          0
## 4       100.34167  4.628771          0
## 5        31.99799  4.075075          0
## 6        54.91786  2.559708          0
```

```r
nrow(no_ouliers_incomplete_sd)
```

```
## [1] 3161
```

Comprobamos con summary() que es correcto:

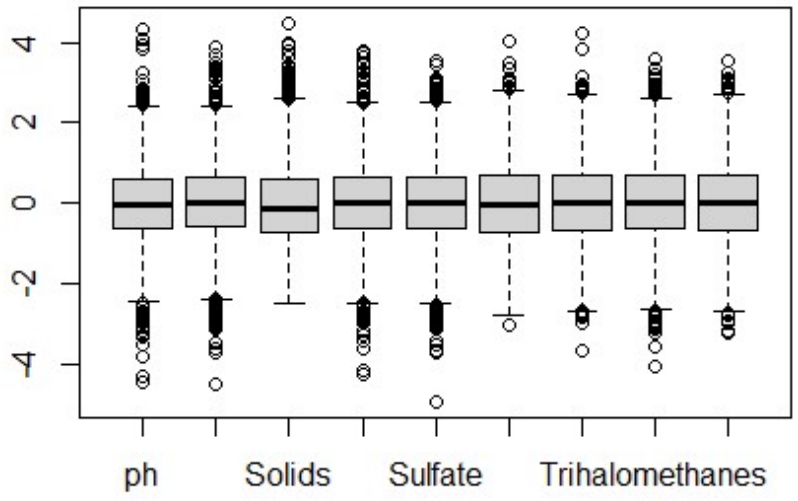
```r
summary(no_ouliers_incomplete_sd)
```

```
##        ph            Hardness          Solids         Chloramines
##  Min.   : 2.377   Min.   : 98.37   Min.   :  320.9   Min.   : 2.387
```

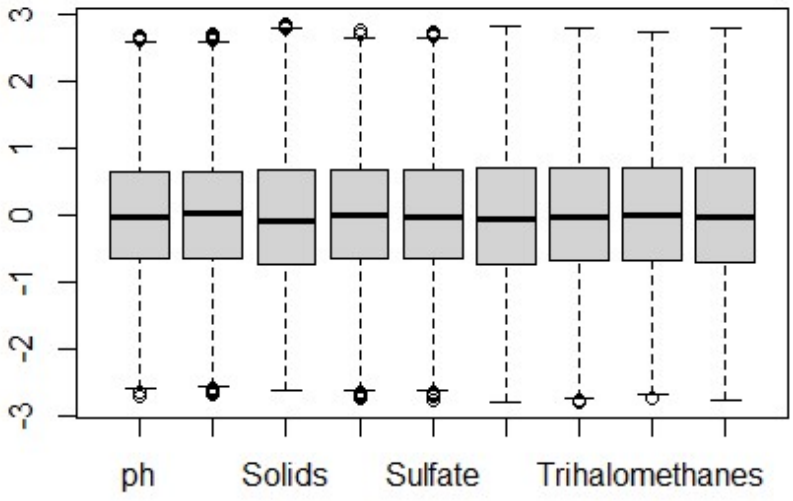
```
##  1st Qu.: 6.106   1st Qu.:177.17   1st Qu.:15547.2   1st Qu.: 6.136
##  Median : 7.037   Median :196.89   Median :20828.0   Median : 7.127
##  Mean   : 7.084   Mean   :196.24   Mean   :21771.3   Mean   : 7.112
##  3rd Qu.: 8.036   3rd Qu.:216.26   3rd Qu.:27141.4   3rd Qu.: 8.098
##  Max.   :11.621   Max.   :287.98   Max.   :48204.2   Max.   :11.754
##  NA's   :482
##     Sulfate       Conductivity   Organic_carbon   Trihalomethanes
##  Min.   :211.9   Min.   :201.6   Min.   : 4.372   Min.   : 17.92
##  1st Qu.:308.7   1st Qu.:365.7   1st Qu.:12.091   1st Qu.: 55.88
##  Median :333.2   Median :421.3   Median :14.236   Median : 66.58
##  Mean   :334.3   Mean   :425.7   Mean   :14.298   Mean   : 66.42
##  3rd Qu.:359.8   3rd Qu.:481.3   3rd Qu.:16.569   3rd Qu.: 77.30
##  Max.   :455.5   Max.   :666.7   Max.   :23.952   Max.   :114.21
##  NA's   :754                                      NA's   :155
##    Turbidity       Potability
##  Min.   :1.642   Min.   :0.0000
##  1st Qu.:3.440   1st Qu.:0.0000
##  Median :3.953   Median :0.0000
##  Mean   :3.965   Mean   :0.3831
##  3rd Qu.:4.498   3rd Qu.:1.0000
##  Max.   :6.308   Max.   :1.0000
##
```

Y comprobamos los boxplots de ambos conjuntos con los datos originales:

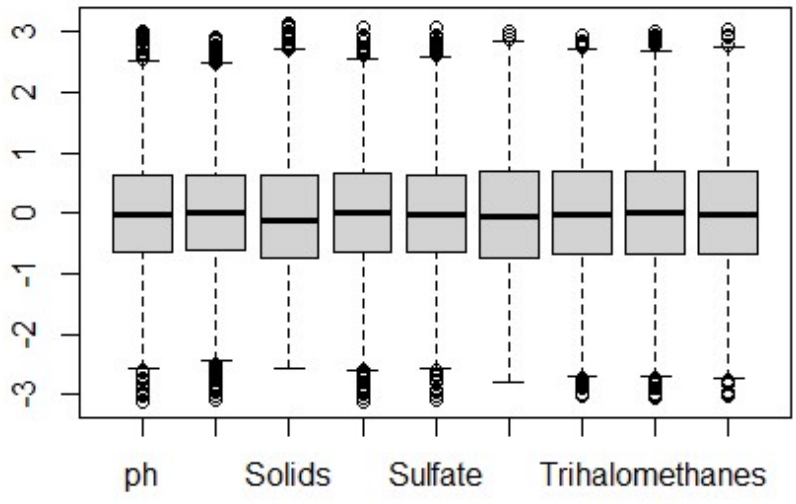
```
boxplot(scale(data[,1:9]))
```



```
boxplot(scale(no_ouliers_incomplete_Tukey[,1:9]))
```




```
boxplot(scale(no_ouliers_incomplete_sd[,1:9]))
```



Siguen presentando outliers, pero se puede apreciar el cambio, la función boxplot no tiene los mismos parametros para definir lo que es un outlier, podemos ver en cada columna los valores eliminados

```
sort(boxplot.stats(data$ph)$out)
```

```
##  [1]  0.0000000  0.2274991  0.9755780  0.9899122  1.4317816  1.7570371
##  [7]  1.8445384  1.9853834  2.1285314  2.3767681  2.5381158  2.5581028
## [13]  2.5692436  2.6120359  2.6908312  2.7985491  2.8035631  2.9251743
## [19]  2.9454691  2.9744294  3.1020756 11.0278799 11.0694563 11.1802845
## [25] 11.1806947 11.2191347 11.2354260 11.2445071 11.2678284 11.3017940
## [31] 11.3905431 11.4497393 11.4910109 11.4967025 11.4968589 11.5348805
## [37] 11.5631691 11.5687680 11.6211401 11.8980780 11.9077398 12.2469281
## [43] 13.1754017 13.3498886 13.5412402 14.0000000
```

```
cat("-----------------------------------------------------------------------
-----------------------------------------------\n")
```

```
## -----------------------------------------------------------------------
-----------------------------------------------
```

```
sort(boxplot.stats(data$Hardness)$out)
```

```
##  [1]  47.43200  73.49223  77.45959  81.71090  94.09131  94.81255
94.90898
##  [8]  97.28091  98.36791  98.45293  98.77164 100.45762 100.80652
103.17359
## [15] 103.46476 104.75242 105.85926 106.38011 107.34198 107.38333
```

```
108.69908
## [22] 108.91663 110.86579 110.90360 111.24641 111.47858 111.99403
112.29949
## [29] 112.82025 113.02447 113.17596 113.50470 113.83111 114.37145
114.46390
## [36] 114.73354 114.80758 115.39298 116.06195 116.29933 116.33828
116.72512
## [43] 116.90548 117.05731 276.69976 276.73357 277.06571 277.11695
278.03636
## [50] 278.05632 278.08145 278.14752 278.23175 278.34036 278.58511
278.61945
## [57] 279.23242 279.35717 280.08241 280.08965 281.26867 281.58216
281.59423
## [64] 282.73902 283.40957 283.89586 283.99728 284.09835 286.20176
286.56799
## [71] 287.37021 287.97554 291.46190 298.09868 300.29248 303.70263
304.23591
## [78] 306.62748 307.70602 308.25383 311.38396 317.33812 323.12400

cat("-----------------------------------------------------------------
-----------------------------------------------\n")

## -----------------------------------------------------------------
-------------------------------------------

sort(boxplot.stats(data$Solids)$out)

##  [1] 44868.46 44896.98 44982.73 45041.15 45050.00 45141.69 45148.81
45166.64
##  [9] 45166.91 45222.51 45243.03 45249.45 45510.58 45939.69 46077.36
46113.96
## [17] 46140.13 46718.56 46931.88 47022.75 47580.99 47591.28 47852.89
48002.08
## [25] 48007.87 48175.85 48204.17 48410.47 48621.56 49009.92 49074.73
49125.36
## [33] 49341.42 49456.59 50166.53 50279.26 50793.90 51731.82 52060.23
52318.92
## [41] 53735.90 55334.70 56320.59 56351.40 56488.67 56867.86 61227.20

cat("-----------------------------------------------------------------
-----------------------------------------------\n")

## -----------------------------------------------------------------
-------------------------------------------

sort(boxplot.stats(data$Chloramines)$out)

##  [1]  0.3520000  0.5303513  1.3908709  1.6839926  1.9202714  2.1026910
##  [7]  2.3866535  2.3979850  2.4560136  2.4586092  2.4843800  2.4985967
## [13]  2.5622555  2.5775553  2.6212676  2.6483899  2.6544910  2.7267656
## [19]  2.7417121  2.7508373  2.7857184  2.8557898  2.8625354  2.8660730
## [25]  2.9813790  2.9937441  3.0160326  3.0743161  3.1174410  3.1248326
```

```
## [31]  3.1395527 11.1016281 11.1291537 11.1431096 11.1707886 11.2086883
## [37] 11.2243946 11.2400004 11.2515073 11.2643858 11.2993902 11.3028312
## [43] 11.4484693 11.5235975 11.5431905 11.5861511 11.7539037 11.9304480
## [49] 11.9942902 11.9960151 12.0625362 12.2271753 12.2463941 12.2793742
## [55] 12.3632848 12.5800265 12.6268997 12.6533620 12.9121866 13.0438061
## [61] 13.1270000

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-----------------------------------------------

sort(boxplot.stats(data$Sulfate)$out)

##  [1] 129.0000 180.2067 182.3974 187.1707 187.4241 192.0336 203.4445
205.9351
##  [9] 206.2472 207.8905 209.4711 211.8516 214.4608 217.0006 219.1489
219.5534
## [17] 223.2358 224.2125 225.5166 227.3485 227.6656 439.7879 440.6355
441.5877
## [25] 441.8268 442.7614 444.3757 444.9706 445.3595 445.9384 446.7240
447.4180
## [33] 449.2677 450.9145 455.4512 458.4411 460.1071 462.4742 475.7375
476.5397
## [41] 481.0306

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-----------------------------------------------

sort(boxplot.stats(data$Conductivity)$out)

##  [1] 181.4838 656.9241 657.5704 660.2549 666.6906 669.7251 672.5570
674.4435
##  [9] 695.3695 708.2264 753.3426

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-----------------------------------------------

sort(boxplot.stats(data$Organic_carbon)$out)

##  [1]  2.200000  4.371899  4.466772  4.473092  4.861631  4.902888
4.966862
##  [8]  5.051695  5.159380  5.188466  5.196717  5.218233  5.315287
23.317699
## [15] 23.373265 23.399516 23.514774 23.569645 23.604298 23.667667
```

```
23.917601
## [22] 23.952450 24.755392 27.006707 28.300000

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-------------------------------------------

sort(boxplot.stats(data$Trihalomethanes)$out)

##  [1]   0.738000   8.175876   8.577013  14.343161  15.684877  16.291505
##  [7]  17.000683  17.527765  17.915723  18.015272  18.101222  18.400012
## [13]  19.175175  20.337753  21.355275  22.219327  22.749735  23.075806
## [19]  23.136611 110.431080 110.739299 111.115310 111.595448 112.061027
## [25] 112.412210 112.622733 113.048886 114.034946 114.208671 116.161622
## [31] 118.357275 120.030077 124.000000

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-------------------------------------------

sort(boxplot.stats(data$Turbidity)$out)

##  [1] 1.450000 1.492207 1.496101 1.641515 1.659799 1.680554 1.687625
1.801327
##  [9] 1.812529 1.844372 6.099632 6.204846 6.226580 6.307678 6.357439
6.389161
## [17] 6.494249 6.494749 6.739000

cat("------------------------------------------------------------------
-----------------------------------------------\n")

## ------------------------------------------------------------------
-------------------------------------------
```

Y podemos geenrar un conjunto de datos con los valores intermedios de cada atributo:

```
test <- data
test <- subset(test, (ph < 11.0278799 & ph > 3.1020756) | is.na(ph))
test <- subset(test, Hardness < 276.69976 & Hardness > 117.05731)
test <- subset(test, Solids < 44868.46 & Solids > -4291.62)
test <- subset(test, Chloramines < 11.1016281 & Chloramines > 3.1395527)
test <- subset(test, (Sulfate < 439.7879 & Sulfate > 227.6656) |
is.na(Sulfate))
test <- subset(test, Conductivity < 656.9241 & Conductivity > 181.4838)
test <- subset(test, Organic_carbon < 23.317699 & Organic_carbon >
5.315287)
test <- subset(test, (Trihalomethanes < 110.431080 & Trihalomethanes >
23.136611) | is.na(Trihalomethanes))
test <- subset(test, Turbidity < 6.099632 & Turbidity > 1.844372)
```

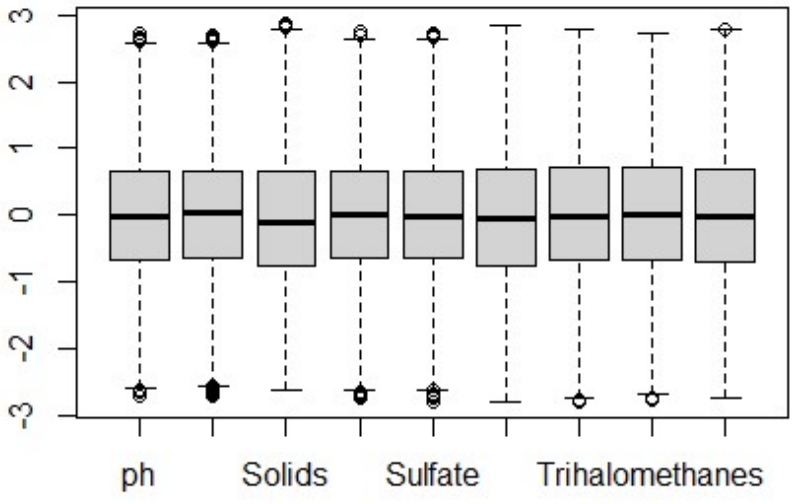
```r
no_ouliers_incomplete_boxplot <- test
head(no_ouliers_incomplete_boxplot)
```

```
##         ph Hardness   Solids Chloramines  Sulfate Conductivity
Organic_carbon
## 1       NA 204.8905 20791.32    7.300212 368.5164     564.3087
10.379783
## 2 3.716080 129.4229 18630.06    6.635246       NA     592.8854
15.180013
## 3 8.099124 224.2363 19909.54    9.275884       NA     418.6062
16.868637
## 4 8.316766 214.3734 22018.42    8.059332 356.8861     363.2665
18.436524
## 5 9.092223 181.1015 17978.99    6.546600 310.1357     398.4108
11.558279
## 6 5.584087 188.3133 28748.69    7.544869 326.6784     280.4679
8.399735
##   Trihalomethanes Turbidity Potability
## 1        86.99097  2.963135          0
## 2        56.32908  4.500656          0
## 3        66.42009  3.055934          0
## 4       100.34167  4.628771          0
## 5        31.99799  4.075075          0
## 6        54.91786  2.559708          0
```

```r
nrow(no_ouliers_incomplete_boxplot)
```

```
## [1] 2957
```

```r
boxplot(scale(no_ouliers_incomplete_boxplot[,1:9]))
```



Aunque no apreciamos apenas diferencias con nuetro método por cuartiles.

A continuación procederemos a la imputación de datos con los conjuntos de datos generados, dando lugar a más conjuntos por cada uno, usaremos dos métodos de imputación para cada conjunto (missForest y KNN):

Imputación de valores nulos con los métodos seleccionados

missForest:

```
set.seed(100)
imp_1 <- missForest(data, verbose=TRUE)
```

```
##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.001839853
##     difference(s): 7.943625e-08
##     time: 5.82 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.001825295
##     difference(s): 1.442821e-08
##     time: 5.6 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.001832885
##     difference(s): 1.518799e-08
##     time: 5.87 seconds
```

```r
mF_complete <- imp_1$ximp
imp_2 <- missForest(no_ouliers_incomplete_Tukey, verbose=TRUE)
```

```
##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.001767359
##     difference(s): 6.414244e-08
##     time: 4.76 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.001755669
##     difference(s): 1.314957e-08
##     time: 4.82 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.001753547
##     difference(s): 1.21459e-08
##     time: 4.75 seconds
##
##   missForest iteration 4 in progress...done!
##     estimated error(s): 0.00176082
##     difference(s): 1.265906e-08
##     time: 4.95 seconds
```

```r
mF_Tukey <- imp_2$ximp
imp_3 <- missForest(no_ouliers_incomplete_sd, verbose=TRUE)
```

```
##   missForest iteration 1 in progress...done!
##     estimated error(s): 0.001802774
##     difference(s): 6.597583e-08
##     time: 5.16 seconds
##
##   missForest iteration 2 in progress...done!
##     estimated error(s): 0.001789161
##     difference(s): 1.338087e-08
##     time: 5.2 seconds
##
##   missForest iteration 3 in progress...done!
##     estimated error(s): 0.001795452
##     difference(s): 1.306673e-08
##     time: 5.27 seconds
##
##   missForest iteration 4 in progress...done!
##     estimated error(s): 0.001801473
##     difference(s): 1.28061e-08
##     time: 5.33 seconds
##
##   missForest iteration 5 in progress...done!
##     estimated error(s): 0.001801995
##     difference(s): 1.237846e-08
##     time: 5.45 seconds
##
```

```
##   missForest iteration 6 in progress...done!
##     estimated error(s): 0.001798555
##     difference(s): 1.280096e-08
##     time: 5.19 seconds

mF_sd <- imp_3$ximp
```

Ahora deberíamos buscar outliers en el conjunto completo, pero también podemos ver los resultados en los demás, vemaos el número de registros que perdemos:

```
nrow(mF_complete)

## [1] 3276

nrow(mF_Tukey)

## [1] 2951

nrow(mF_sd)

## [1] 3161

nrow(remove_outliers_Tukey(mF_complete))

## [1] 2743

nrow(remove_outliers_sd(mF_complete))

## [1] 3126

nrow(remove_outliers_Tukey(mF_Tukey))

## [1] 2662

nrow(remove_outliers_sd(mF_sd))

## [1] 3094
```

Generaremos los conjuntos de datos para comprobar resultados después.

```
mF_complete_Tuk <- remove_outliers_Tukey(mF_complete)

mF_complete_sd <- (mF_complete)

mF_Tukey_Tuk <- remove_outliers_Tukey(mF_Tukey)

mF_sd_sd <- remove_outliers_sd(mF_sd)
```

KNN:

Seguiremos el mismo proceso que para missForest, como en el caso anterior omitiremos el conjunto de datos generado con los boxplots ya que es muy similar a

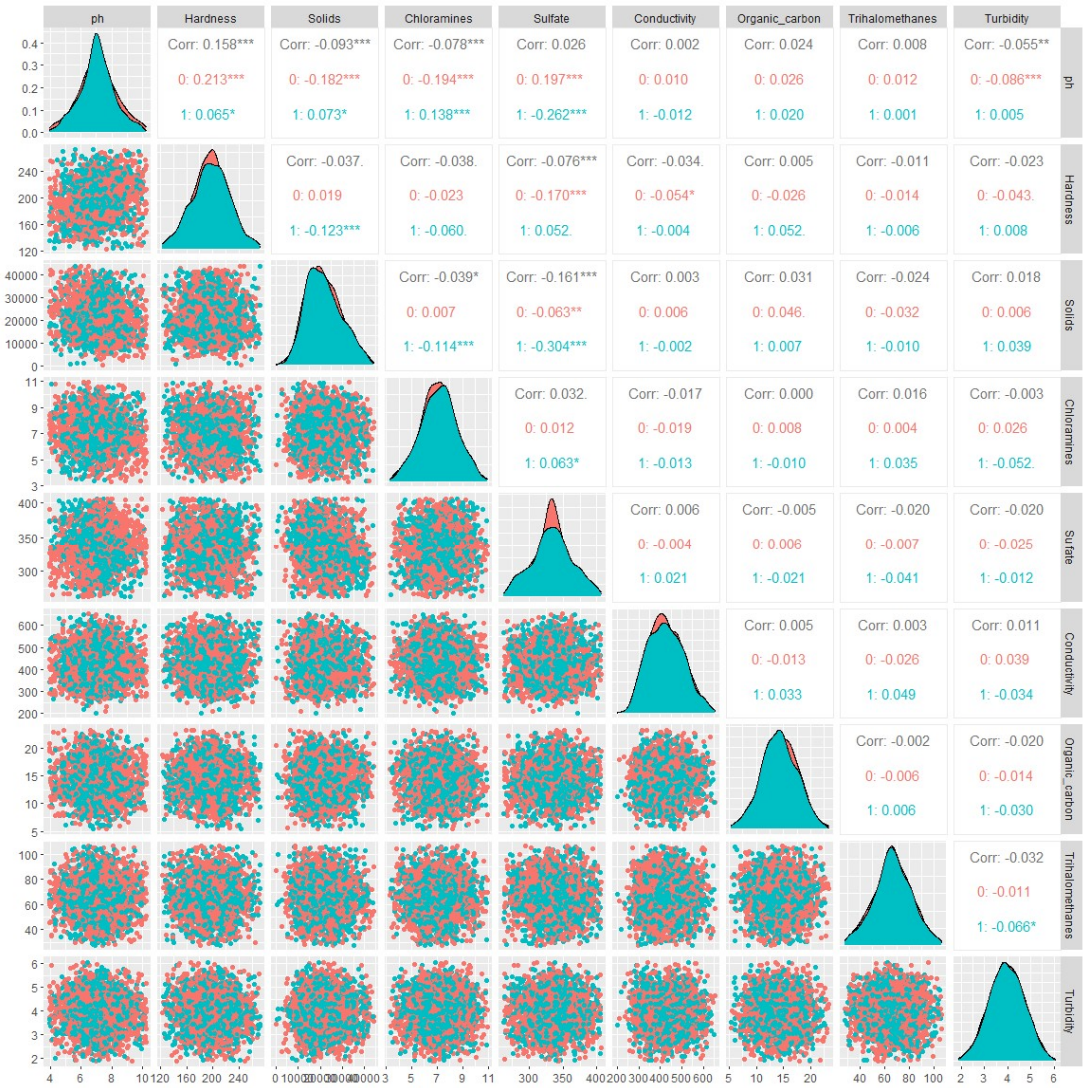
no_ouliers_incomplete_Tukey. También escalaremos los datos para no influir en las distancias:

```
set.seed(100)

data_scaled = as.data.frame(cbind(scale(data[,1:9]), data[,10]))
names(data_scaled)[names(data_scaled) == "V10"] <- "Potability"

data_scaled_tuk =
as.data.frame(cbind(scale(no_ouliers_incomplete_Tukey[,1:9]),
no_ouliers_incomplete_Tukey[,10]))
names(data_scaled_tuk)[names(data_scaled_tuk) == "V10"] <- "Potability"

data_scaled_sd =
as.data.frame(cbind(scale(no_ouliers_incomplete_sd[,1:9]),
no_ouliers_incomplete_sd[,10]))
names(data_scaled_sd)[names(data_scaled_sd) == "V10"] <- "Potability"

imp_1 <- kNN(data_scaled, k=100)
KNN_complete <- imp_1[,1:10]
imp_2 <- kNN(data_scaled_tuk, k=100)
KNN_Tukey <- imp_2[,1:10]
imp_3 <- kNN(data_scaled_sd, k=100)
KNN_sd <- imp_3[,1:10]

nrow(KNN_complete)

## [1] 3276

nrow(KNN_Tukey)

## [1] 2951

nrow(KNN_sd)

## [1] 3161

nrow(remove_outliers_Tukey(KNN_complete))

## [1] 2677

nrow(remove_outliers_sd(KNN_complete))

## [1] 3123

nrow(remove_outliers_Tukey(KNN_Tukey))

## [1] 2596

nrow(remove_outliers_sd(KNN_sd))

## [1] 3094
```

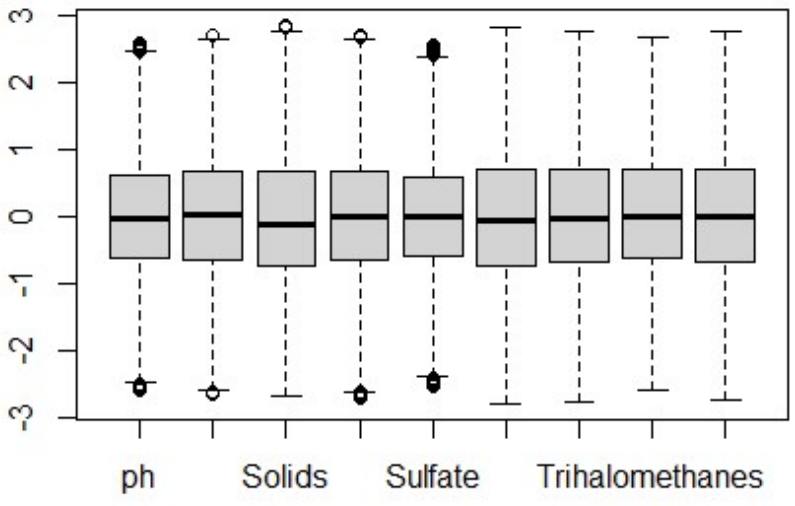
```
KNN_complete_Tuk <- remove_outliers_Tukey(KNN_complete)

KNN_complete_sd <- (KNN_complete)

KNN_Tukey_Tuk <- remove_outliers_Tukey(KNN_Tukey)

KNN_sd_sd <- remove_outliers_sd(KNN_sd)
```

Las distribuciones resultantes son similares en todos los conjuntos y ya hemos visto los boxplots previamente, aportamos el ejemplo del conjunto mF_Tukey_Tuk y KNN_Tukey_Tuk (los dos modelos de imputación):

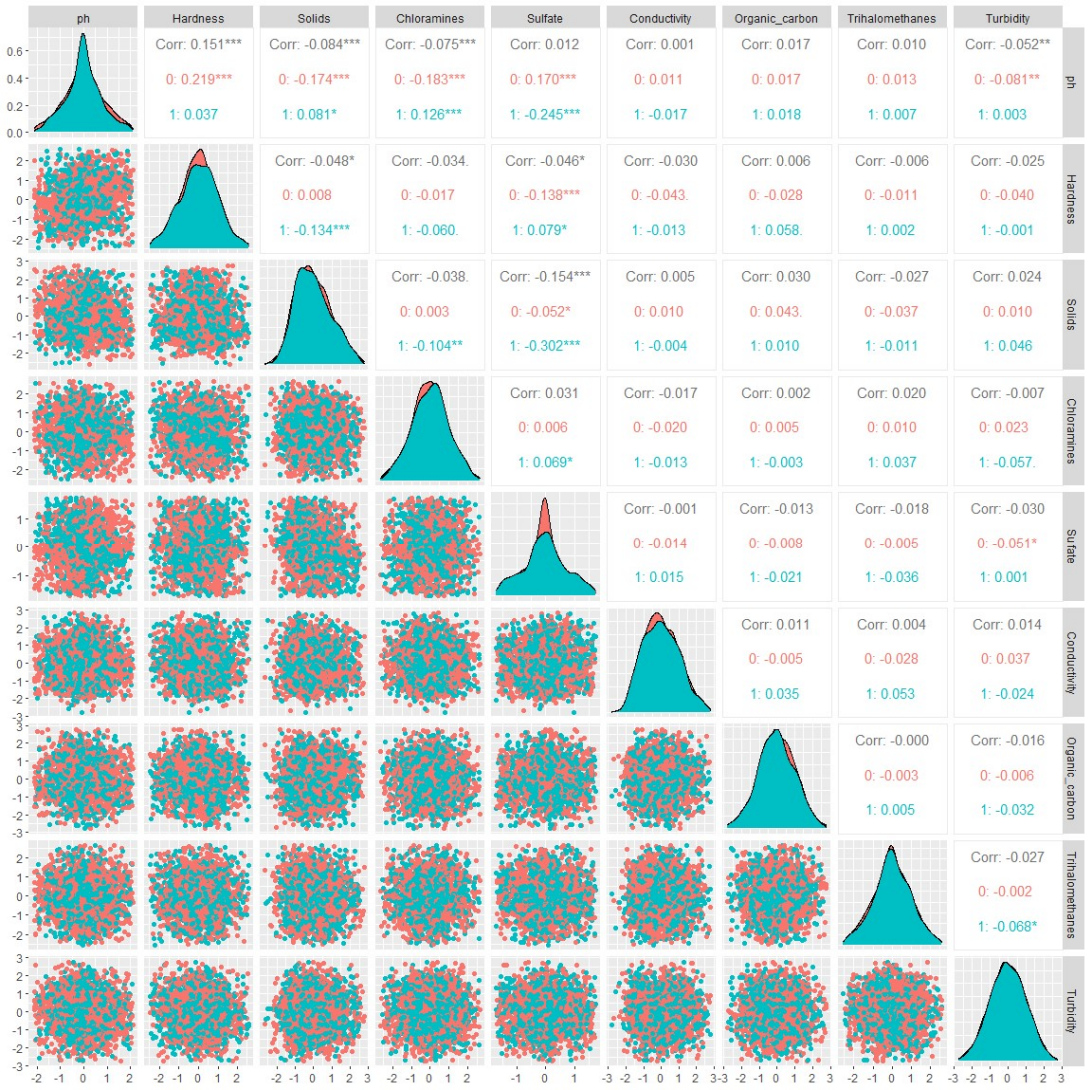
```
library(GGally)
ggpairs(mF_Tukey_Tuk, columns = 1:9,
ggplot2::aes(colour=as.factor(Potability)), progress = FALSE)
```



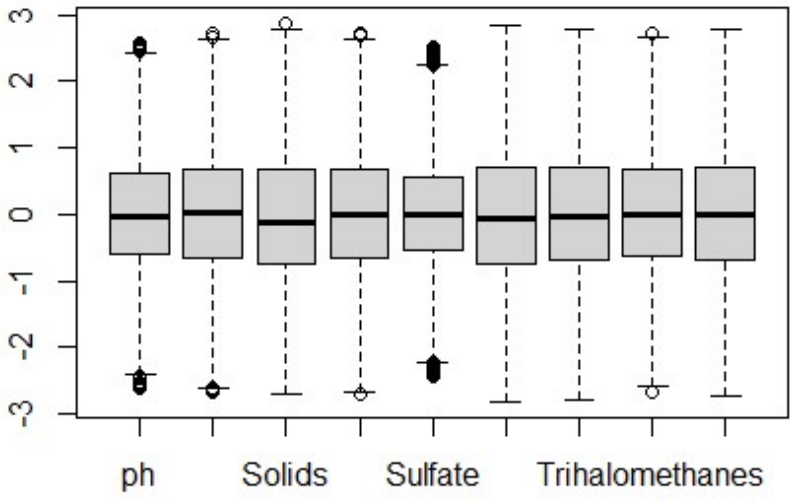

```
boxplot(scale(mF_Tukey_Tuk[,1:9]))
```




```
library(GGally)
ggpairs(KNN_Tukey_Tuk, columns = 1:9,
ggplot2::aes(colour=as.factor(Potability)), progress = FALSE)
```




```
boxplot(scale(KNN_Tukey_Tuk[,1:9]))
```



****** #

Análisis de los datos. ******

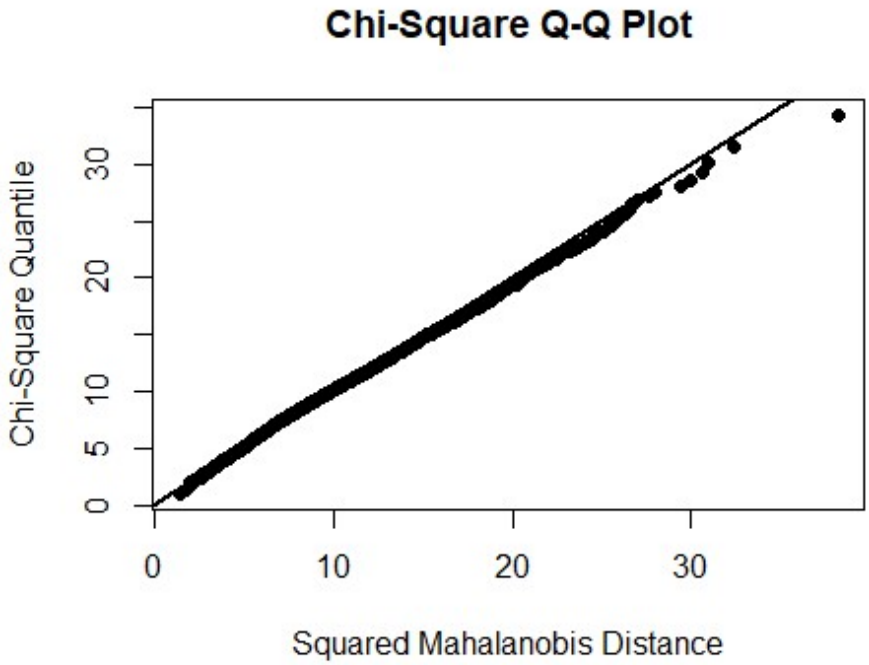
Selección de los grupos de datos que se quieren analizar/comparar (planificación de los análisis a aplicar).

Puesto que en nuestro caso tenemos un único grupo de datos con la etiqueta potabilidad para clasificar las muestras de agua entre las potables y las no potables, no tenemos ningún elemento comparativo además de la potabilidad, por lo que nos centraremos en clasificar el conjunto de datos con distintos algoritmos de clasificación.

Comprobación de la normalidad y homogeneidad de la varianza.

En cuanto al estudio de normalidad, hemos visto anteriormente información sobre los registros completos, estudiando la normalidad de cada atributo y test multivariables, la conclusión fue que no podemos asegurar que las poblaciones son normales, haremos un último ejemplo con uno de los conjuntos con los datos imputados, concretamente, mF_sd_sd, pero ninguno de ellos pasa los test de normalidad, para los atributos Organic_carbon y Turbidity en uno de los conjuntos se acepta la hipotesis de normalidad. No compararemos con los conjuntos generados por KNN porque los resultados son muy similares.

```
library(MVN)
result <- mvn(mF_sd_sd, multivariatePlot = "qq", showOutliers = TRUE)
```




```
result$multivariateNormality
```

```
##              Test        Statistic               p value Result
## 1 Mardia Skewness 1087.73275134228 8.08812009335732e-115     NO
## 2 Mardia Kurtosis 4.07853808522237  4.53197790204474e-05     NO
## 3             MVN             <NA>                   <NA>     NO
```

```
result$univariateNormality
```

```
##            Test        Variable Statistic   p value Normality
## 1  Shapiro-Wilk       ph           0.9945  <0.001      NO
## 2  Shapiro-Wilk    Hardness        0.9973  <0.001      NO
## 3  Shapiro-Wilk     Solids         0.9847  <0.001      NO
## 4  Shapiro-Wilk   Chloramines      0.9987  0.0147      NO
## 5  Shapiro-Wilk     Sulfate        0.9892  <0.001      NO
## 6  Shapiro-Wilk  Conductivity      0.9925  <0.001      NO
## 7  Shapiro-Wilk Organic_carbon     0.9992  0.1594      YES
## 8  Shapiro-Wilk Trihalomethanes    0.9982  0.0016      NO
## 9  Shapiro-Wilk    Turbidity       0.9991  0.0946      YES
## 10 Shapiro-Wilk   Potability       0.6157  <0.001      NO
```

En el estudio de la homocedasticidad, la igualdad de varianzas entre los grupos que se van a comparar, debemos tener en cuenta los datos anteriores, es decir, si no se puede alcanzar cierta seguridad de que las poblaciones que se comparan son de tipo normal, es recomendable recurrir a test que comparen la mediana de la varianza. Para esto optaremos por el test de Fligner-Killeen, con la librería stats. Lo haremos con un

conjunto de datos, los resultados con los conjuntos generados por KNN son muy similares.

```
library(stats)
fligner.test(ph ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  ph by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 40.293, df = 1, p-value = 2.186e-10
```

```
fligner.test(Hardness ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Hardness by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 8.2399, df = 1, p-value = 0.004098
```

```
fligner.test(Solids ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Solids by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 1.6063, df = 1, p-value = 0.205
```

```
fligner.test(Chloramines ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Chloramines by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 5.1396, df = 1, p-value = 0.02339
```

```
fligner.test(Sulfate ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Sulfate by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 50.683, df = 1, p-value = 1.086e-12
```

```
fligner.test(Conductivity ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Conductivity by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 3.8935, df = 1, p-value = 0.04847
```

```
fligner.test(Organic_carbon ~ as.factor(Potability), data=mF_sd_sd)
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Organic_carbon by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 0.098979, df = 1, p-value = 0.7531
```

```
fligner.test(Trihalomethanes ~ as.factor(Potability), data=mF_sd_sd)
```

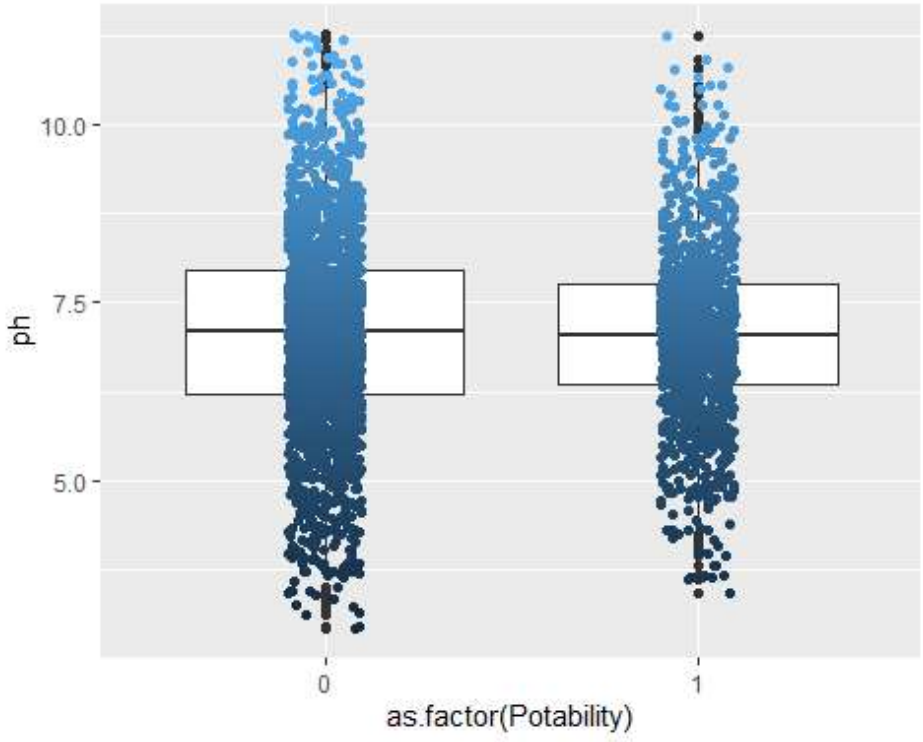
```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Trihalomethanes by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 0.0062732, df = 1, p-value = 0.9369
```

```
fligner.test(Turbidity ~ as.factor(Potability), data=mF_sd_sd)
```

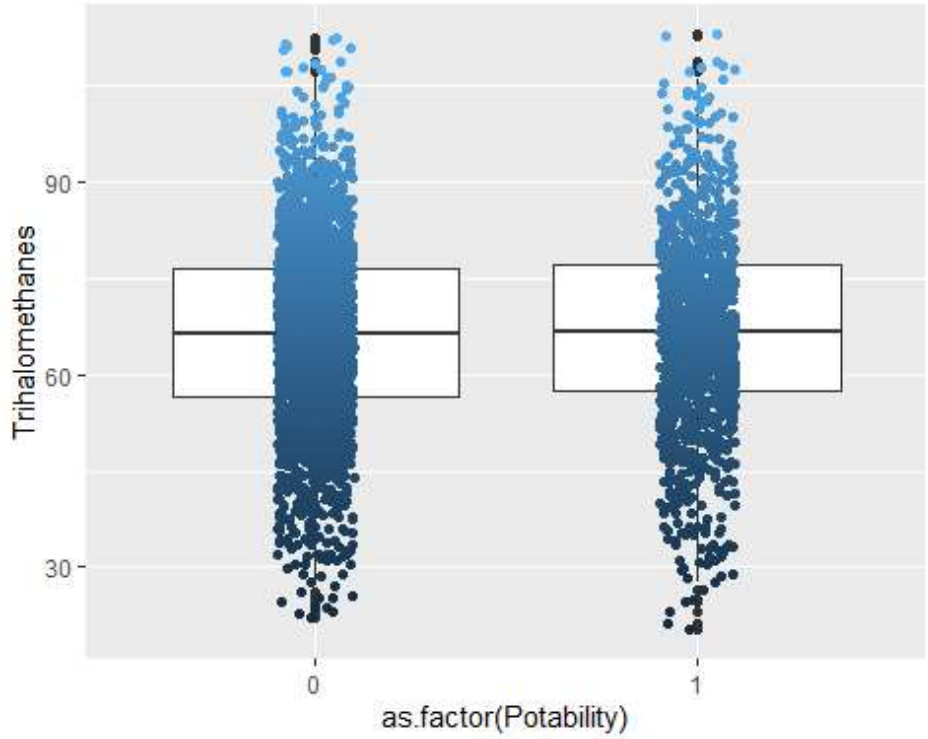
```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Turbidity by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 0.032333, df = 1, p-value = 0.8573
```

Hay formas de visualizarlo, como por ejemplo:

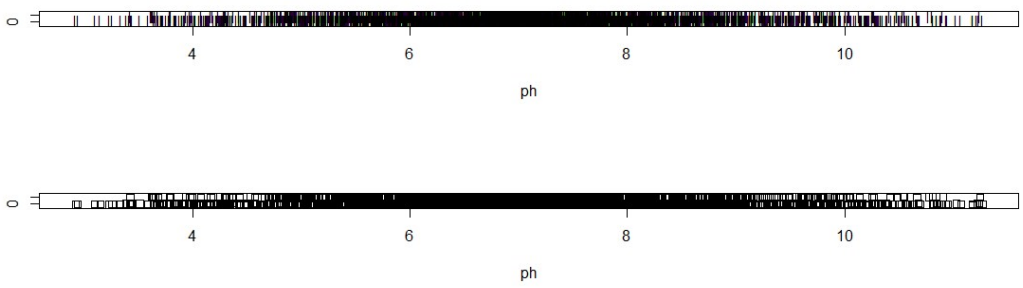
```
ggplot(mF_sd_sd,aes(x=as.factor(Potability),y=ph, col=ph)) +
geom_boxplot() +
  geom_jitter(position=position_jitter(0.1)) + guides(col=FALSE)
```



```
ggplot(mF_sd_sd,aes(x=as.factor(Potability),y=Trihalomethanes,
col=Trihalomethanes)) + geom_boxplot() +
  geom_jitter(position=position_jitter(0.1)) + guides(col=FALSE)
```

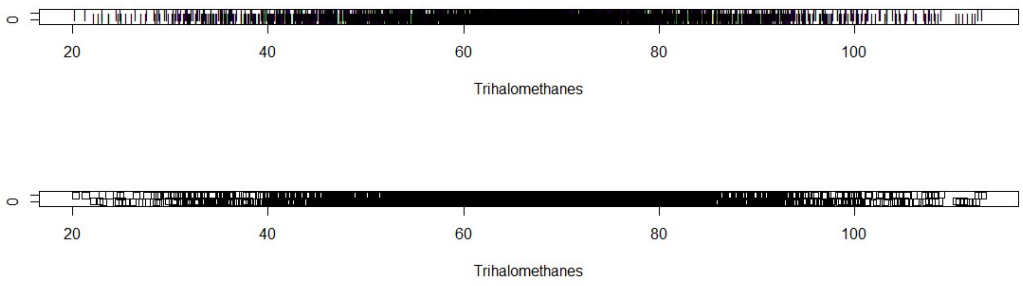



```
par(mfrow=c(2,1))                  # enable two panels per plot
  stripchart(ph ~ Potability, data=mF_sd_sd, pch="|", ylim=c(.5, 2.5))
# narrow plotting symbol
  stripchart(ph ~ Potability, data=mF_sd_sd, meth="j", ylim=c(.5, 2.5))
# jittered to mitigate overplotting
```




```
par(mfrow=c(1,1))                  # return to single-panel plotting

par(mfrow=c(2,1))                  # enable two panels per plot
  stripchart(Trihalomethanes  ~ Potability, data=mF_sd_sd, pch="|",
ylim=c(.5, 2.5))    # narrow plotting symbol
```

```
  stripchart(Trihalomethanes  ~ Potability, data=mF_sd_sd, meth="j",
ylim=c(.5, 2.5))  # jittered to mitigate overplotting
```

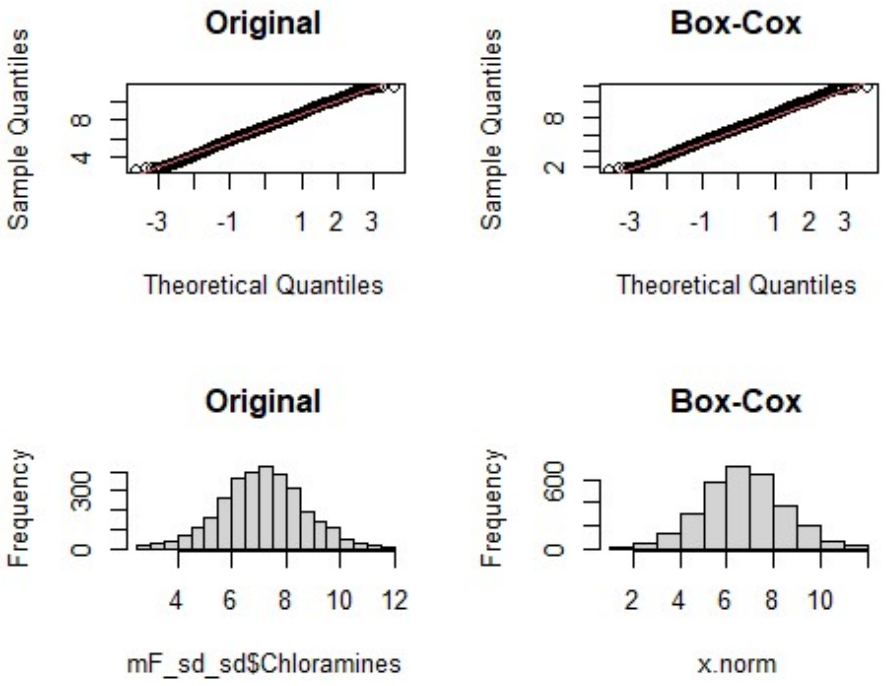



```
par(mfrow=c(1,1))
```

Podemos recurrir a transformaciones para tratar de corregirlo, en este caso aplicaremos transformación Box-Cox para una única variable, tal como vemos en la documentación de la asignatura, probemos con un atributo con p-value más cercano a 0.05 (dentro de las posibilidades de los datos), a ver los resultados, probaremos con Chloramines, que nos ha dado un p-value de 0.0147 en saphiro-test.

```
library(DescTools)

##
## Attaching package: 'DescTools'

## The following objects are masked from 'package:caret':
##
##     MAE, RMSE

## The following object is masked from 'package:foreach':
##
##     %:%

## The following objects are masked from 'package:QuantPsyc':
##
##     Kurt, Skew

## The following objects are masked from 'package:Hmisc':
##
##     %nin%, Label, Mean, Quantile

x.norm <- BoxCox(mF_sd_sd$Chloramines, lambda =
BoxCoxLambda(mF_sd_sd$Chloramines))

par(mfrow=c(2,2))
qqnorm(mF_sd_sd$Chloramines, main="Original")
qqline(mF_sd_sd$Chloramines,col=2)
qqnorm(x.norm, main="Box-Cox")
qqline(x.norm,col=2)
```

```r
hist(mF_sd_sd$Chloramines,main="Original")
hist(x.norm, main="Box-Cox")
```



Y comprobamos los resultados con los test de normalidad y homocedasticidad.

```r
shapiro.test(mF_sd_sd$Chloramines)

##
##  Shapiro-Wilk normality test
##
## data:  mF_sd_sd$Chloramines
## W = 0.99868, p-value = 0.01473

shapiro.test(x.norm)

##
##  Shapiro-Wilk normality test
##
## data:  x.norm
## W = 0.99875, p-value = 0.02112

fligner.test(Chloramines ~ as.factor(Potability), data=mF_sd_sd)

##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  Chloramines by as.factor(Potability)
## Fligner-Killeen:med chi-squared = 5.1396, df = 1, p-value = 0.02339
```

```
fligner.test(x.norm ~ as.factor(mF_sd_sd$Potability))
```

```
##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  x.norm by as.factor(mF_sd_sd$Potability)
## Fligner-Killeen:med chi-squared = 5.1841, df = 1, p-value = 0.02279
```

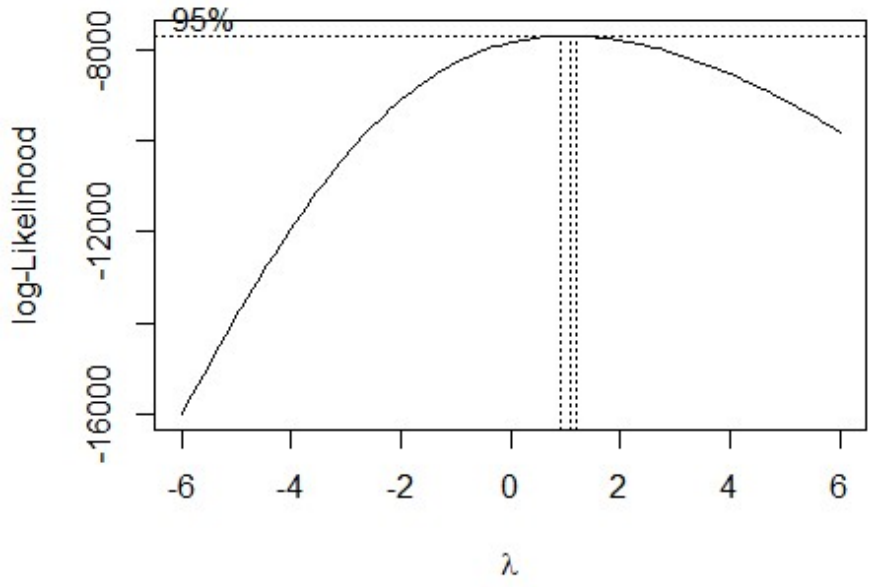
Es posible hacerlo con la librería MASS, pero los resultados son un poco diefrentes, incluso peores, no estamos limitando un rango de lambda, en cualquier caso se trata de una prueba:

```
library(MASS)
library(rcompanion)
```

```
##
## Attaching package: 'rcompanion'
```

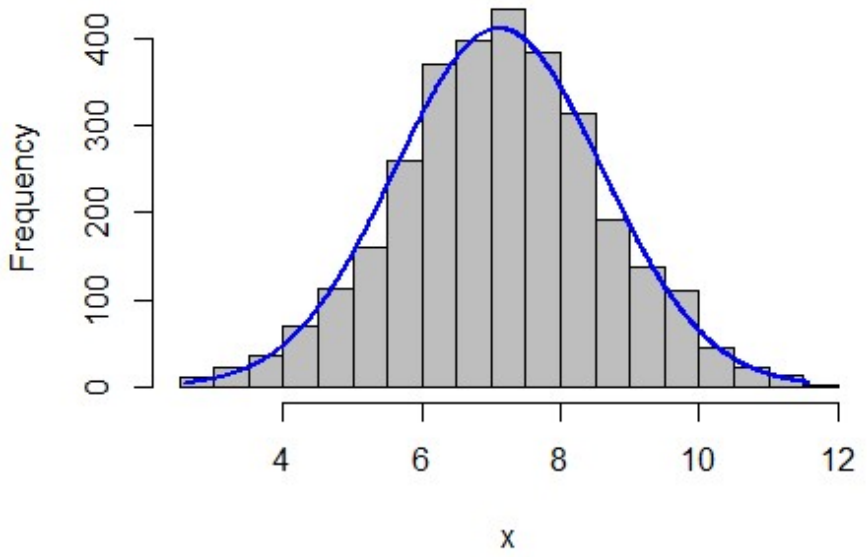
```
## The following object is masked from 'package:Metrics':
##
##     accuracy
```

```
Box = boxcox(mF_sd_sd$Chloramines ~ 1,
             lambda = seq(-6,6,0.1)
             )
```

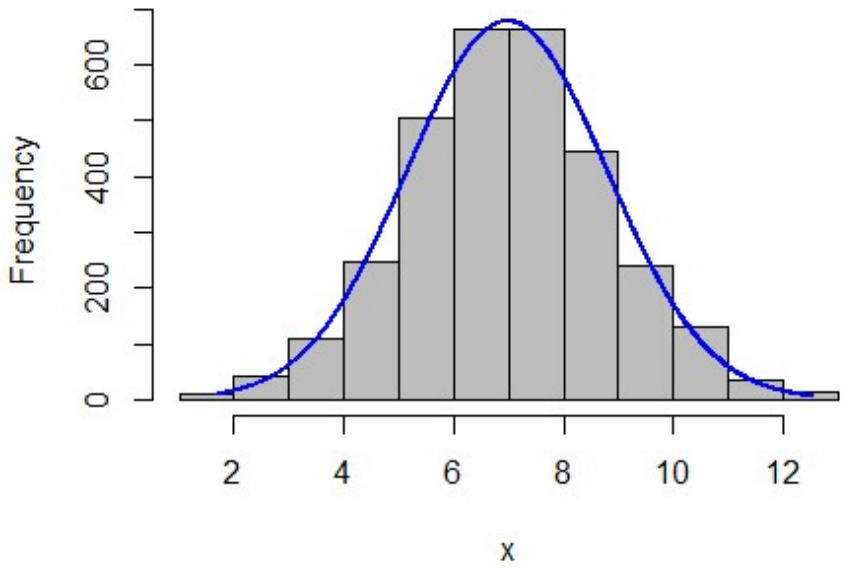



```
Cox = data.frame(Box$x, Box$y)
```

```
Cox2 = Cox[with(Cox, order(-Cox$Box.y)),]

Cox2[1,]

##    Box.x     Box.y
## 72   1.1 -7693.982

lambda = Cox2[1, "Box.x"]

T_box = (mF_sd_sd$Chloramines ^ lambda - 1)/lambda

plotNormalHistogram(mF_sd_sd$Chloramines)
```




```
plotNormalHistogram(T_box)
```




```
shapiro.test(T_box)

##
##  Shapiro-Wilk normality test
##
## data:  T_box
## W = 0.9987, p-value = 0.01657

fligner.test(T_box ~ as.factor(mF_sd_sd$Potability))

##
##  Fligner-Killeen test of homogeneity of variances
##
## data:  T_box by as.factor(mF_sd_sd$Potability)
## Fligner-Killeen:med chi-squared = 5.2376, df = 1, p-value = 0.0221
```

Existen procesos que nos podrían ayudar a intentar corregir tanto la normalidad como la homocedasticidad (no siempre es posible), pero viendo los resultados de los diferentes test hechos hasta ahora, y teniendo en cuenta que no debería afectar mucho a los modelos de clasificación que tenemos pensado aplicar, no merece la pena profundizar mucho más en esto.

Aplicación de pruebas estadísticas para comparar los grupos de datos.

Lo primero que debemos hacer es dividir los conjuntos de datos en dos, una parte para train y otra para test, los conjuntos resultantes son:

mF_complete_Tuk mF_complete_sd mF_Tukey_Tuk mF_sd_sd

KNN_complete_Tuk KNN_complete_sd KNN_Tukey_Tuk KNN_sd_sd

```
library(rminer)
set.seed(100)

h<-holdout(mF_complete_Tuk$Potability,ratio=2/3,mode="stratified")
mF_complete_Tuk_train<-mF_complete_Tuk[h$tr,]
mF_complete_Tuk_test<-mF_complete_Tuk[h$ts,]
print(table(mF_complete_Tuk_train$Potability))

##
##    0    1
## 1125  703

print(table(mF_complete_Tuk_test$Potability))

##
##   0   1
## 597 318

h<-holdout(mF_complete_sd$Potability,ratio=2/3,mode="stratified")
mF_complete_sd_train<-mF_complete_sd[h$tr,]
mF_complete_sd_test<-mF_complete_sd[h$ts,]
print(table(mF_complete_sd_train$Potability))

##
##    0    1
## 1353  831

print(table(mF_complete_sd_test$Potability))

##
##   0   1
## 645 447

h<-holdout(mF_Tukey_Tuk$Potability,ratio=2/3,mode="stratified")
mF_Tukey_Tuk_train<-mF_Tukey_Tuk[h$tr,]
mF_Tukey_Tuk_test<-mF_Tukey_Tuk[h$ts,]
print(table(mF_Tukey_Tuk_train$Potability))

##
##    0    1
## 1105  669
```

```r
print(table(mF_Tukey_Tuk_test$Potability))
```

```
##
##   0   1
## 563 325
```

```r
h<-holdout(mF_sd_sd$Potability,ratio=2/3,mode="stratified")
mF_sd_sd_train<-mF_sd_sd[h$tr,]
mF_sd_sd_test<-mF_sd_sd[h$ts,]
print(table(mF_sd_sd_train$Potability))
```

```
##
##    0    1
## 1299  763
```

```r
print(table(mF_sd_sd_test$Potability))
```

```
##
##   0   1
## 616 416
```

```r
h<-holdout(KNN_complete_Tuk$Potability,ratio=2/3,mode="stratified")
KNN_complete_Tuk_train<-KNN_complete_Tuk[h$tr,]
KNN_complete_Tuk_test<-KNN_complete_Tuk[h$ts,]
print(table(KNN_complete_Tuk_train$Potability))
```

```
##
##    0    1
## 1118  666
```

```r
print(table(KNN_complete_Tuk_test$Potability))
```

```
##
##   0   1
## 558 335
```

```r
h<-holdout(KNN_complete_sd$Potability,ratio=2/3,mode="stratified")
KNN_complete_sd_train<-KNN_complete_sd[h$tr,]
KNN_complete_sd_test<-KNN_complete_sd[h$ts,]
print(table(KNN_complete_sd_train$Potability))
```

```
##
##    0    1
## 1315  869
```

```r
print(table(KNN_complete_sd_test$Potability))
```

```
##
##   0   1
## 683 409
```

```r
h<-holdout(KNN_Tukey_Tuk$Potability,ratio=2/3,mode="stratified")
KNN_Tukey_Tuk_train<-KNN_Tukey_Tuk[h$tr,]
```

```r
KNN_Tukey_Tuk_test<-KNN_Tukey_Tuk[h$ts,]
print(table(KNN_Tukey_Tuk_train$Potability))
```

```
##
##    0    1
## 1100  630
```

```r
print(table(KNN_Tukey_Tuk_test$Potability))
```

```
##
##   0   1
## 523 343
```

```r
h<-holdout(KNN_sd_sd$Potability,ratio=2/3,mode="stratified")
KNN_sd_sd_train<-KNN_sd_sd[h$tr,]
KNN_sd_sd_test<-KNN_sd_sd[h$ts,]
print(table(KNN_sd_sd_train$Potability))
```

```
##
##    0    1
## 1254  808
```

```r
print(table(KNN_sd_sd_test$Potability))
```

```
##
##   0   1
## 661 371
```

```r
library(caret)
library(parallel)
library(doParallel)

set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

model_1 <- caret::train(mF_complete_Tuk_train[, -10],
as.factor(mF_complete_Tuk_train$Potability),
                 method = "rf", trControl = caret::trainControl(method =
"cv", p = 0.8, number = 5))

stopCluster(cl)
model_1
```

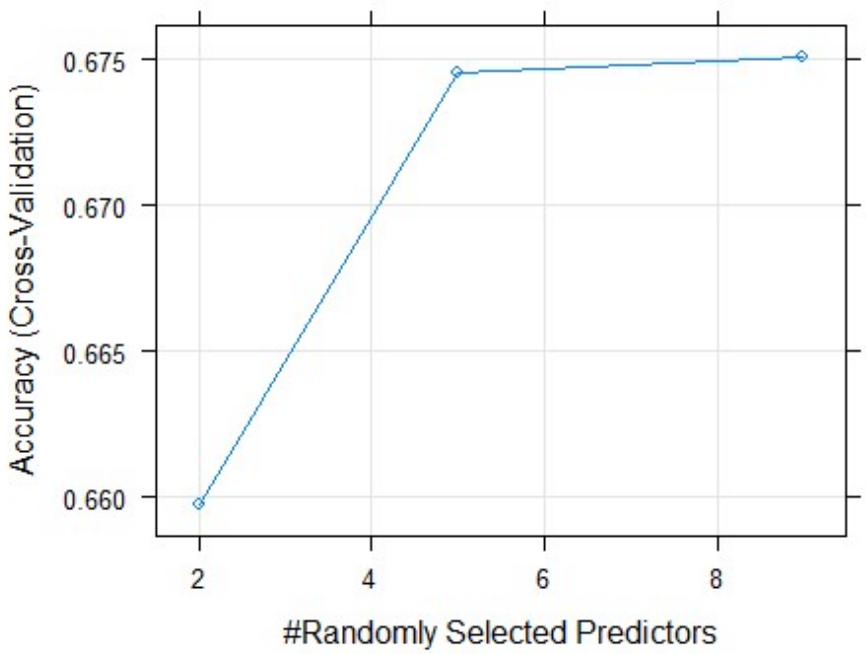
```
## Random Forest
##
## 1828 samples
##    9 predictor
##    2 classes: '0', '1'
##
## No pre-processing
```

```
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 1463, 1462, 1462, 1462, 1463
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.6597380  0.2053890
##   5     0.6745176  0.2591831
##   9     0.6750670  0.2657547
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 9.
```

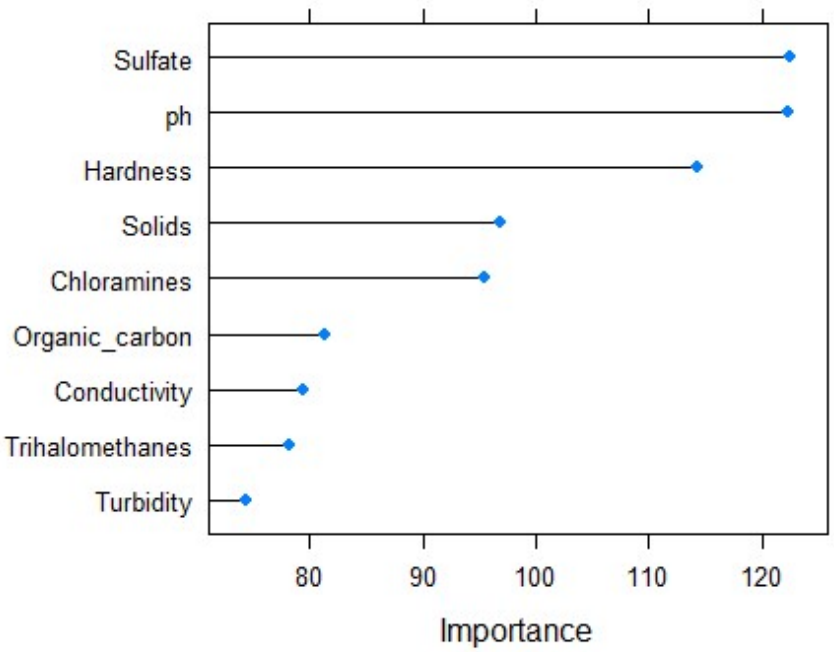
```
predict <- predict(model_1, mF_complete_Tuk_test)
confusionMatrix(predict, as.factor(mF_complete_Tuk_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 516 170
##          1  81 148
##
##                Accuracy : 0.7257
##                  95% CI : (0.6955, 0.7544)
##     No Information Rate : 0.6525
##     P-Value [Acc > NIR] : 1.294e-06
##
##                   Kappa : 0.3528
##
##  Mcnemar's Test P-Value : 2.784e-08
##
##             Sensitivity : 0.8643
##             Specificity : 0.4654
##          Pos Pred Value : 0.7522
##          Neg Pred Value : 0.6463
##              Prevalence : 0.6525
##          Detection Rate : 0.5639
##    Detection Prevalence : 0.7497
##       Balanced Accuracy : 0.6649
##
##        'Positive' Class : 0
##
```

```
plot(model_1)
```




```
plot(varImp(model_1, scale = FALSE))
```




```
library(randomForest)
```

```r
set.seed(100)

model_1.1<-
randomForest(as.factor(Potability)~.,mF_complete_Tuk_train,ntree=10000)
model_1.1
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
mF_complete_Tuk_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 31.56%
## Confusion matrix:
##     0   1 class.error
## 0 999 126   0.1120000
## 1 451 252   0.6415363
```

```r
predict_rf_2 <- predict(model_1.1, mF_complete_Tuk_test)
cm <- confusionMatrix(predict_rf_2,
as.factor(mF_complete_Tuk_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 528 187
##          1  69 131
##
##                Accuracy : 0.7202
##                  95% CI : (0.6899, 0.7491)
##     No Information Rate : 0.6525
##     P-Value [Acc > NIR] : 7.114e-06
##
##                   Kappa : 0.3245
##
##  Mcnemar's Test P-Value : 2.622e-13
##
##             Sensitivity : 0.8844
##             Specificity : 0.4119
##          Pos Pred Value : 0.7385
##          Neg Pred Value : 0.6550
##              Prevalence : 0.6525
##          Detection Rate : 0.5770
##    Detection Prevalence : 0.7814
##       Balanced Accuracy : 0.6482
##
```

```
##        'Positive' Class : 0
##

overall <- cm$overall
overall.accuracy <- overall['Accuracy']
accuracy_data <- data.frame("Nombre"="model_1",
"accuracy"=c(overall.accuracy))
accuracy_data

##           Nombre  accuracy
## Accuracy model_1 0.7202186

set.seed(100)

model_2<-
randomForest(as.factor(Potability)~.,mF_complete_sd_train,ntree=10000)
model_2

##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
mF_complete_sd_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 27.24%
## Confusion matrix:
##      0   1 class.error
## 0 1220 133  0.09830007
## 1  462 369  0.55595668

predict_rf <- predict(model_2, mF_complete_sd_test)
cm <- confusionMatrix(predict_rf,
as.factor(mF_complete_sd_test$Potability))
cm

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 580 276
##          1  65 171
##
##                Accuracy : 0.6877
##                  95% CI : (0.6593, 0.7151)
##     No Information Rate : 0.5907
##     P-Value [Acc > NIR] : 2.139e-11
##
##                   Kappa : 0.3038
##
```

```
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8992
##             Specificity : 0.3826
##          Pos Pred Value : 0.6776
##          Neg Pred Value : 0.7246
##              Prevalence : 0.5907
##          Detection Rate : 0.5311
##    Detection Prevalence : 0.7839
##       Balanced Accuracy : 0.6409
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_2", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```
set.seed(100)
```

```
model_3<-
randomForest(as.factor(Potability)~.,mF_Tukey_Tuk_train,ntree=10000)
model_3
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
mF_Tukey_Tuk_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 31.79%
## Confusion matrix:
##     0   1 class.error
## 0 991 114   0.1031674
## 1 450 219   0.6726457
```

```
predict_rf <- predict(model_3, mF_Tukey_Tuk_test)
cm <- confusionMatrix(predict_rf,
as.factor(mF_Tukey_Tuk_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 511 209
##          1  52 116
##
```

```
##                Accuracy : 0.7061
##                  95% CI : (0.6749, 0.7359)
##     No Information Rate : 0.634
##     P-Value [Acc > NIR] : 3.526e-06
##
##                   Kappa : 0.2946
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9076
##             Specificity : 0.3569
##          Pos Pred Value : 0.7097
##          Neg Pred Value : 0.6905
##              Prevalence : 0.6340
##          Detection Rate : 0.5755
##    Detection Prevalence : 0.8108
##       Balanced Accuracy : 0.6323
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_3", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```
set.seed(100)
```

```
model_4<-randomForest(as.factor(Potability)~.,mF_sd_sd_train,ntree=10000)
model_4
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
mF_sd_sd_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 28.42%
## Confusion matrix:
##      0   1 class.error
## 0 1168 131   0.1008468
## 1  455 308   0.5963303
```

```
predict_rf <- predict(model_4, mF_sd_sd_test)
cm <- confusionMatrix(predict_rf, as.factor(mF_sd_sd_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
```

```
##           Reference
## Prediction   0   1
##          0 525 258
##          1  91 158
##
##                Accuracy : 0.6618
##                  95% CI : (0.632, 0.6907)
##     No Information Rate : 0.5969
##     P-Value [Acc > NIR] : 1.015e-05
##
##                   Kappa : 0.2483
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8523
##             Specificity : 0.3798
##          Pos Pred Value : 0.6705
##          Neg Pred Value : 0.6345
##              Prevalence : 0.5969
##          Detection Rate : 0.5087
##    Detection Prevalence : 0.7587
##       Balanced Accuracy : 0.6160
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_4", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```
set.seed(100)
```

```
model_5<-
randomForest(as.factor(Potability)~.,KNN_complete_Tuk_train,ntree=10000)
model_5
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
KNN_complete_Tuk_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 32.06%
## Confusion matrix:
##     0   1 class.error
## 0 999 119   0.1064401
## 1 453 213   0.6801802
```

```r
predict_rf <- predict(model_5, KNN_complete_Tuk_test)
cm <- confusionMatrix(predict_rf,
as.factor(KNN_complete_Tuk_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 509 214
##          1  49 121
##
##                Accuracy : 0.7055
##                  95% CI : (0.6744, 0.7352)
##     No Information Rate : 0.6249
##     P-Value [Acc > NIR] : 2.533e-07
##
##                   Kappa : 0.3032
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9122
##             Specificity : 0.3612
##          Pos Pred Value : 0.7040
##          Neg Pred Value : 0.7118
##              Prevalence : 0.6249
##          Detection Rate : 0.5700
##    Detection Prevalence : 0.8096
##       Balanced Accuracy : 0.6367
##
##        'Positive' Class : 0
##
```

```r
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_5", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```r
set.seed(100)

model_6<-
randomForest(as.factor(Potability)~.,KNN_complete_sd_train,ntree=10000,
mtry=3)
model_6
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
KNN_complete_sd_train,      ntree = 10000, mtry = 3)
##                Type of random forest: classification
##                      Number of trees: 10000
```

```
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 30.54%
## Confusion matrix:
##      0   1 class.error
## 0 1139 176   0.1338403
## 1  491 378   0.5650173
```

```
predict_rf <- predict(model_6, KNN_complete_sd_test)
cm <- confusionMatrix(predict_rf,
as.factor(KNN_complete_sd_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 614 231
##          1  69 178
##
##                Accuracy : 0.7253
##                  95% CI : (0.6978, 0.7516)
##     No Information Rate : 0.6255
##     P-Value [Acc > NIR] : 2.05e-12
##
##                   Kappa : 0.363
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8990
##             Specificity : 0.4352
##          Pos Pred Value : 0.7266
##          Neg Pred Value : 0.7206
##              Prevalence : 0.6255
##          Detection Rate : 0.5623
##    Detection Prevalence : 0.7738
##       Balanced Accuracy : 0.6671
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_6", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```
set.seed(100)
```

```
model_7<-
randomForest(as.factor(Potability)~.,KNN_Tukey_Tuk_train,ntree=10000)
model_7
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
KNN_Tukey_Tuk_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 29.83%
## Confusion matrix:
##      0   1 class.error
## 0 1001  99   0.0900000
## 1  417 213   0.6619048
```

```
predict_rf <- predict(model_7, KNN_Tukey_Tuk_test)
cm <- confusionMatrix(predict_rf,
as.factor(KNN_Tukey_Tuk_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 479 229
##          1  44 114
##
##                Accuracy : 0.6848
##                  95% CI : (0.6526, 0.7156)
##     No Information Rate : 0.6039
##     P-Value [Acc > NIR] : 4.901e-07
##
##                   Kappa : 0.2736
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9159
##             Specificity : 0.3324
##          Pos Pred Value : 0.6766
##          Neg Pred Value : 0.7215
##              Prevalence : 0.6039
##          Detection Rate : 0.5531
##    Detection Prevalence : 0.8176
##       Balanced Accuracy : 0.6241
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_7", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

```r
set.seed(100)

model_8<-
randomForest(as.factor(Potability)~.,KNN_sd_sd_train,ntree=10000)
model_8
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
KNN_sd_sd_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 29.29%
## Confusion matrix:
##      0   1 class.error
## 0 1105 149   0.1188198
## 1  455 353   0.5631188
```

```r
predict_rf <- predict(model_8, KNN_sd_sd_test)
cm <- confusionMatrix(predict_rf, as.factor(KNN_sd_sd_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 585 214
##          1  76 157
##
##                Accuracy : 0.719
##                  95% CI : (0.6905, 0.7462)
##     No Information Rate : 0.6405
##     P-Value [Acc > NIR] : 5.252e-08
##
##                   Kappa : 0.3356
##
##  Mcnemar's Test P-Value : 8.630e-16
##
##             Sensitivity : 0.8850
##             Specificity : 0.4232
##          Pos Pred Value : 0.7322
##          Neg Pred Value : 0.6738
##              Prevalence : 0.6405
##          Detection Rate : 0.5669
##    Detection Prevalence : 0.7742
##       Balanced Accuracy : 0.6541
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_8", "accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

Probamos eliminando outliers unicamente antes de imputar, los valores mejoran respecto a la eliminación de outliers 2 veces.

```
h<-holdout(mF_Tukey$Potability,ratio=2/3,mode="stratified")
mF_Tukey_train<-mF_Tukey[h$tr,]
mF_Tukey_test<-mF_Tukey[h$ts,]
print(table(mF_Tukey_train$Potability))

##
##    0    1
## 1220  747

print(table(mF_Tukey_test$Potability))

##
##   0   1
## 604 380

h<-holdout(mF_sd$Potability,ratio=2/3,mode="stratified")
mF_sd_train<-mF_sd[h$tr,]
mF_sd_test<-mF_sd[h$ts,]
print(table(mF_sd_train$Potability))

##
##    0    1
## 1296  811

print(table(mF_sd_test$Potability))

##
##   0   1
## 654 400

set.seed(100)

model_test_1<-
randomForest(as.factor(Potability)~.,mF_Tukey_train,ntree=10000)
model_test_1

##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data =
mF_Tukey_train,      ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
```

```
##         OOB estimate of  error rate: 29.84%
## Confusion matrix:
##      0   1 class.error
## 0 1106 114  0.09344262
## 1  473 274  0.63319946
```

```
predict_rf <- predict(model_test_1, mF_Tukey_test)
confusionMatrix(predict_rf, as.factor(mF_Tukey_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 550 234
##          1  54 146
##
##                Accuracy : 0.7073
##                  95% CI : (0.6778, 0.7356)
##     No Information Rate : 0.6138
##     P-Value [Acc > NIR] : 5.268e-10
##
##                   Kappa : 0.3232
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.9106
##             Specificity : 0.3842
##          Pos Pred Value : 0.7015
##          Neg Pred Value : 0.7300
##              Prevalence : 0.6138
##          Detection Rate : 0.5589
##    Detection Prevalence : 0.7967
##       Balanced Accuracy : 0.6474
##
##        'Positive' Class : 0
##
```

```
set.seed(100)

model_test_2<-
randomForest(as.factor(Potability)~.,mF_sd_train,ntree=10000)
model_test_2
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ ., data = mF_sd_train,
ntree = 10000)
##                Type of random forest: classification
##                      Number of trees: 10000
## No. of variables tried at each split: 3
##
```

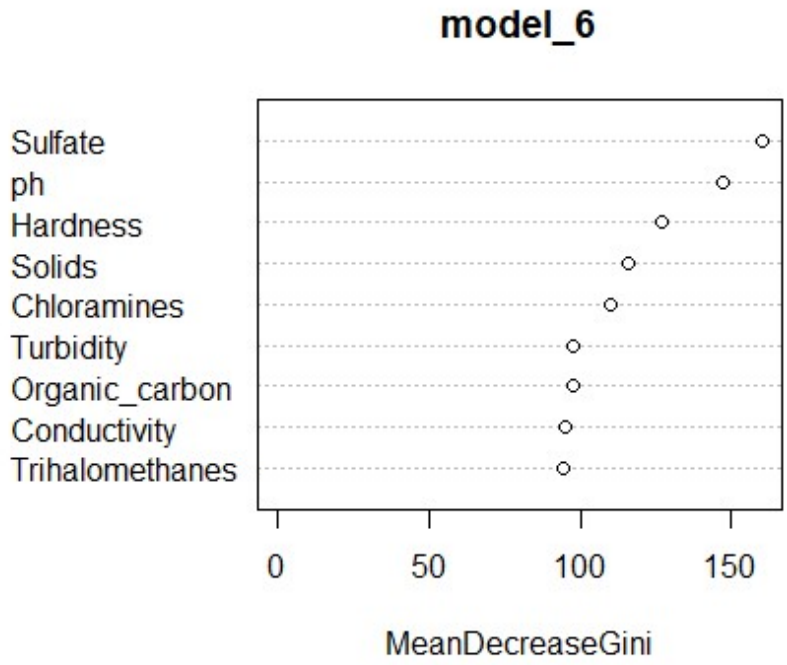
```
##         OOB estimate of  error rate: 31.37%
## Confusion matrix:
##      0   1 class.error
## 0 1138 158   0.1219136
## 1  503 308   0.6202219
```

```
predict_rf <- predict(model_test_2, mF_sd_test)
confusionMatrix(predict_rf, as.factor(mF_sd_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 586 228
##          1  68 172
##
##                Accuracy : 0.7192
##                  95% CI : (0.691, 0.7461)
##     No Information Rate : 0.6205
##     P-Value [Acc > NIR] : 9.946e-12
##
##                   Kappa : 0.3535
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.8960
##             Specificity : 0.4300
##          Pos Pred Value : 0.7199
##          Neg Pred Value : 0.7167
##              Prevalence : 0.6205
##          Detection Rate : 0.5560
##    Detection Prevalence : 0.7723
##       Balanced Accuracy : 0.6630
##
##        'Positive' Class : 0
##
```

Model 6 ha sido el más prometedor, probemos a elegir las variables más importantes

```
varImpPlot(model_6)
```




```
KNN_complete_sd_train$Potability <-
as.factor(KNN_complete_sd_train$Potability)
KNN_complete_sd_test$Potability <-
as.factor(KNN_complete_sd_test$Potability)
```

A continuación vamos a tunear el modelo en base a distintos valores de los parámetros permitidos en este tipo de modelos. Probaremos con distintos valores mediante las librerías ranger y tidymodels de forma que podamos ver cuáles son los valores más interesantes de los parámetros para aplicar este tipo de modelos.

Además, utilizaremos paraller y doParallel para tratar de reducir los tiempos de ejecución debido al alto volumen de combinaciones posibles que surgen de la cantidad de valores de los parámetros.

```
library(ranger)

##
## Attaching package: 'ranger'

## The following object is masked from 'package:randomForest':
##
##     importance

library(tidymodels)

## Registered S3 method overwritten by 'tune':
##   method                    from
##   required_pkgs.model_spec parsnip
```

```
## -- Attaching packages --------------------------------------
tidymodels 0.1.3 --

## v broom        0.7.6      v rsample      0.1.0
## v dials        0.0.9      v tune         0.1.5
## v infer        0.5.4      v workflows    0.2.2
## v modeldata    0.1.0      v workflowsets 0.0.2
## v parsnip      0.1.6      v yardstick    0.0.8
## v recipes      0.1.16

## -- Conflicts -----------------------------------------
tidymodels_conflicts() --
## x purrr::accumulate()      masks foreach::accumulate()
## x yardstick::accuracy()    masks rcompanion::accuracy(),
Metrics::accuracy()
## x dplyr::arrange()         masks plyr::arrange()
## x randomForest::combine()  masks dplyr::combine()
## x purrr::compact()         masks plyr::compact()
## x mi::complete()           masks tidyr::complete(), mice::complete()
## x dplyr::count()           masks plyr::count()
## x scales::discard()        masks purrr::discard()
## x Matrix::expand()         masks tidyr::expand()
## x dplyr::failwith()        masks plyr::failwith()
## x mice::filter()           masks dplyr::filter(), stats::filter()
## x .GlobalEnv::fit()        masks parsnip::fit(), rminer::fit()
## x recipes::fixed()         masks stringr::fixed()
## x dplyr::id()              masks plyr::id()
## x dplyr::lag()             masks stats::lag()
## x caret::lift()            masks purrr::lift()
## x yardstick::mae()         masks Metrics::mae()
## x yardstick::mape()        masks Metrics::mape()
## x randomForest::margin()   masks ggplot2::margin()
## x yardstick::mase()        masks Metrics::mase()
## x yardstick::metrics()     masks rminer::metrics()
## x dplyr::mutate()          masks plyr::mutate()
## x Matrix::pack()           masks tidyr::pack()
## x yardstick::precision()   masks caret::precision(),
Metrics::precision()
## x recipes::prepare()       masks VIM::prepare()
## x yardstick::recall()      masks caret::recall(), Metrics::recall()
## x dplyr::rename()          masks plyr::rename()
## x yardstick::rmse()        masks Metrics::rmse()
## x MASS::select()           masks dplyr::select()
## x yardstick::sensitivity() masks caret::sensitivity()
## x yardstick::smape()       masks Metrics::smape()
## x yardstick::spec()        masks readr::spec()
## x yardstick::specificity() masks caret::specificity()
## x dplyr::src()             masks Hmisc::src()
## x recipes::step()          masks stats::step()
## x dplyr::summarise()       masks plyr::summarise()
```

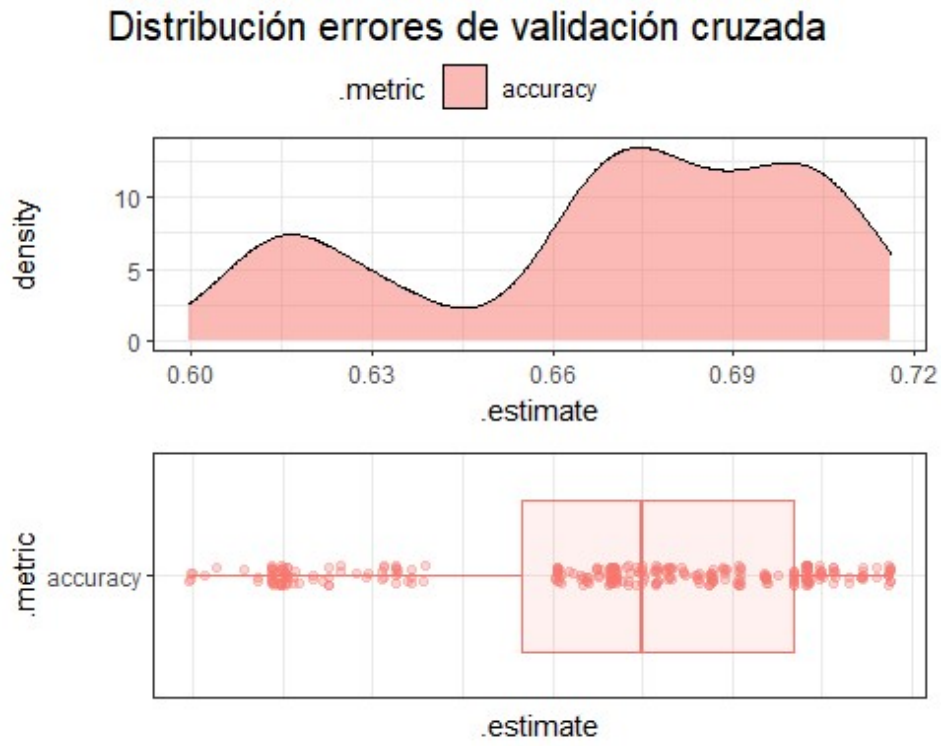
```
## x dplyr::summarize()       masks plyr::summarize(), Hmisc::summarize()
## x parsnip::translate()     masks Hmisc::translate()
## x Matrix::unpack()         masks tidyr::unpack()
## x recipes::update()        masks stats4::update(), Matrix::update(),
stats::update()
## x purrr::when()            masks foreach::when()
## * Use tidymodels_prefer() to resolve common conflicts.

library(parallel)
library(doParallel)

# DEFINICIÓN DEL MODELO Y DE LOS HIPERPARÁMETROS A OPTIMIZAR
#
================================================================================
=====
modelo <- rand_forest(
             mode  = "classification",
             mtry  = tune(),
             trees = tune()
          ) %>%
          set_engine(
            engine     = "ranger",
            max.depth  = tune(),
            importance = "none",
            seed       = 100
          )

control <- control_resamples(save_pred = TRUE)
# DEFINICIÓN DEL PREPROCESADO
#
================================================================================
=====
# En este caso no hay preprocesado, por lo que el transformer solo
contiene
# la definición de la fórmula y los datos de entrenamiento.
transformer <- recipe(
                  formula = Potability ~ .,
                  data    =  KNN_complete_sd_train
               )

# DEFINICIÓN DE LA ESTRATEGIA DE VALIDACIÓN Y CREACIÓN DE PARTICIONES
#
================================================================================
=====
set.seed(100)
cv_folds <- vfold_cv(
              data    = KNN_complete_sd_train,
              v       = 5,
              strata  = Potability
            )
```

```r
# WORKFLOW
#
================================================================================
=====
workflow_modelado <- workflow() %>%
                     add_recipe(transformer) %>%
                     add_model(modelo)


# GRID DE HIPERPARÁMETROS
#
================================================================================
=====
hiperpar_grid <- expand_grid(
                  'trees'     = c(100, 500, 1000, 2000),
                  'mtry'      = c( 4, 5),
                  'max.depth' = c(1, 3, 10, 20, 40, 60, 80, 100)
                 )

# EJECUCIÓN DE LA OPTIMIZACIÓN DE HIPERPARÁMETROS
#
================================================================================
=====
cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

grid_fit <- tune_grid(
              object    = workflow_modelado,
              resamples = cv_folds,
              metrics   = metric_set(yardstick::accuracy),
              grid      = hiperpar_grid,
              control = control
            )

stopCluster(cl)

grid_fit %>% collect_metrics(summarize = FALSE) %>% head()

## # A tibble: 6 x 8
##   id     mtry trees max.depth .metric  .estimator .estimate .config
##   <chr> <dbl> <dbl>     <dbl> <chr>    <chr>          <dbl> <chr>
## 1 Fold1     4   100         1 accuracy binary         0.616
Preprocessor1_Model~
## 2 Fold2     4   100         1 accuracy binary         0.613
Preprocessor1_Model~
## 3 Fold3     4   100         1 accuracy binary         0.613
Preprocessor1_Model~
## 4 Fold4     4   100         1 accuracy binary         0.602
Preprocessor1_Model~
```

```
## 5 Fold5     4   100         1 accuracy binary         0.615
Preprocessor1_Model~
## 6 Fold1     4   100         3 accuracy binary         0.616
Preprocessor1_Model~
```

Podemos ver la distribución de la exactitud en función de los parámetros seleccionados.

```
library(ggpubr)

##
## Attaching package: 'ggpubr'

## The following object is masked from 'package:plyr':
##
##     mutate

p1 <- ggplot(
        data = grid_fit %>% collect_metrics(summarize = FALSE),
        aes(x = .estimate, fill = .metric)) +
      geom_density(alpha = 0.5) +
      theme_bw()
p2 <- ggplot(
        data = grid_fit %>% collect_metrics(summarize = FALSE),
        aes(x = .metric, y = .estimate, fill = .metric, color = .metric))
+
      geom_boxplot(outlier.shape = NA, alpha = 0.1) +
      geom_jitter(width = 0.05, alpha = 0.3) +
      coord_flip() +
      theme_bw() +
      theme(axis.text.x = element_blank(), axis.ticks.x =
element_blank())

ggarrange(p1, p2, nrow = 2, common.legend = TRUE, align = "v") %>%
annotate_figure(
  top = text_grob("Distribución errores de validación cruzada", size =
15)
)
```



Podemos observar que los modelos generados se concentran alrededor de un 67% de exactitud.

```
show_best(grid_fit, metric="accuracy")
```

```
## # A tibble: 5 x 9
##    mtry trees max.depth .metric  .estimator  mean     n std_err
.config
##   <dbl> <dbl>     <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1     4   500        20 accuracy binary     0.690     5 0.00817
Preprocessor1_M~
## 2     5  2000        10 accuracy binary     0.690     5 0.00844
Preprocessor1_M~
## 3     5   500        10 accuracy binary     0.690     5 0.00931
Preprocessor1_M~
## 4     5   100        40 accuracy binary     0.689     5 0.00743
Preprocessor1_M~
## 5     5   100        60 accuracy binary     0.689     5 0.00743
Preprocessor1_M~
```

Con la función select_best() podemos generar un modelo con los mejores parámetros.

```
mejores_hiperpar <- select_best(grid_fit, metric="accuracy")

modelo_final_fit <- finalize_workflow(
                        x = workflow_modelado,
                        parameters = mejores_hiperpar
```

```
                    ) %>%
                    fit(
                      data = KNN_complete_sd_train
                    ) %>%
                    pull_workflow_fit()

predicciones <- modelo_final_fit %>%
                predict(new_data = KNN_complete_sd_test)

predicciones <- predicciones %>%
                bind_cols(KNN_complete_sd_test %>%
dplyr::select(Potability))

accuracy_test  <- accuracy(
                     data     = predicciones,
                     truth    = Potability,
                     estimate = .pred_class,
                     na_rm    = TRUE
                   )
accuracy_test
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.724
```

```
mat_confusion <- predicciones %>%
                 conf_mat(
                   truth     = Potability,
                   estimate  = .pred_class
                 )
mat_confusion
```

```
##           Truth
## Prediction   0   1
##          0 608 226
##          1  75 183
```

Vamos a realizar una prueba eliminando Organic_carbon:

```
library(randomForest)
set.seed(100)
model_6_1<-
randomForest(as.factor(Potability)~Sulfate+ph+Hardness+Solids+Chloramines
+Turbidity,KNN_complete_sd_train,ntree=2000, mtry=4)
model_6_1
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ Sulfate + ph +
Hardness + Solids + Chloramines + Turbidity, data =
```

```
KNN_complete_sd_train,      ntree = 2000, mtry = 4)
##                Type of random forest: classification
##                      Number of trees: 2000
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 30.36%
## Confusion matrix:
##      0   1 class.error
## 0 1097 218   0.1657795
## 1  445 424   0.5120829

predict_rf <- predict(model_6_1, KNN_complete_sd_test)
confusionMatrix(predict_rf, as.factor(KNN_complete_sd_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 588 202
##          1  95 207
##
##                Accuracy : 0.728
##                  95% CI : (0.7006, 0.7542)
##     No Information Rate : 0.6255
##     P-Value [Acc > NIR] : 4.984e-13
##
##                   Kappa : 0.3873
##
##  Mcnemar's Test P-Value : 7.712e-10
##
##             Sensitivity : 0.8609
##             Specificity : 0.5061
##          Pos Pred Value : 0.7443
##          Neg Pred Value : 0.6854
##              Prevalence : 0.6255
##          Detection Rate : 0.5385
##    Detection Prevalence : 0.7234
##       Balanced Accuracy : 0.6835
##
##        'Positive' Class : 0
##
```

Vamos a probar ahora eliminando Turbudity:

```
library(randomForest)
set.seed(100)
model_6_2<-
randomForest(as.factor(Potability)~Sulfate+ph+Hardness+Solids+Chloramines
+Organic_carbon,KNN_complete_sd_train,ntree=2000, mtry=4)
model_6_2
```

```
##
## Call:
##  randomForest(formula = as.factor(Potability) ~ Sulfate + ph +
Hardness + Solids + Chloramines + Organic_carbon, data =
KNN_complete_sd_train,      ntree = 2000, mtry = 4)
##                Type of random forest: classification
##                      Number of trees: 2000
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 29.58%
## Confusion matrix:
##      0   1 class.error
## 0 1107 208   0.1581749
## 1  438 431   0.5040276
```

```
predict_rf <- predict(model_6_2, KNN_complete_sd_test)
cm <- confusionMatrix(predict_rf,
as.factor(KNN_complete_sd_test$Potability))
cm
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 587 200
##          1  96 209
##
##                Accuracy : 0.7289
##                  95% CI : (0.7015, 0.7551)
##     No Information Rate : 0.6255
##     P-Value [Acc > NIR] : 3.083e-13
##
##                   Kappa : 0.3904
##
##  Mcnemar's Test P-Value : 2.141e-09
##
##             Sensitivity : 0.8594
##             Specificity : 0.5110
##          Pos Pred Value : 0.7459
##          Neg Pred Value : 0.6852
##              Prevalence : 0.6255
##          Detection Rate : 0.5375
##    Detection Prevalence : 0.7207
##       Balanced Accuracy : 0.6852
##
##        'Positive' Class : 0
##
```

```
overall <- cm$overall
overall.accuracy <- overall['Accuracy']
temp <- data.frame("Nombre"="model_6_mod",
```

```
"accuracy"=c(overall.accuracy))
accuracy_data <- rbind(accuracy_data, temp)
```

Vemos que eliminando Turbidity mejoramos la exactitud, por encima del 72% y coincidiendo con OOB error.

Probamos eliminando ambas:

```
library(randomForest)
set.seed(100)
model_6_3<-
randomForest(as.factor(Potability)~Sulfate+ph+Hardness+Solids+Chloramines
,KNN_complete_sd_train,ntree=2000, mtry=4)
model_6_3

##
## Call:
##  randomForest(formula = as.factor(Potability) ~ Sulfate + ph +
Hardness + Solids + Chloramines, data = KNN_complete_sd_train,      ntree
= 2000, mtry = 4)
##                Type of random forest: classification
##                      Number of trees: 2000
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 29.85%
## Confusion matrix:
##      0   1 class.error
## 0 1096 219   0.1665399
## 1  433 436   0.4982739

predict_rf <- predict(model_6_3, KNN_complete_sd_test)
confusionMatrix(predict_rf, as.factor(KNN_complete_sd_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 582 200
##          1 101 209
##
##                Accuracy : 0.7244
##                  95% CI : (0.6968, 0.7507)
##     No Information Rate : 0.6255
##     P-Value [Acc > NIR] : 3.255e-12
##
##                   Kappa : 0.3817
##
##  Mcnemar's Test P-Value : 1.617e-08
##
##             Sensitivity : 0.8521
##             Specificity : 0.5110
```

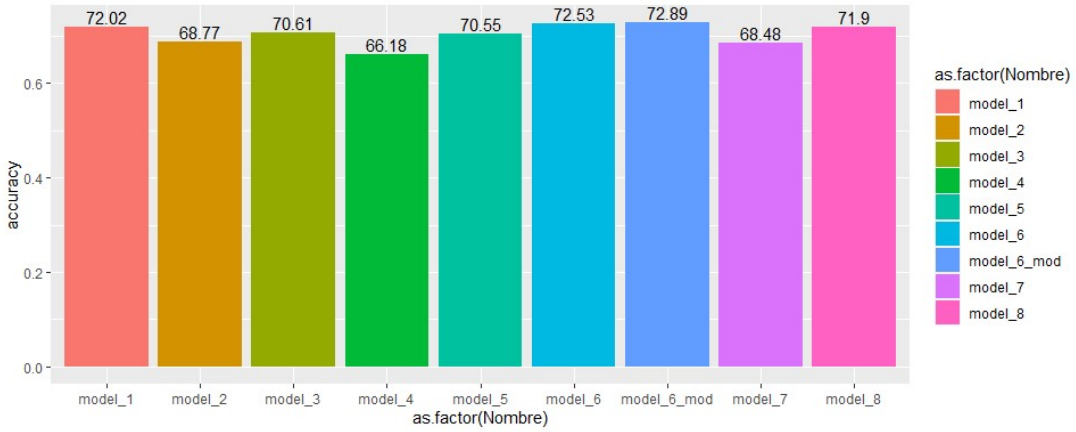
```
##          Pos Pred Value : 0.7442
##          Neg Pred Value : 0.6742
##              Prevalence : 0.6255
##          Detection Rate : 0.5330
##    Detection Prevalence : 0.7161
##       Balanced Accuracy : 0.6816
##
##        'Positive' Class : 0
##
```

En cambio, eliminando las dos, el resultado es peor.

Por tanto, podemos concluir que el mejor conjunto para la predicción es KNN_complete_sd, eliminando Turbidity.

Podemos mostrar los resultados de todos los modelos y el que hemos considerado mejor y ajustado mejor los parámetros:

```
ggplot(accuracy_data, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity")  + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```



XGBOOST

Pasamos a realizar los mismos pasos que en el caso de RabdomForest con el algoritmo xgboost classifier. Para cada modelo, evaluaremos los valores de los parámetros que luego utilizaremos en cada uno.

Tras diferentes pruebas y para reducir el tiempo de ejecución, en el caso de los parámetros colsample_bytree, subsample y min_child_weight hemos decidido dejarlos constantes para todos los modelos con valor 1.

Para evitar errores, volvemos a lanzar la división de los conjuntos de entrenamiento y test.

```r
library(rminer)
set.seed(100)

h<-holdout(mF_complete_Tuk$Potability,ratio=2/3,mode="stratified")
mF_complete_Tuk_train<-mF_complete_Tuk[h$tr,]
mF_complete_Tuk_test<-mF_complete_Tuk[h$ts,]
print(table(mF_complete_Tuk_train$Potability))

##
##    0    1
## 1125  703

print(table(mF_complete_Tuk_test$Potability))

##
##   0   1
## 597 318

h<-holdout(mF_complete_sd$Potability,ratio=2/3,mode="stratified")
mF_complete_sd_train<-mF_complete_sd[h$tr,]
mF_complete_sd_test<-mF_complete_sd[h$ts,]
print(table(mF_complete_sd_train$Potability))

##
##    0    1
## 1353  831

print(table(mF_complete_sd_test$Potability))

##
##   0   1
## 645 447

h<-holdout(mF_Tukey_Tuk$Potability,ratio=2/3,mode="stratified")
mF_Tukey_Tuk_train<-mF_Tukey_Tuk[h$tr,]
mF_Tukey_Tuk_test<-mF_Tukey_Tuk[h$ts,]
print(table(mF_Tukey_Tuk_train$Potability))

##
##    0    1
## 1105  669

print(table(mF_Tukey_Tuk_test$Potability))

##
##   0   1
## 563 325

h<-holdout(mF_sd_sd$Potability,ratio=2/3,mode="stratified")
mF_sd_sd_train<-mF_sd_sd[h$tr,]
mF_sd_sd_test<-mF_sd_sd[h$ts,]
print(table(mF_sd_sd_train$Potability))
```

```
##
##    0    1
## 1299  763

print(table(mF_sd_sd_test$Potability))

##
##   0   1
## 616 416

h<-holdout(KNN_complete_Tuk$Potability,ratio=2/3,mode="stratified")
KNN_complete_Tuk_train<-KNN_complete_Tuk[h$tr,]
KNN_complete_Tuk_test<-KNN_complete_Tuk[h$ts,]
print(table(KNN_complete_Tuk_train$Potability))

##
##    0    1
## 1118  666

print(table(KNN_complete_Tuk_test$Potability))

##
##   0   1
## 558 335

h<-holdout(KNN_complete_sd$Potability,ratio=2/3,mode="stratified")
KNN_complete_sd_train<-KNN_complete_sd[h$tr,]
KNN_complete_sd_test<-KNN_complete_sd[h$ts,]
print(table(KNN_complete_sd_train$Potability))

##
##    0    1
## 1315  869

print(table(KNN_complete_sd_test$Potability))

##
##   0   1
## 683 409

h<-holdout(KNN_Tukey_Tuk$Potability,ratio=2/3,mode="stratified")
KNN_Tukey_Tuk_train<-KNN_Tukey_Tuk[h$tr,]
KNN_Tukey_Tuk_test<-KNN_Tukey_Tuk[h$ts,]
print(table(KNN_Tukey_Tuk_train$Potability))

##
##    0    1
## 1100  630

print(table(KNN_Tukey_Tuk_test$Potability))
```

```
##
##   0   1
## 523 343
```

```
h<-holdout(KNN_sd_sd$Potability,ratio=2/3,mode="stratified")
KNN_sd_sd_train<-KNN_sd_sd[h$tr,]
KNN_sd_sd_test<-KNN_sd_sd[h$ts,]
print(table(KNN_sd_sd_train$Potability))
```

```
##
##    0    1
## 1254  808
```

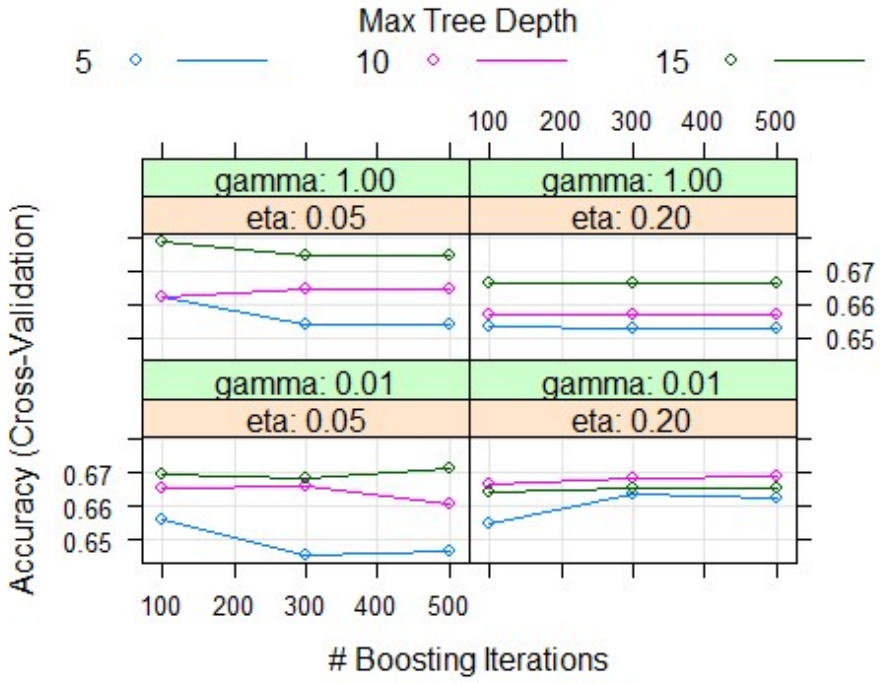
```
print(table(KNN_sd_sd_test$Potability))
```

```
##
##   0   1
## 661 371
```

Modelo 1

```
library(caret)

# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_1 <- train(as.factor(Potability) ~., data =
mF_complete_Tuk_train, method = "xgbTree",
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados gráficos de los modelos, que nos permitirán elegir los
```

```r
mejores valores de los parámetros.
plot(model_xgb_1)
```




```r
# Test y matriz de confusión
test_predict <- predict(model_xgb_1, mF_complete_Tuk_test)
confusionMatrix(test_predict, as.factor(mF_complete_Tuk_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 496 175
##          1 101 143
##
##                Accuracy : 0.6984
##                  95% CI : (0.6675, 0.728)
##     No Information Rate : 0.6525
##     P-Value [Acc > NIR] : 0.001809
##
##                   Kappa : 0.2966
##
##  Mcnemar's Test P-Value : 1.112e-05
##
##             Sensitivity : 0.8308
##             Specificity : 0.4497
##          Pos Pred Value : 0.7392
##          Neg Pred Value : 0.5861
```

```
##              Prevalence : 0.6525
##          Detection Rate : 0.5421
##    Detection Prevalence : 0.7333
##       Balanced Accuracy : 0.6403
##
##        'Positive' Class : 0
##
```

Elegimos los siguientes valores para este modelo:

tune_grid <- expand.grid(nrounds=c(100), max_depth = c(15), eta = c(0.05), gamma = c(1)

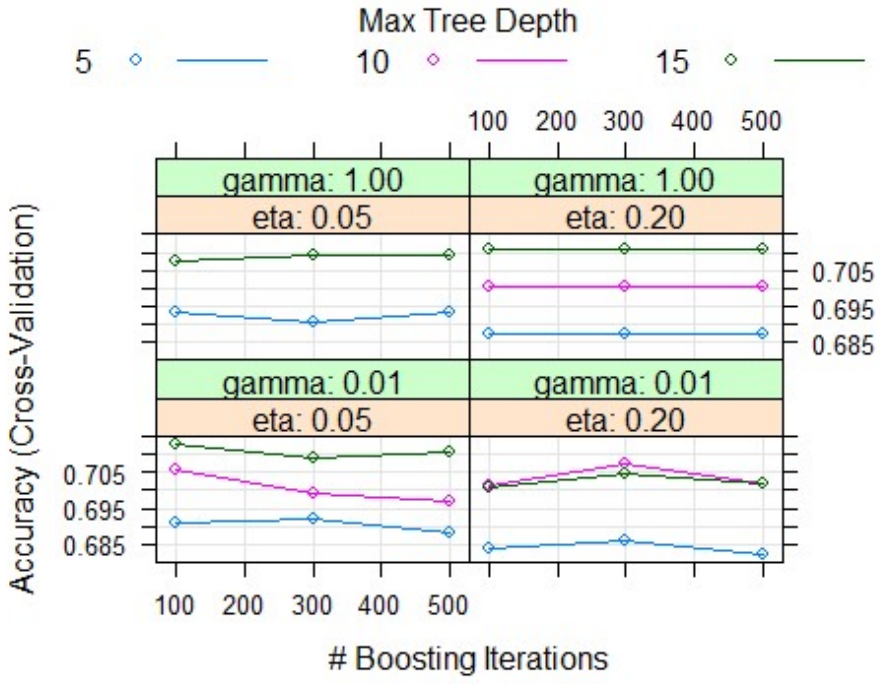
```
library(xgboost)

##
## Attaching package: 'xgboost'

## The following object is masked from 'package:dplyr':
##
##     slice

set.seed(100)

train.data = as.matrix(mF_complete_Tuk_train[, -10])
train.label = mF_complete_Tuk_train[, 10]
test.data = as.matrix(mF_complete_Tuk_test[, -10])
test.label = mF_complete_Tuk_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(mF_complete_Tuk_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 15,
  gamma = 1,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 100
```

```
)

# Resultados
xgb.fit

## ##### xgb.Booster
## raw: 4.3 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 100)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "15", gamma = "1",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 100
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(mF_complete_Tuk_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(mF_complete_Tuk_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb1 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb1)))

## [1] "Final Accuracy = 69.73%"

accuracy_data2<- data.frame("Nombre"="model_xgb_1",
"accuracy"=result_xgb1)
```

Modelo 2

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)
```

```r
tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_2 <- train(as.factor(Potability) ~., data =
mF_complete_sd_train, method = "xgbTree",
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_2)
```




```r
# Test y matriz de confusión
test_predict <- predict(model_xgb_2, mF_complete_sd_test)
confusionMatrix(test_predict, as.factor(mF_complete_sd_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction   0   1
##          0 537 246
##          1 108 201
##
##                Accuracy : 0.6758
##                  95% CI : (0.6472, 0.7035)
##     No Information Rate : 0.5907
##     P-Value [Acc > NIR] : 4.022e-09
##
##                   Kappa : 0.2963
##
##  Mcnemar's Test P-Value : 3.302e-13
##
##             Sensitivity : 0.8326
##             Specificity : 0.4497
##          Pos Pred Value : 0.6858
##          Neg Pred Value : 0.6505
##              Prevalence : 0.5907
##          Detection Rate : 0.4918
##    Detection Prevalence : 0.7170
##       Balanced Accuracy : 0.6411
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(100), max_depth = c(15), eta = c(0.05), gamma = c(0.01),

```
set.seed(100)

train.data = as.matrix(mF_complete_sd_train[, -10])
train.label = mF_complete_sd_train[, 10]
test.data = as.matrix(mF_complete_sd_test[, -10])
test.label = mF_complete_sd_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(mF_complete_sd_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 15,
  gamma = 0.01,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
```

```
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 100
)

# Resultados
xgb.fit
```

```
## ##### xgb.Booster
## raw: 4.2 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 100)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "15", gamma = "0.01",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 100
## nfeatures : 9
```

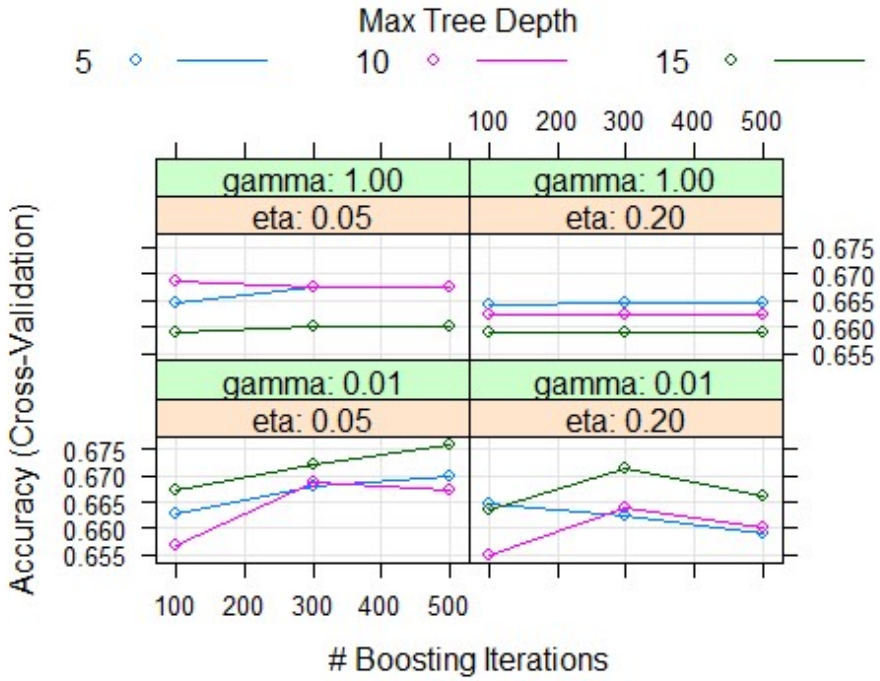
```
# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(mF_complete_sd_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(mF_complete_sd_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb2 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb2)))
```

```
## [1] "Final Accuracy = 67.77%"
```

```
temp <- data.frame("Nombre"="model_xgb_2", "accuracy"=c(result_xgb2))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 3

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_3 <- train(as.factor(Potability) ~., data = mF_Tukey_Tuk_train,
method = "xgbTree",
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_3)
```




```
# Test y matriz de confusión
test_predict <- predict(model_xgb_3, mF_Tukey_Tuk_test)
confusionMatrix(test_predict, as.factor(mF_Tukey_Tuk_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 460 175
##          1 103 150
##
##                Accuracy : 0.6869
##                  95% CI : (0.6553, 0.7173)
##     No Information Rate : 0.634
##     P-Value [Acc > NIR] : 0.0005323
##
##                   Kappa : 0.2923
##
##  Mcnemar's Test P-Value : 2.06e-05
##
##             Sensitivity : 0.8171
##             Specificity : 0.4615
##          Pos Pred Value : 0.7244
##          Neg Pred Value : 0.5929
##              Prevalence : 0.6340
##          Detection Rate : 0.5180
##    Detection Prevalence : 0.7151
```

```
##       Balanced Accuracy : 0.6393
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(500), max_depth = c(15), eta = c(0.05), gamma = c(0.01),

```
set.seed(100)

train.data = as.matrix(mF_Tukey_Tuk_train[, -10])
train.label = mF_Tukey_Tuk_train[, 10]
test.data = as.matrix(mF_Tukey_Tuk_test[, -10])
test.label = mF_Tukey_Tuk_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(mF_Tukey_Tuk_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 15,
  gamma = 0.01,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 500
)

# Resultados
xgb.fit
```

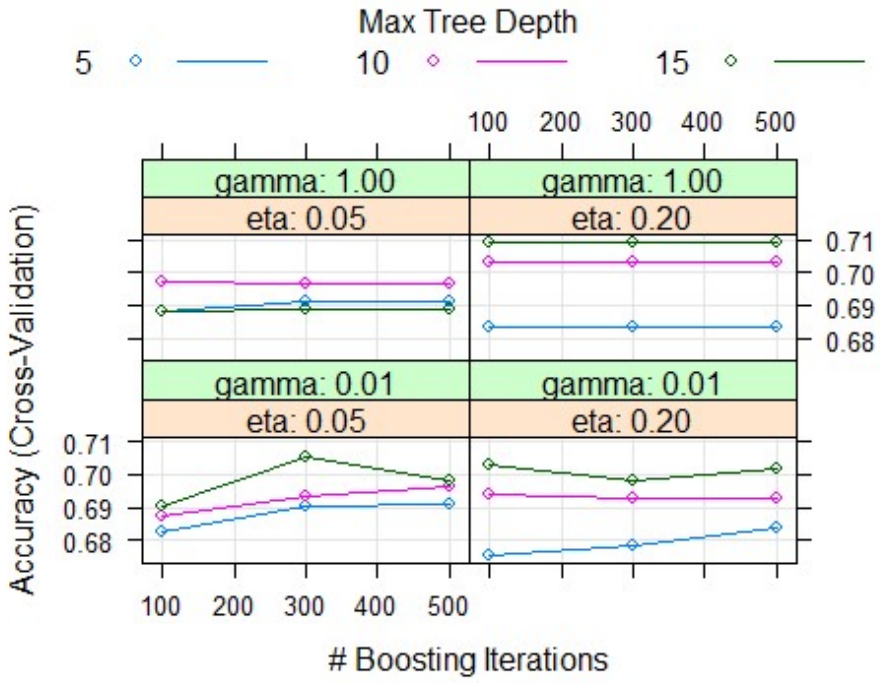
```
## ##### xgb.Booster
## raw: 9.4 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 500)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "15", gamma = "0.01",
```

```
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 500
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(mF_Tukey_Tuk_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(mF_Tukey_Tuk_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb3 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb3)))

## [1] "Final Accuracy = 68.81%"

temp <- data.frame("Nombre"="model_xgb_3", "accuracy"=c(result_xgb3))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 4

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_4 <- train(as.factor(Potability) ~., data = mF_sd_sd_train,
method = "xgbTree",
```

```
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_4)
```




```
# Test y matriz de confusión
test_predict <- predict(model_xgb_4, mF_sd_sd_test)
confusionMatrix(test_predict, as.factor(mF_sd_sd_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 496 216
##          1 120 200
##
##                Accuracy : 0.6744
##                  95% CI : (0.6449, 0.703)
##     No Information Rate : 0.5969
##     P-Value [Acc > NIR] : 1.634e-07
##
##                   Kappa : 0.2971
##
```

```
##  Mcnemar's Test P-Value : 2.187e-07
##
##             Sensitivity : 0.8052
##             Specificity : 0.4808
##          Pos Pred Value : 0.6966
##          Neg Pred Value : 0.6250
##              Prevalence : 0.5969
##          Detection Rate : 0.4806
##    Detection Prevalence : 0.6899
##       Balanced Accuracy : 0.6430
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(500), max_depth = c(15), eta = c(0.2), gamma = c(1),

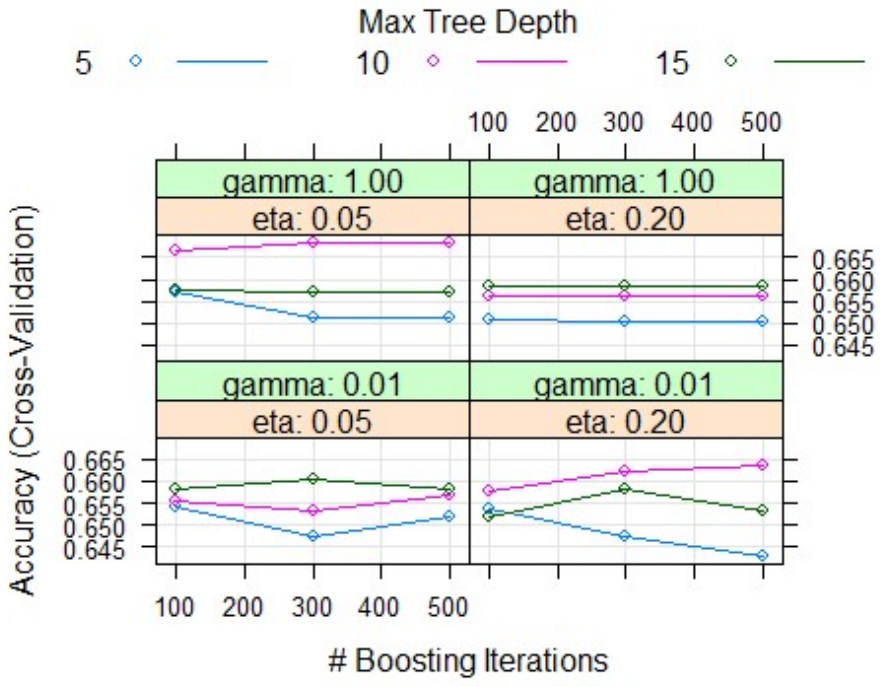
```
set.seed(100)

train.data = as.matrix(mF_sd_sd_train[, -10])
train.label = mF_sd_sd_train[, 10]
test.data = as.matrix(mF_sd_sd_test[, -10])
test.label = mF_sd_sd_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(mF_sd_sd_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.2,
  max_depth = 15,
  gamma = 1,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 500
)
```

```r
# Resultados
xgb.fit

## ##### xgb.Booster
## raw: 19.2 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 500)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.2", max_depth = "15", gamma = "1",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 500
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(mF_sd_sd_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(mF_sd_sd_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb4 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb4)))

## [1] "Final Accuracy = 66.18%"

temp <- data.frame("Nombre"="model_xgb_4", "accuracy"=c(result_xgb4))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 5

```r
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
```

```
                            eta = c(0.05, 0.2),
                            gamma = c(0.01, 1),
                            colsample_bytree = c(1),
                            subsample = c(1),
                            min_child_weight = c(1))

model_xgb_5 <- train(as.factor(Potability) ~., data =
KNN_complete_Tuk_train, method = "xgbTree",
                     trControl=trctrl,
                     tuneGrid = tune_grid,
                     tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_5)
```




```
# Test y matriz de confusión
test_predict <- predict(model_xgb_5, KNN_complete_Tuk_test)
confusionMatrix(test_predict,
as.factor(KNN_complete_Tuk_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 479 196
```

```
##          1  79 139
##
##                Accuracy : 0.692
##                  95% CI : (0.6606, 0.7222)
##     No Information Rate : 0.6249
##     P-Value [Acc > NIR] : 1.548e-05
##
##                   Kappa : 0.2939
##
##  Mcnemar's Test P-Value : 2.651e-12
##
##             Sensitivity : 0.8584
##             Specificity : 0.4149
##          Pos Pred Value : 0.7096
##          Neg Pred Value : 0.6376
##              Prevalence : 0.6249
##          Detection Rate : 0.5364
##    Detection Prevalence : 0.7559
##       Balanced Accuracy : 0.6367
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(500), max_depth = c(10), eta = c(0.05), gamma = c(1)

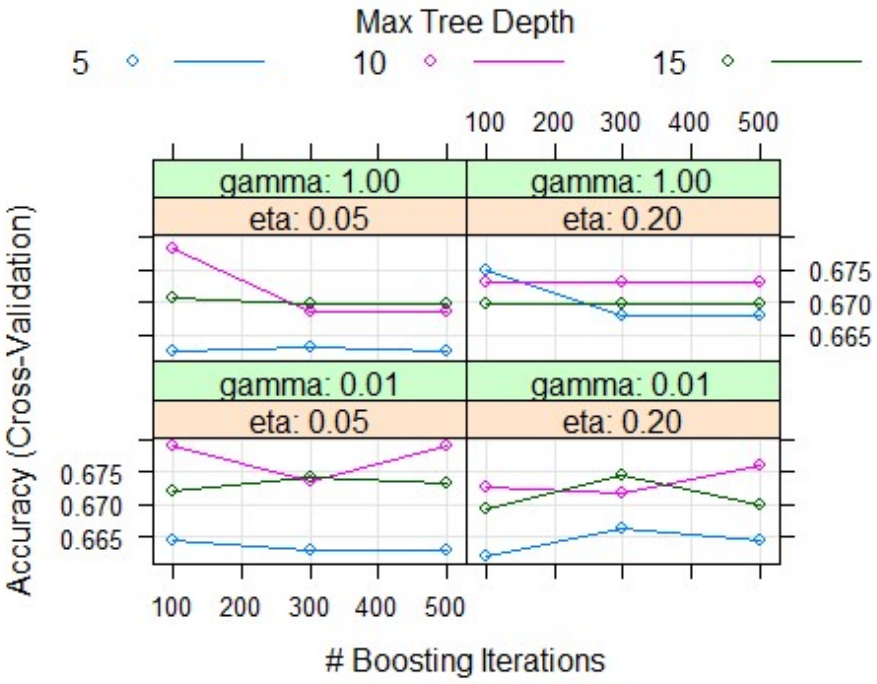
```
set.seed(100)

train.data = as.matrix(KNN_complete_Tuk_train[, -10])
train.label = KNN_complete_Tuk_train[, 10]
test.data = as.matrix(KNN_complete_Tuk_test[, -10])
test.label = KNN_complete_Tuk_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(KNN_complete_Tuk_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 10,
  gamma = 1,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
```

```
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 500
)

# Resultados
xgb.fit

## ##### xgb.Booster
## raw: 6.6 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 500)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "10", gamma = "1",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 500
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(KNN_complete_Tuk_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(KNN_complete_Tuk_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb5 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb5)))

## [1] "Final Accuracy = 67.75%"

temp <- data.frame("Nombre"="model_xgb_5", "accuracy"=c(result_xgb5))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 6

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)
```

```r
# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_6 <- train(as.factor(Potability) ~., data =
KNN_complete_sd_train, method = "xgbTree",
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_6)
```



```
# Test y matriz de confusión
test_predict <- predict(model_xgb_6, KNN_complete_sd_test)
confusionMatrix(test_predict, as.factor(KNN_complete_sd_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 558 209
##          1 125 200
##
##                Accuracy : 0.6941
##                  95% CI : (0.6659, 0.7214)
##     No Information Rate : 0.6255
##     P-Value [Acc > NIR] : 1.168e-06
##
##                   Kappa : 0.3191
##
##  Mcnemar's Test P-Value : 5.584e-06
##
##             Sensitivity : 0.8170
##             Specificity : 0.4890
##          Pos Pred Value : 0.7275
##          Neg Pred Value : 0.6154
##              Prevalence : 0.6255
##          Detection Rate : 0.5110
##    Detection Prevalence : 0.7024
##       Balanced Accuracy : 0.6530
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(500), max_depth = c(10), eta = c(0.05), gamma = c(0.01)

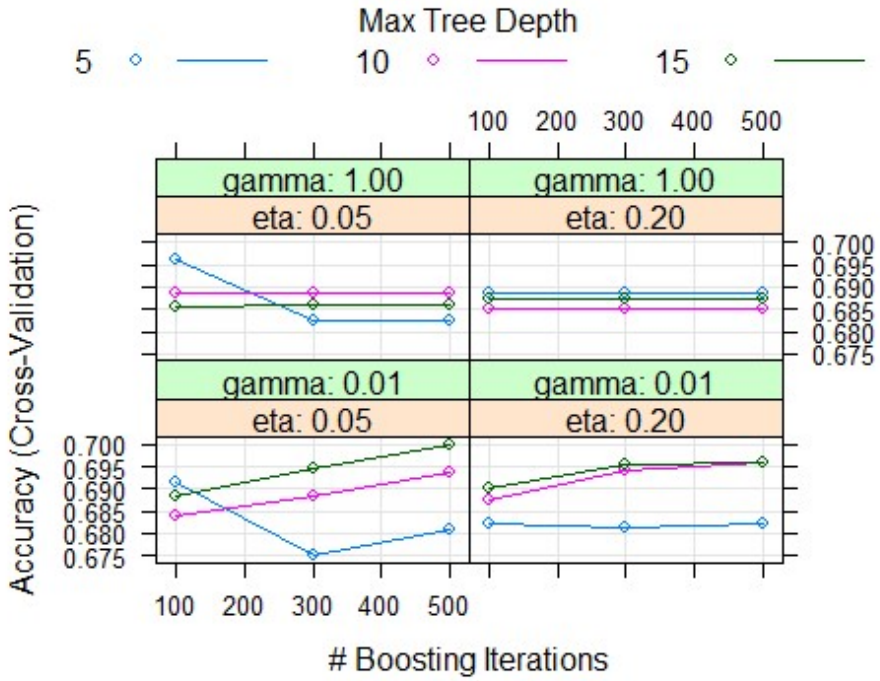
```
set.seed(100)

train.data = as.matrix(KNN_complete_sd_train[, -10])
train.label = KNN_complete_sd_train[, 10]
test.data = as.matrix(KNN_complete_sd_test[, -10])
test.label = KNN_complete_sd_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(KNN_complete_sd_train$Potability))
params = list(
```

```
  booster="gbtree",
  eta = 0.05,
  max_depth = 10,
  gamma = 0.01,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 500
)

# Resultados
xgb.fit

## ##### xgb.Booster
## raw: 7.7 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 500)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "10", gamma = "0.01",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 500
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(KNN_complete_sd_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(KNN_complete_sd_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb6 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb6)))
```

```
## [1] "Final Accuracy = 70.60%"

temp <- data.frame("Nombre"="model_xgb_6", "accuracy"=c(result_xgb6))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 7

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_7 <- train(as.factor(Potability) ~., data =
KNN_Tukey_Tuk_train, method = "xgbTree",
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_7)
```




```
# Test y matriz de confusión
test_predict <- predict(model_xgb_7, KNN_Tukey_Tuk_test)
confusionMatrix(test_predict, as.factor(KNN_Tukey_Tuk_test$Potability))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 449 203
##          1  74 140
##
##                Accuracy : 0.6801
##                  95% CI : (0.6479, 0.7111)
##     No Information Rate : 0.6039
##     P-Value [Acc > NIR] : 2.022e-06
##
##                   Kappa : 0.2851
##
##  Mcnemar's Test P-Value : 1.462e-14
##
##             Sensitivity : 0.8585
##             Specificity : 0.4082
##          Pos Pred Value : 0.6887
##          Neg Pred Value : 0.6542
##              Prevalence : 0.6039
##          Detection Rate : 0.5185
##    Detection Prevalence : 0.7529
```

```
##       Balanced Accuracy : 0.6333
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(500), max_depth = c(15), eta = c(0.05), gamma = c(0.01)

```
set.seed(100)

train.data = as.matrix(KNN_Tukey_Tuk_train[, -10])
train.label = KNN_Tukey_Tuk_train[, 10]
test.data = as.matrix(KNN_Tukey_Tuk_test[, -10])
test.label = KNN_Tukey_Tuk_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(KNN_Tukey_Tuk_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 15,
  gamma = 0.01,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 500
)

# Resultados
xgb.fit
```

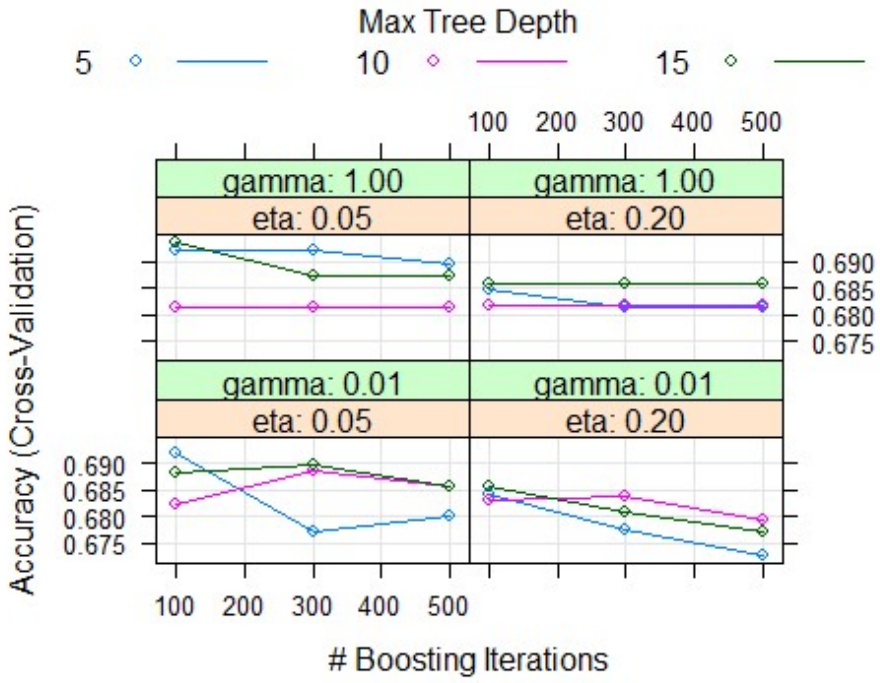
```
## ##### xgb.Booster
## raw: 9.1 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 500)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "15", gamma = "0.01",
```

```
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 500
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(KNN_Tukey_Tuk_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(KNN_Tukey_Tuk_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb7 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb7)))

## [1] "Final Accuracy = 67.09%"

temp <- data.frame("Nombre"="model_xgb_7", "accuracy"=c(result_xgb7))
accuracy_data2 <- rbind(accuracy_data2, temp)
```

Modelo 8

```
# Seleccionamos el método, Cross Validation y el número de folds, 5.
trctrl <- trainControl(method = "cv", number = 5)

# Lanzamos múltiples modelos con distintos valores para cada uno de los
parámetros.
set.seed(100)

cl <- makePSOCKcluster(parallel::detectCores() - 1)
registerDoParallel(cl)

tune_grid <- expand.grid(nrounds=c(100,300,500),
                         max_depth = c(5, 10, 15),
                         eta = c(0.05, 0.2),
                         gamma = c(0.01, 1),
                         colsample_bytree = c(1),
                         subsample = c(1),
                         min_child_weight = c(1))

model_xgb_8 <- train(as.factor(Potability) ~., data = KNN_sd_sd_train,
method = "xgbTree",
```

```
                trControl=trctrl,
                tuneGrid = tune_grid,
                tuneLength = 10)

stopCluster(cl)

# Resultados de los modelos.
plot(model_xgb_8)
```




```
# Test y matriz de confusión
test_predict <- predict(model_xgb_8, KNN_sd_sd_test)
confusionMatrix(test_predict, as.factor(KNN_sd_sd_test$Potability))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 540 193
##          1 121 178
##
##                Accuracy : 0.6957
##                  95% CI : (0.6667, 0.7237)
##     No Information Rate : 0.6405
##     P-Value [Acc > NIR] : 0.0001044
##
##                   Kappa : 0.3099
##
```

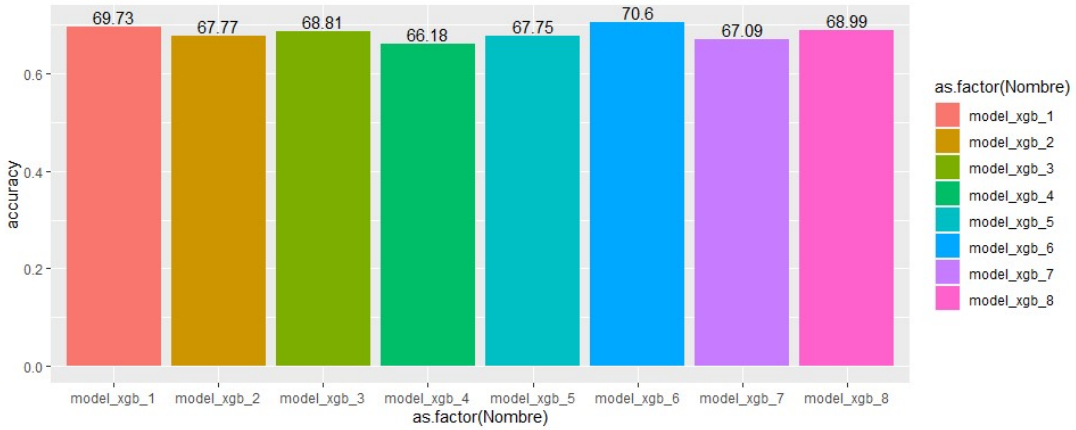
```
##  Mcnemar's Test P-Value : 6.156e-05
##
##             Sensitivity : 0.8169
##             Specificity : 0.4798
##          Pos Pred Value : 0.7367
##          Neg Pred Value : 0.5953
##              Prevalence : 0.6405
##          Detection Rate : 0.5233
##    Detection Prevalence : 0.7103
##       Balanced Accuracy : 0.6484
##
##        'Positive' Class : 0
##
```

En este caso, nos quedamos con:

tune_grid <- expand.grid(nrounds=c(100), max_depth = c(15), eta = c(0.05), gamma = c(1)

```
set.seed(100)

train.data = as.matrix(KNN_sd_sd_train[, -10])
train.label = KNN_sd_sd_train[, 10]
test.data = as.matrix(KNN_sd_sd_test[, -10])
test.label = KNN_sd_sd_test[, 10]

xgb.train = xgb.DMatrix(data=train.data,label=(train.label))
xgb.test = xgb.DMatrix(data=test.data,label=(test.label))

# Definición de los parámetros seleccionados
num_class = length(unique(KNN_sd_sd_train$Potability))
params = list(
  booster="gbtree",
  eta = 0.05,
  max_depth = 15,
  gamma = 1,
  subsample = 1,
  colsample_bytree = 1,
  objective="multi:softprob",
  eval_metric="mlogloss",
  num_class=num_class
)

# Entrenamiento del modelo
xgb.fit=xgb.train(
  params=params,
  data=xgb.train,
  nrounds = 100
)
```

```r
# Resultados
xgb.fit

## ##### xgb.Booster
## raw: 4.3 Mb
## call:
##   xgb.train(params = params, data = xgb.train, nrounds = 100)
## params (as set within xgb.train):
##   booster = "gbtree", eta = "0.05", max_depth = "15", gamma = "1",
subsample = "1", colsample_bytree = "1", objective = "multi:softprob",
eval_metric = "mlogloss", num_class = "2", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
## # of features: 9
## niter: 100
## nfeatures : 9

# Predicción
xgb.pred = predict(xgb.fit,test.data,reshape=T)
xgb.pred = as.data.frame(xgb.pred)
colnames(xgb.pred) = unique(KNN_sd_sd_train$Potability)

# Use the predicted label with the highest probability
xgb.pred$prediction = apply(xgb.pred,1,function(x)
colnames(xgb.pred)[which.max(x)])
xgb.pred$label = unique(KNN_sd_sd_train$Potability)[test.label+1]

# Calculate the final accuracy
result_xgb8 = sum(xgb.pred$prediction==xgb.pred$label)/nrow(xgb.pred)
print(paste("Final Accuracy =",sprintf("%1.2f%%", 100*result_xgb8)))

## [1] "Final Accuracy = 68.99%"

temp <- data.frame("Nombre"="model_xgb_8", "accuracy"=c(result_xgb8))
accuracy_data2 <- rbind(accuracy_data2, temp)

ggplot(accuracy_data2, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity") + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```



Como en el caso de randomForest, el mejor resultado lo hemos obtenido con el modelo 6, con una exactitud de 70.15%.

Regresión logística

Modelo 1

```
library(dplyr)

train.data  <- mF_complete_Tuk_train
test.data <- mF_complete_Tuk_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)

##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0789  -0.9900  -0.9606   1.3702   1.4762
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -7.898e-01  9.294e-01  -0.850    0.395
## ph              -1.709e-02  3.929e-02  -0.435    0.664
## Hardness         1.051e-03  1.750e-03   0.601    0.548
## Solids           6.985e-06  6.233e-06   1.121    0.262
## Chloramines      3.192e-03  3.447e-02   0.093    0.926
## Sulfate          6.640e-04  1.693e-03   0.392    0.695
## Conductivity    -1.351e-04  5.994e-04  -0.225    0.822
## Organic_carbon  -7.784e-03  1.493e-02  -0.521    0.602
## Trihalomethanes -1.060e-05  3.159e-03  -0.003    0.997
```

```
## Turbidity        2.337e-03  6.218e-02   0.038    0.970
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2435.8  on 1827  degrees of freedom
## Residual deviance: 2433.7  on 1818  degrees of freedom
## AIC: 2453.7
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.652459
```

```
accuracy_data_3<- data.frame("Nombre"="model_1_reg", "accuracy"=result)
```

Modelo 2

```
library(dplyr)

train.data  <- mF_complete_sd_train
test.data <- mF_complete_sd_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.1014  -0.9849  -0.9511   1.3782   1.5190
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -4.151e-01  7.624e-01  -0.545    0.586
## ph              -1.153e-02  2.989e-02  -0.386    0.700
## Hardness         4.761e-04  1.378e-03   0.346    0.730
## Solids           2.288e-06  5.142e-06   0.445    0.656
## Chloramines      3.074e-02  2.794e-02   1.100    0.271
## Sulfate         -1.065e-03  1.253e-03  -0.850    0.395
## Conductivity    -6.359e-05  5.441e-04  -0.117    0.907
```

```
## Organic_carbon  -5.513e-03  1.320e-02  -0.418    0.676
## Trihalomethanes  1.942e-03  2.816e-03   0.689    0.491
## Turbidity       -5.732e-03  5.684e-02  -0.101    0.920
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2901.7  on 2183  degrees of freedom
## Residual deviance: 2898.3  on 2174  degrees of freedom
## AIC: 2918.3
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.5906593
```

```
temp <- data.frame("Nombre"="model_2_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 3

```
library(dplyr)

train.data  <- mF_Tukey_Tuk_train
test.data <- mF_Tukey_Tuk_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0733  -0.9793  -0.9418   1.3813   1.5040
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -2.666e-01  9.741e-01  -0.274    0.784
## ph              -3.125e-02  4.057e-02  -0.770    0.441
## Hardness        -7.136e-04  1.804e-03  -0.395    0.692
## Solids           2.471e-06  6.326e-06   0.391    0.696
```

```
## Chloramines      3.622e-02  3.515e-02   1.030    0.303
## Sulfate         -6.940e-05  1.777e-03  -0.039    0.969
## Conductivity    -4.895e-04  6.219e-04  -0.787    0.431
## Organic_carbon  -1.922e-03  1.529e-02  -0.126    0.900
## Trihalomethanes  2.465e-04  3.263e-03   0.076    0.940
## Turbidity        1.453e-02  6.516e-02   0.223    0.824
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2351  on 1773  degrees of freedom
## Residual deviance: 2348  on 1764  degrees of freedom
## AIC: 2368
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.634009
```

```
temp <- data.frame("Nombre"="model_3_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 4

```
library(dplyr)

train.data  <- mF_sd_sd_train
test.data <- mF_sd_sd_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0830  -0.9673  -0.9320   1.3929   1.5295
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -9.534e-01  7.998e-01  -1.192    0.233
```

```
## ph              -1.844e-02  3.386e-02  -0.544    0.586
## Hardness         5.815e-04  1.494e-03   0.389    0.697
## Solids           6.317e-06  5.690e-06   1.110    0.267
## Chloramines      2.646e-02  3.076e-02   0.860    0.390
## Sulfate          1.039e-03  1.383e-03   0.751    0.453
## Conductivity    -5.304e-04  5.733e-04  -0.925    0.355
## Organic_carbon  -5.687e-03  1.393e-02  -0.408    0.683
## Trihalomethanes  9.470e-04  2.974e-03   0.318    0.750
## Turbidity        2.308e-03  6.020e-02   0.038    0.969
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2717.6  on 2061  degrees of freedom
## Residual deviance: 2713.8  on 2052  degrees of freedom
## AIC: 2733.8
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.5968992
```

```
temp <- data.frame("Nombre"="model_4_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 5

```
library(dplyr)

train.data  <- KNN_complete_Tuk_train
test.data <- KNN_complete_Tuk_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0776  -0.9753  -0.9359   1.3828   1.5225
##
```

```
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -0.5196204  0.0491599 -10.570   <2e-16 ***
## ph               0.0156204  0.0652486   0.239   0.8108
## Hardness         0.0189696  0.0572227   0.332   0.7403
## Solids          -0.0012915  0.0548812  -0.024   0.9812
## Chloramines      0.0302555  0.0549867   0.550   0.5822
## Sulfate         -0.0329265  0.0757348  -0.435   0.6637
## Conductivity    -0.0241911  0.0499926  -0.484   0.6285
## Organic_carbon  -0.0878045  0.0508895  -1.725   0.0845 .
## Trihalomethanes -0.0001904  0.0529227  -0.004   0.9971
## Turbidity        0.0090958  0.0499124   0.182   0.8554
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2357.4  on 1783  degrees of freedom
## Residual deviance: 2353.5  on 1774  degrees of freedom
## AIC: 2373.5
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.62486
```

```
temp <- data.frame("Nombre"="model_5_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 6

```
library(dplyr)

train.data  <- KNN_complete_sd_train
test.data <- KNN_complete_sd_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
```

```
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.1959  -1.0143  -0.9457   1.3300   1.5809
##
## Coefficients:
##                  Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -0.418018   0.043902  -9.522   <2e-16 ***
## ph               0.019789   0.048705   0.406   0.6845
## Hardness        -0.051998   0.044061  -1.180   0.2379
## Solids           0.079645   0.045353   1.756   0.0791 .
## Chloramines      0.068382   0.043273   1.580   0.1140
## Sulfate         -0.018591   0.051797  -0.359   0.7196
## Conductivity    -0.009369   0.044101  -0.212   0.8318
## Organic_carbon  -0.091400   0.044470  -2.055   0.0399 *
## Trihalomethanes  0.034061   0.044096   0.772   0.4399
## Turbidity       -0.031613   0.044541  -0.710   0.4779
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2935.9  on 2183  degrees of freedom
## Residual deviance: 2923.2  on 2174  degrees of freedom
## AIC: 2943.2
##
## Number of Fisher Scoring iterations: 4
```

```
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.6272894
```

```
temp <- data.frame("Nombre"="model_6_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 7

```
library(dplyr)

train.data  <- KNN_Tukey_Tuk_train
test.data <- KNN_Tukey_Tuk_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)
```

```r
# Mostramos el resumen de los resultados del modelo
summary(model)
```

```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0740  -0.9611  -0.9191   1.3913   1.5784
##
## Coefficients:
##                  Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -0.562632   0.050200 -11.208   <2e-16 ***
## ph               0.021396   0.061160   0.350    0.726
## Hardness        -0.031323   0.054037  -0.580    0.562
## Solids          -0.009757   0.053027  -0.184    0.854
## Chloramines      0.062370   0.052177   1.195    0.232
## Sulfate         -0.077079   0.072840  -1.058    0.290
## Conductivity    -0.010852   0.050479  -0.215    0.830
## Organic_carbon  -0.026694   0.049610  -0.538    0.591
## Trihalomethanes  0.059770   0.052288   1.143    0.253
## Turbidity       -0.014889   0.049430  -0.301    0.763
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2269.0  on 1729  degrees of freedom
## Residual deviance: 2264.3  on 1720  degrees of freedom
## AIC: 2284.3
##
## Number of Fisher Scoring iterations: 4
```

```r
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.6039261
```

```r
temp <- data.frame("Nombre"="model_7_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

Modelo 8

```r
library(dplyr)

train.data  <- KNN_sd_sd_train
```

```r
test.data <- KNN_sd_sd_test

# Creamos el modelo
model <- glm( Potability ~., data = train.data, family = binomial)

# Mostramos el resumen de los resultados del modelo
summary(model)
```

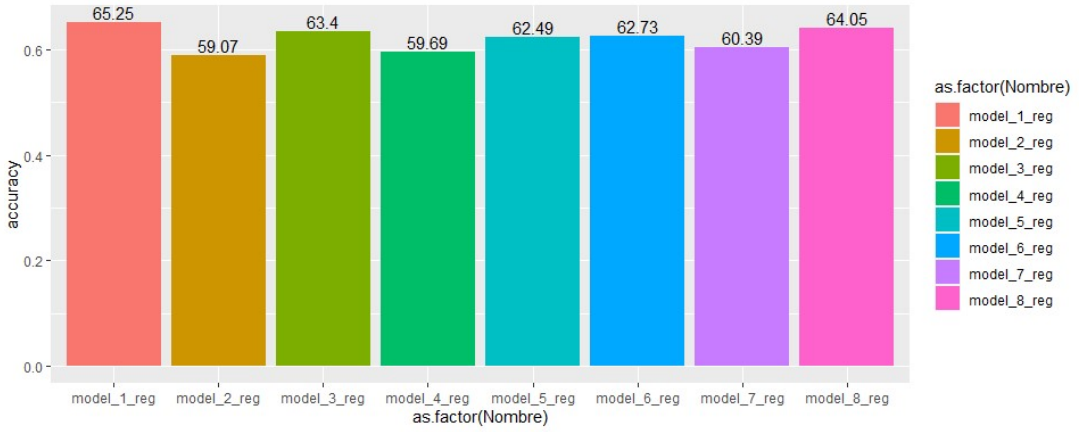
```
##
## Call:
## glm(formula = Potability ~ ., family = binomial, data = train.data)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.1277  -1.0043  -0.9624   1.3524   1.4997
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -0.4402740  0.0452067  -9.739   <2e-16 ***
## ph              -0.0227548  0.0519514  -0.438    0.661
## Hardness        -0.0431745  0.0468678  -0.921    0.357
## Solids           0.0471085  0.0475150   0.991    0.321
## Chloramines     -0.0334318  0.0461564  -0.724    0.469
## Sulfate         -0.0064331  0.0549887  -0.117    0.907
## Conductivity    -0.0468435  0.0452313  -1.036    0.300
## Organic_carbon  -0.0087337  0.0461575  -0.189    0.850
## Trihalomethanes  0.0360778  0.0470044   0.768    0.443
## Turbidity       -0.0004001  0.0454832  -0.009    0.993
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2761.3  on 2061  degrees of freedom
## Residual deviance: 2756.9  on 2052  degrees of freedom
## AIC: 2776.9
##
## Number of Fisher Scoring iterations: 4
```

```r
# Creamos las predicciones
probabilities <- model %>% predict(test.data, type = "response")
predicted.classes <- ifelse(probabilities > 0.5, 1, 0)

# Observamos la exactitud del modelo
result <- mean(predicted.classes == test.data$Potability)
result
```

```
## [1] 0.6405039
```

```r
temp <- data.frame("Nombre"="model_8_reg", "accuracy"=c(result))
accuracy_data_3 <- rbind(accuracy_data_3, temp)
```

```
ggplot(accuracy_data_3, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity") + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```

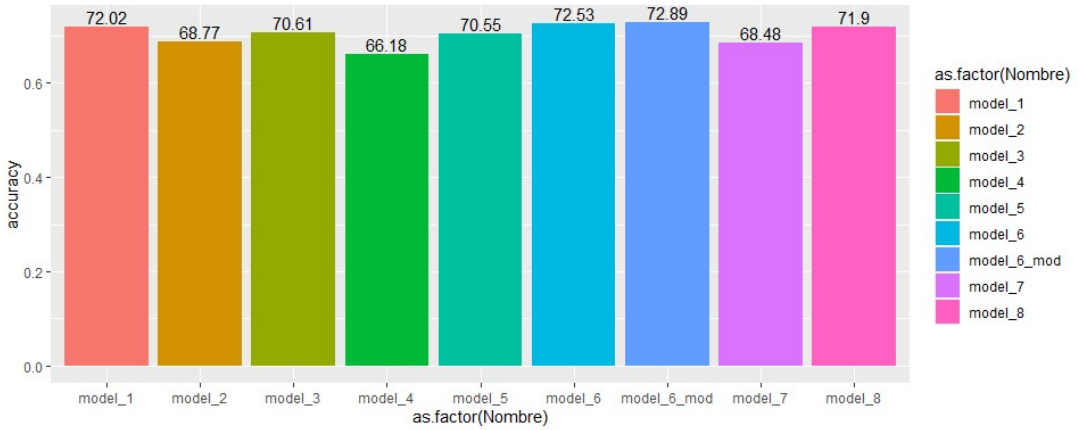


En este caso, utilizando la regresión logística para la clasificación, el mejor resultado lo hemos obtenido con el modelo 1, con una exactitud del 65.25%.

Representación de los resultados a partir de tablas y gráficas.

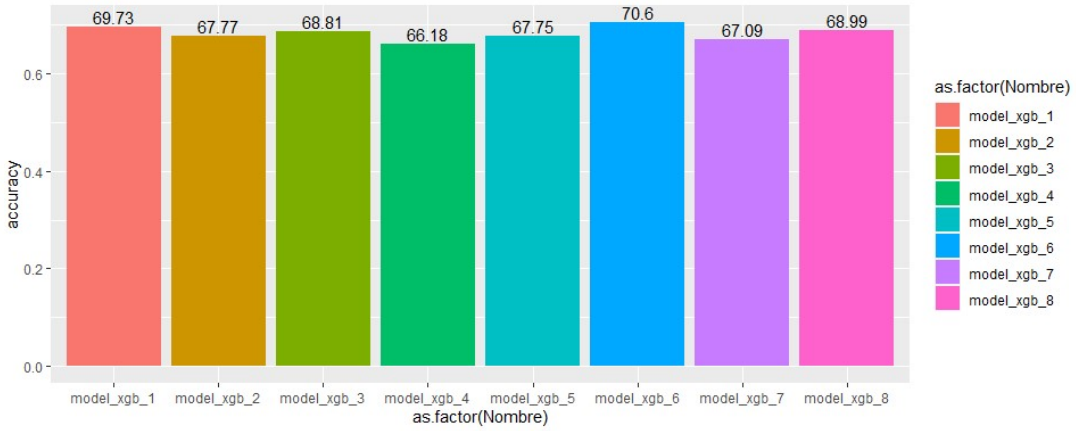
Podemos observar los resultados de los distintos modelos aplicados y sus resultados:

```
ggplot(accuracy_data, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity")  + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```

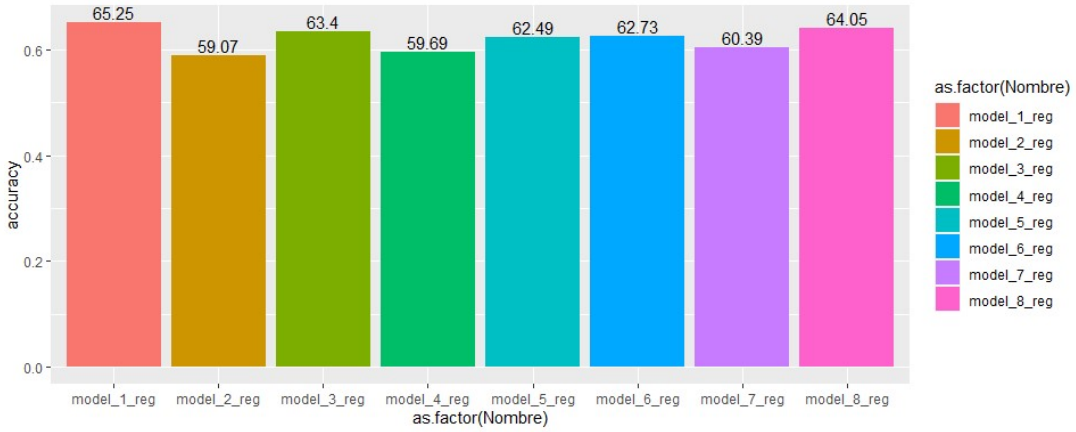



```
ggplot(accuracy_data2, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
```

```
  geom_bar(stat = "identity")  + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```

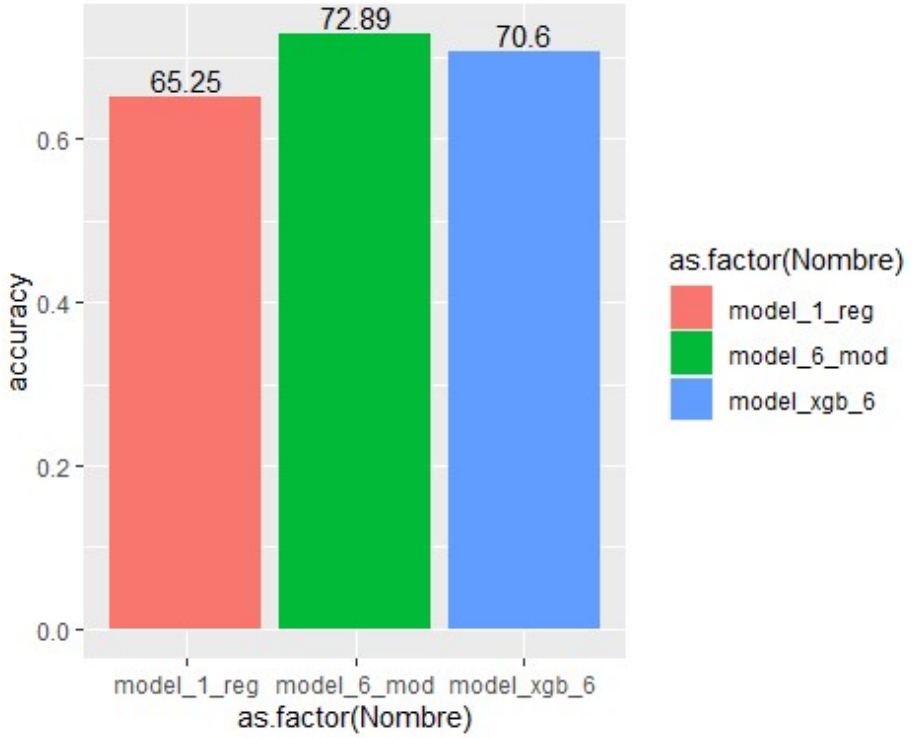



```
ggplot(accuracy_data_3, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity") + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```



Y comparamos los mejores resultados de los tres modelos de clasificación:

```
accuracy_data_final <- data.frame("Nombre"= c(accuracy_data[9,1],
accuracy_data2[6,1], accuracy_data_3[1,1]),
                                  "accuracy"= c(accuracy_data[9,2],
accuracy_data2[6,2], accuracy_data_3[1,2]))

ggplot(accuracy_data_final, aes(x=as.factor(Nombre), y=accuracy,
fill=as.factor(Nombre))) +
  geom_bar(stat = "identity") + geom_text(aes(label=round(accuracy*100,
digits = 2)), position=position_dodge(width=0.9), vjust=-0.25)
```



Tanto con randomForest como con Xgboost, el modelo 6 (KNN_complete_sd) nos ofrece los mejores resultados (por encima del 72%), con diferencias mínimas entre el modelo de entrenamiento y test. En el caso de regresión logistica obtenemos resultados bastante peores, como era de esperar.

Hemos enfocado la práctica en el impacto de las distintas estrategias de preparación de los datos en la posterior aplicación de los modelos de predicción, podemos observar claramente en las gráficas la variación de los resultados en función de dichas estrategias.

Conclusiones

En el caso estudiado, podemos sacar varias conclusiones: - En cuanto al tratamiento de outliers, hemos obtenido el mejor resultando excluyendo los valores que sobrepasan 3 desviaciones típicas - Respecto a los modelos de imputación, KNN y missForest nos han ofrecido los valores con menos RMSE respecto a los datos originales. - En cuanto a los algoritmos de clasificación, hemos obtenido los mejores resultados con randomForest, superando las soluciones propuestas hasta ahora en Kaggle, no es una diferencia muy significativa. El mejor resultado que podemos observar en Kaggle es del 68.5%, mientras que nosostros hemos logrado resultados

superiores al 72%. Tras el análisis de la naturaleza de los datos creemos que es dificil conseguir resultados significativamente superiores.

Bibliografía

Dealing with Missing Data using R https://medium.com/coinmonks/dealing-with-missing-data-using-r-3ae428da2d17

Flexible Imputation of Missing Data https://stefvanbuuren.name/fimd/

Root-Mean-Square Error in R Programming https://www.geeksforgeeks.org/root-mean-square-error-in-r-programming/

MissForest - missing data imputation using iterated random forests https://rpubs.com/lmorgan95/MissForest

Hmisc https://www.rdocumentation.org/packages/Hmisc/versions/4.5-0

Package 'Hmisc' https://cran.r-project.org/web/packages/Hmisc/Hmisc.pdf

Extra - Missing Data Tools https://unc-libraries-data.github.io/R-Open-Labs/Extras/Missing/missing.html

Package 'mi' https://cran.r-project.org/web/packages/mi/mi.pdf

Package 'xgboost' https://cran.r-project.org/web/packages/xgboost/xgboost.pdf

trainControl: Control parameters for train https://www.rdocumentation.org/packages/caret/versions/6.0-88/topics/trainControl

expand.grid: Create a Data Frame from All Combinations of Factor Variables https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/expand.grid

Simple R - xgboost - caret kernel https://www.kaggle.com/nagsdata/simple-r-xgboost-caret-kernel

XGBoost Multinomial Classification Iris Example in R https://rpubs.com/dalekube/XGBoost-Iris-Classification-Example-in-R

Tune Machine Learning Algorithms in R https://machinelearningmastery.com/tune-machine-learning-algorithms-in-r/

Tuning xgboost in R: Part I https://insightr.wordpress.com/2018/05/17/tuning-xgboost-in-r-part-i/

Tuning xgboost in R: Part II https://www.r-bloggers.com/2018/07/tuning-xgboost-in-r-part-ii/

Logistic Regression Essentials in R http://www.sthda.com/english/articles/36-classification-methods-essentials/151-logistic-regression-essentials-in-r/