Práctica final: kNN

Antonio Manuel Milán Jiménez

6 de marzo de 2019

Conjunto de datos

Antes de proceder al preprocesamiento es importante analizar el conjunto de datos con el que se está trabajando. De esta forma, se leen los datos, tanto 'train' como 'test', y se estudia su estructura:

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------------
```

```
## v ggplot2 3.1.0     v purrr   0.2.5
## v tibble  1.4.2     v dplyr   0.7.8
## v tidyr   0.8.2     v stringr 1.3.1
## v readr   1.1.1     v forcats 0.3.0
```

```
## -- Conflicts ------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(dplyr)
library(Hmisc)
```

```
## Loading required package: lattice
```

```
## Loading required package: survival
```

```
## Loading required package: Formula
```

```
##
## Attaching package: 'Hmisc'
```

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

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

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

#Lectura de train
datosTrain <- read.csv("train.csv", na.strings=c(" ","NA","?"))
#Lectura de test
datosTest <- read.csv("test.csv", na.strings=c(" ","NA","?"))

#Convertir a tibble
datosTrain <- as.tibble(datosTrain)
datosTest <- as.tibble(datosTest)
```

```r
#Dimensiones de los datos
dim(datosTrain)
```

```
## [1] 9144   51
```

```r
dim(datosTest)
```

```
## [1] 3919   50
```

```r
#Variables
colnames(datosTrain)
```

```
##  [1] "X1"  "X2"  "X3"  "X4"  "X5"  "X6"  "X7"  "X8"  "X9"  "X10" "X11"
## [12] "X12" "X13" "X14" "X15" "X16" "X17" "X18" "X19" "X20" "X21" "X22"
## [23] "X23" "X24" "X25" "X26" "X27" "X28" "X29" "X30" "X31" "X32" "X33"
## [34] "X34" "X35" "X36" "X37" "X38" "X39" "X40" "X41" "X42" "X43" "X44"
## [45] "X45" "X46" "X47" "X48" "X49" "X50" "C"
```

```r
colnames(datosTest)
```

```
##  [1] "X1"  "X2"  "X3"  "X4"  "X5"  "X6"  "X7"  "X8"  "X9"  "X10" "X11"
## [12] "X12" "X13" "X14" "X15" "X16" "X17" "X18" "X19" "X20" "X21" "X22"
## [23] "X23" "X24" "X25" "X26" "X27" "X28" "X29" "X30" "X31" "X32" "X33"
## [34] "X34" "X35" "X36" "X37" "X38" "X39" "X40" "X41" "X42" "X43" "X44"
## [45] "X45" "X46" "X47" "X48" "X49" "X50"
```

Los datos de entrenamiento consisten en 9144 instancias de 51 variables, mientras que los datos de test constan de 3919 instancias de 50 variables. Observando el nombre de las variables se comprueba que son precisamente la numeración de éstas. Finalmente, la última variable de los datos de entrenamiento, 'C', se corresponda de la variable de clase a predecir.

Si se estudian detenidamente las distribución de estas variables:

```r
Hmisc::describe(datosTrain[,1:7])
```

```
## datosTrain[, 1:7]
##
##  7  Variables      9144  Observations
## ---------------------------------------------------------------------------
## X1
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9118       26     9015        1    107.9    543.7    210.1    231.5
##      .25      .50      .75      .90      .95
##    276.8    331.0    388.7    441.5    471.4
##
## Value      -69000      0    500   1000
## Frequency      30   1374   7712      2
## Proportion  0.003  0.151  0.846  0.000
## ---------------------------------------------------------------------------
## X2
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9127       17     9018        1   -129.9    531.6    24.10    34.90
##      .25      .50      .75      .90      .95
##    60.91    96.48   136.53   177.76   206.35
##
## Value      -69000      0    500
## Frequency      31   8881    215
```

```
## Proportion  0.003  0.973  0.024
## ---------------------------------------------------------------------------
## X3
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9120       24     9014        1     8498     8785    492.8   1094.8
##      .25      .50      .75      .90      .95
##   3213.1   7231.5   9532.2  13581.2  22913.7
##
## lowest : -68932.50138 -68931.00000 -68929.97765 -68929.91993      3.05413
## highest: 194874.63000 208172.31000 210263.01000 229017.89284 327589.23000
## ---------------------------------------------------------------------------
## X4
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9125       19     8932        1   -214.7    471.4    14.07    14.99
##      .25      .50      .75      .90      .95
##    16.62    18.78    21.83    24.98    27.29
##
## Value      -69000      0
## Frequency      31   9094
## Proportion  0.003  0.997
## ---------------------------------------------------------------------------
## X5
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9126       18     4806    0.992   -233.5    467.7  -0.1442   0.0000
##      .25      .50      .75      .90      .95
##   0.0000   0.3670   0.9531   1.8087   2.4420
##
## Value      -69000      0
## Frequency      31   9095
## Proportion  0.003  0.997
## ---------------------------------------------------------------------------
## X6
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9109       35     7859    0.998   -223.5      477    0.000    0.000
##      .25      .50      .75      .90      .95
##    5.788    9.757   14.647   24.403   29.003
##
## Value      -69000      0
## Frequency      31   9078
## Proportion  0.003  0.997
## ---------------------------------------------------------------------------
## X7
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     9125       19     8983        1   -167.6    493.4    33.72    39.03
##      .25      .50      .75      .90      .95
##    48.92    63.69    81.58    98.94   108.86
##
## Value      -69000      0    500
## Frequency      31   9093      1
## Proportion  0.003  0.996  0.000
## ---------------------------------------------------------------------------
```

Sucede para todas las variables, aunque aquí sólo se han mostrado las primeras 7 variables, que un 0.3% de las instancias poseen un valor en todas ellas en torno a -69000; que observando el resto de la proporción de

valores se identifican rápidamente como 'ruido', valores posiblemente mal tomados que deberán ser tratados. Además, se muestra también que para todas las variables hay valores perdidos que igualmente tendrán que tratarse.

Para la variable 'C' se observa que no hay ningún valor perdido. Además, si se estudia su rango de valores:

```
unique(datosTrain["C"])
```

```
## # A tibble: 2 x 1
##       C
##   <int>
## 1     1
## 2     0
```

Se comprueba que sólo se tienen los valores de clase '0' y '1'. Gracias a que se indique que la media para esta variable predictora es de 0.344, ya se puede saber que para esta variable los datos no están todo lo bien balanceados que se quisiese, algo que se verá más adelante.

Por último, si se realiza este mismo estudio para los datos de test:

```
Hmisc::describe(datosTest[,1:7])
```

```
## datosTest[, 1:7]
##
##  7  Variables      3919  Observations
## ---------------------------------------------------------------------------
## X1
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3892        1    88.18    583.8    210.7    230.2
##      .25      .50      .75      .90      .95
##    275.5    330.4    387.7    441.8    477.0
##
## Value      -69000      0    500   1000
## Frequency      14    615   3288      2
## Proportion  0.004  0.157  0.839  0.001
## ---------------------------------------------------------------------------
## X2
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3884        1   -143.7    554.7    23.54    33.80
##      .25      .50      .75      .90      .95
##    59.33    95.90   135.53   172.03   198.72
##
## Value      -69000      0    500
## Frequency      14   3816     89
## Proportion  0.004  0.974  0.023
## ---------------------------------------------------------------------------
## X3
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3881        1     8749     9177    540.1   1201.0
##      .25      .50      .75      .90      .95
##   3314.8   7284.6   9690.2  13934.8  22052.2
##
## lowest : -68931.00000      3.91037      9.50240     11.72572     15.13274
## highest: 190385.49000 208939.59000 213595.71000 287563.71000 304910.31000
```

```
## ------------------------------------------------------------------------
## X4
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3861        1   -226.7    495.5    14.17    15.18
##      .25      .50      .75      .90      .95
##    16.73    18.79    21.78    25.17    27.56
##
## Value      -69000      0
## Frequency      14   3905
## Proportion  0.004  0.996
## ------------------------------------------------------------------------
## X5
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     2179    0.987   -245.6    491.7   0.0000   0.0000
##      .25      .50      .75      .90      .95
##   0.0000   0.3422   0.8953   1.7525   2.5409
##
## Value      -69000      0
## Frequency      14   3905
## Proportion  0.004  0.996
## ------------------------------------------------------------------------
## X6
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3323    0.997   -235.3      500    0.000    0.000
##      .25      .50      .75      .90      .95
##    5.865    9.826   14.373   23.466   28.521
##
## Value      -69000      0
## Frequency      14   3905
## Proportion  0.004  0.996
## ------------------------------------------------------------------------
## X7
##        n  missing distinct     Info     Mean      Gmd      .05      .10
##     3919        0     3879        1   -179.4    517.2    34.07    39.59
##      .25      .50      .75      .90      .95
##    49.42    64.07    81.77    98.66   109.26
##
## Value      -69000      0
## Frequency      14   3905
## Proportion  0.004  0.996
## ------------------------------------------------------------------------
```

Se descubre que sucede lo mismo: el 0.4% de las instancias poseen un valor en torno a -69000 para todas las variables que las hace determinar como 'ruido', siendo igualmente necesario que se traten de alguna forma. Sin embargo, para estos datos de test no hay ninguna instancia que tenga algún valor pérdido en alguna de sus variables.

Preprocesamiento

En esta sección se estudiarán diferentes técnicas utilizadas sobre los datos para 'prepararlos' para poder construir el modelo de clasificación sobre ellos.

Datos perdidos

Uno de los primeros problemas encontrados es que los datos de entrenamiento contienen instancias para las que en algunas de sus variables poseen valores perdidos. Estudiando la distribución de estos:

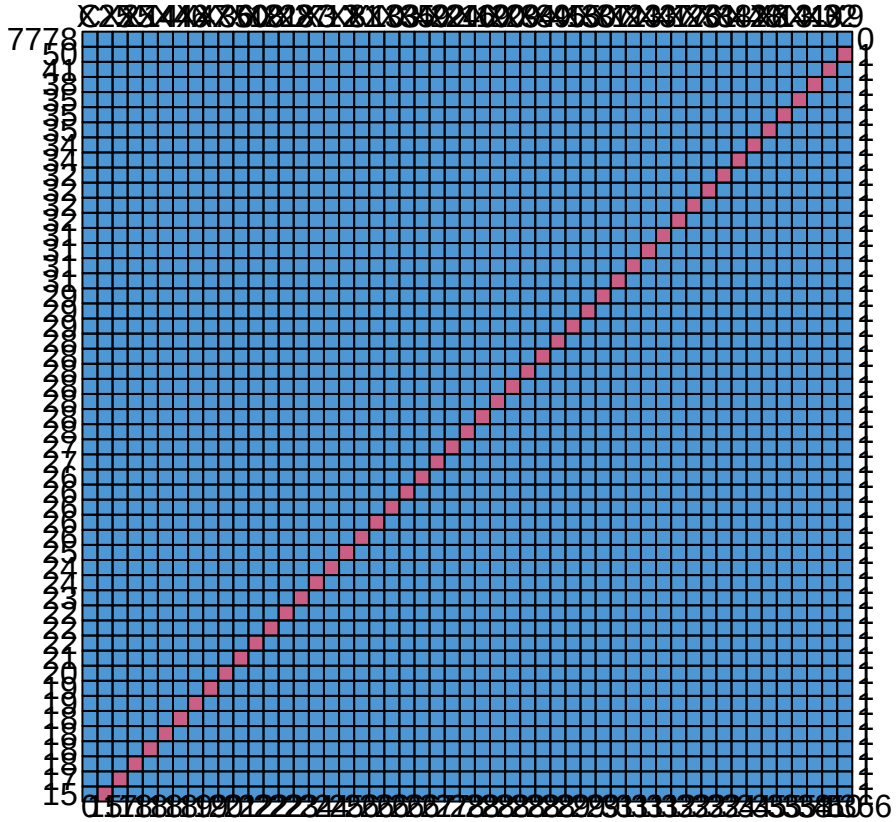
```
library(mice)
```

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

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

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

```
#Gráfico de los datos perdidos
patron <- mice::md.pattern(x=datosTrain,plot = TRUE)
```



```
#Distribución de instancias completas e incompletas
mice::ncc(datosTrain)
```

```
## [1] 7778
```

```r
mice::nic(datosTrain)
```

```
## [1] 1366
```

De las 9144 instancias en los datos de entrenamiento, 7778 de ellas no tienen ningún valor perdido por lo que se cuenta con 1366 instancias para las que sí existe este problema que, gracias al gráfico mostrado, se descubre que cada una de ellas sólo poseé un valor perdido, es decir, ninguna instancia tiene más de una variable para la que no se tenga un valor determinado. Esto es un punto interesante a la hora de elegir qué técnica emplear para este problema:

Eliminación de instancias con algún valor perdido

El método más simple, aunque a la vez más drástico, sería eliminar toda aquella instancia que posea algún 'missing value' de la forma:

```r
dim(na.omit(datosTrain))
```

```
## [1] 7778   51
```

Sin embargo, no resulta la mejor opción cuando hay demasiados datos que presentan este problema y, en este caso particular, cuando se ha observado que tan sólo hay un valor perdido para cada instancia, no resultando entonces una buena idea deshechar toda la instancia por un sólo valor.

Frente a esto surge otra técnica:

Imputación

Consiste en estimar el valor perdido en función del resto de datos para los que sí se tiene información acerca de esa variable. En este caso se ha elegido concretamente el método 'Amelia' que combina el algoritmo 'EM' (Expectation-Maximization) + Boostrap para poder estimar los datos perdidos del 'dataset'. Además, mediante el parámetro 'm', se puede indicar el número de 'datasets' creado para posteriormente hacer la media de los valores imputados y poder conseguir una mejor estimación.

```r
library(Amelia)
```

```
## Loading required package: Rcpp
```

```
## ##
## ## Amelia II: Multiple Imputation
## ## (Version 1.7.5, built: 2018-05-07)
## ## Copyright (C) 2005-2019 James Honaker, Gary King and Matthew Blackwell
## ## Refer to http://gking.harvard.edu/amelia/ for more information
## ##
```

```r
#Imputación mediante Amelia
imputados <- Amelia::amelia(datosTrain, m=3, parallel="multicore",noms="C")
```

```
## -- Imputation 1 --
##
##   1  2
##
## -- Imputation 2 --
##
##   1  2  3
##
```

```
## -- Imputation 3 --
##
##   1  2  3
```

```
datosTrain <- imputados$imputations$imp1
#Comprobación de datos incompletas
mice::nic(datosTrain)
```

```
## [1] 0
```

```
dim(datosTrain)
```

```
## [1] 9144   51
```

Se observa que ya no hay ninguna instancia incompleta y que se han conservado todas ellas.

Eliminación de ruido

Como se ha observado anteriormente, en torno al 0.3%-0.4% de las instancias se podían considerar directamente como ruido (valores en torno a -69000). No obstante, probablemente haya otras instancias también que se puedan considerar como ruido por alguno de sus valores más 'anómalos'. Para solventar este problema, en los datos de entrenamiento, se eliminarán aquellas instancias que se consideren como ruido y que dificultarían el aprendizaje del modelo, siempre que se posean los suficientes datos.

IPF

En este estudio se ha empleado el método 'IPF' para la eliminación de ruido, empleando éste modelo log-lineales que iterativamente se ajustan y van determinando qué instancias se podrían considerar como ruido.

```
library(NoiseFiltersR)
datosTrainConRuido <- datosTrain

#Se estructuran los datos para que sean aceptados por el método
datosTrain <- as.data.frame(datosTrain)
datosTrain[,ncol(datosTrain)] <- as.factor(datosTrain[,ncol(datosTrain)])

#Se realiza IPF
out <- IPF(C~.,data=datosTrain)
```

```
## Iteration 1: 896 noisy instances removed
## Iteration 2: 211 noisy instances removed
## Iteration 3: 139 noisy instances removed
## Iteration 4: 108 noisy instances removed
## Iteration 5: 88 noisy instances removed
## Iteration 6: 80 noisy instances removed
## Iteration 7: 74 noisy instances removed
```

```
length(out$remIdx)
```

```
## [1] 1596
```

```
#Se actualizan los datos eliminando aquellas instancias detectadas como anomalías
datosTrain <- as.tibble(datosTrain[setdiff(1:nrow(datosTrain),out$remIdx),])
datosTrain[,ncol(datosTrain)] <- as.numeric(unlist(datosTrain[,ncol(datosTrain)]))
```

Se detectan un total de 1596 instancias como ruido que son eliminadas.

Detección de anomalías

Otro punto interesante es detectar anomalías que haya en el conjunto de datos mediante 'kMeans', calculando la distancia que haya de las instancias a los diferentes centroides. Así, se identificaron como anomalías aquellas instancias que tengan datos extremos o que sea una combinación de dos o más variables lo que las convierte en instancias inusuales. Dado que no se sabe cuántas anomalías puede haber, se determina un porcentaje de forma que las 'n' instancias con mayor distancia se consideren outliers:

```
#Se van a detectar el 10% de las instancias como outliers
numero.de.outliers = as.integer(dim(datosTrainConRuido)[1]/10)

#
modelo.kmeans <- kmeans(datosTrainConRuido,2)
indices.clustering <- modelo.kmeans$cluster
centroides <- modelo.kmeans$centers

distancias_a_centroides = function (datos,indices.asignacion.clustering,datos.centroides){
      sqrt(rowSums(   (datos - datos.centroides[indices.asignacion.clustering,])^2   ))
}

dist.centroides <- distancias_a_centroides(datosTrainConRuido, indices.clustering, centroides)
top.outliers <- order(dist.centroides, decreasing = TRUE)[1:numero.de.outliers]
datosTrainConRuido <- datosTrainConRuido[-top.outliers,]
dim(datosTrainConRuido)
```

```
## [1] 8230   51
```

De esta forma se eliminarían los datos considerados más ruidosos, dado un determinado porcentaje, en función de los anómalos que sean los valores de éstos.

Eliminación de ruido en datos de test

El tratamiento del ruido en los datos de test tiene que ser diferente pues no se puede eliminar directamente una instancia de test. Para solucionarlo se detecta primero aquellas instancias de test que poseían esos valores 'ruidosos' en torno a -69000:

```
ruido <- which(datosTest[,1] < -68900)
ruido
```

```
##  [1]  111  231  660  848  923 1213 1822 2107 2176 2783 2870 3460 3468 3810
```

Sabiendo ya las instancias con ruido, se procede a sustituir este ruido por la mediana del resto de valores de los datos que no son ruido:

```
medianas <- apply(datosTest[-ruido,],2,median)
cambio <- matrix(rep(medianas,length(ruido)),length(ruido),ncol(datosTest),byrow=TRUE)
datosTest[ruido,]<-cambio
```

Normalización

Por supuesto, otro punto interesante en el preprocesamiento es la normalización de los datos. Esto se realiza para evitar que la distribución y el rango en el que se encuentran las variables influyan en el modelo de modo que prevalezcan unas sobre otras simplemente por este hecho.

```
require(caret)
```

```
## Loading required package: caret
```

```
##
## Attaching package: 'caret'
```

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

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

```
#Se estima el centrado y escalado en función de los datos de entrenamiento
datosPreprocesados <- caret::preProcess(datosTrain[,1:50],method=c("center","scale"))

#Se centran y escalan los datos de entrenamiento
datosTransformados <- predict(datosPreprocesados,datosTrain[,1:50])

#Se añade la variable predecir que no se puede modificar
datosTrain <- cbind(datosTransformados,datosTrain[,51])

#Se centran y escalan los datos de test en función del centrado y escalado que se había calculado
datosTest <- predict(datosPreprocesados,datosTest)
datosTrain <- as.tibble(datosTrain)
datosTest <- as.tibble(datosTest)
```

Selección de variables

Este conjunto de datos consta de 50 variables predictoras. Sin embargo, no necesariamente las 50 variables van a ser igual de independientes entre ellas y todas vayan a ser determinantes para contruir el modelo y determinar la variable de salida. Perfectamente puede suceder que dos variables sean muy similares entre ellas, aportando la misma información para el modelo y siendo redundante utilizar ambas. Otro caso que puede darse es que simplemente una variable no aporte información útil para estimar la variable a predecir y podamos entonces prescindir de ella.

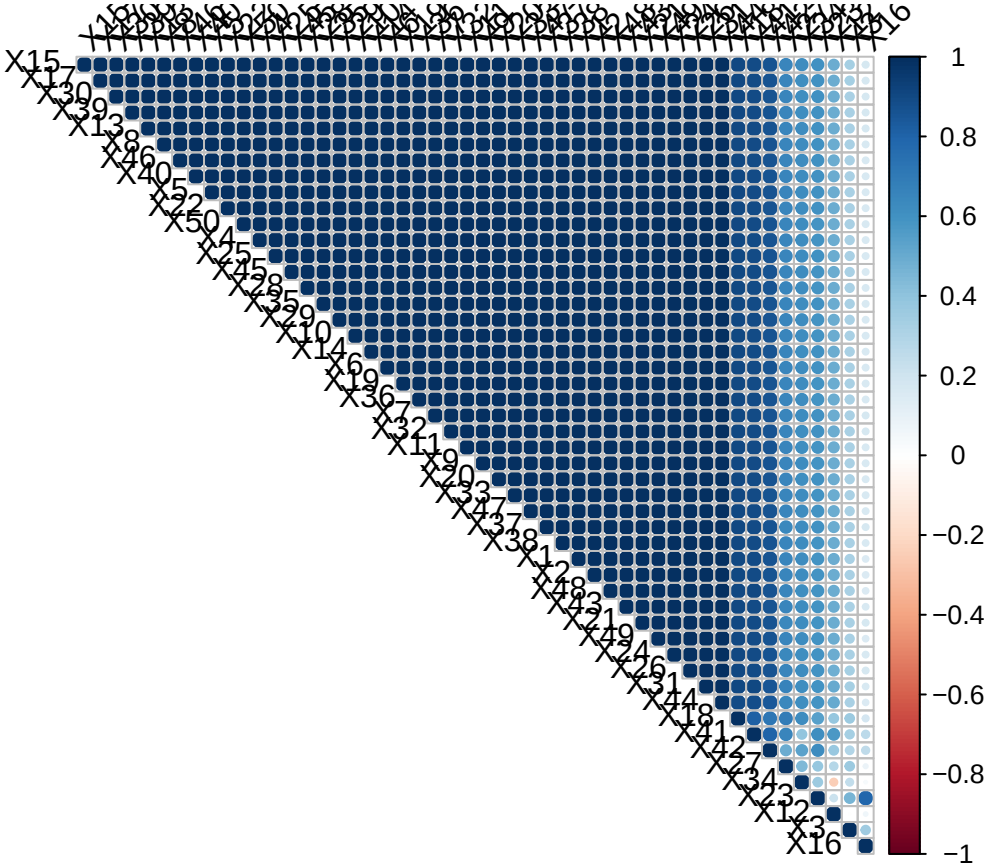
Esto hace que sea interesante seleccionar solamente aquellas variables que sean realmente útiles para poder así disminuir la complejidad del modelo e incluso, en algunos casos, mejorar el comportamiento de éste.

Correlación de variables

Una técnica para la selección de variables consiste en estudiar la correlación entre ellas, descubrir cómo de similares son entre ellas y eliminar así aquellas variables ya estén siendo representadas por otras variables respecto a la información que aportan.

```
#Construcción matriz de correlación

limite <- ncol(datosTrain)-1
```

```
corrMatrix <- cor(na.omit(datosTrain[,1:limite]))
corrplot::corrplot(corrMatrix,order="FPC",type="upper",tl.col="black",tl.srt=45)
```



Gracias al rango de colores de este gráfico descubrimos que un alto porcentaje de las variables están correladas en gran medida, cercanas a 1. Esto ya indica que hay bastante información irrelevante debido a que ya aparece en otras variables y que, aún con un filtrado bastante estricto, se reducirá considerablemente la dimensionalidad de los datos.

Empezando con un filtrado en el que se eliminen aquellas variables con una correlación mayor del 90%:

```
altamenteCorreladas <- caret::findCorrelation(corrMatrix, cutoff=0.9)
length(altamenteCorreladas)
```

```
## [1] 40
```

Se descubre que hay 40 variables, de las 50 presentes en el 'dataset', que cumplen esta condición; una correlación muy alta que ya se presentaba en el gráfico anterior. Probando ahora con un filtrado mucho más estricto:

```
altamenteCorreladas <- caret::findCorrelation(corrMatrix, cutoff=0.99)
length(altamenteCorreladas)
```

```
## [1] 40
```

Incluso con un filtrado de aquellas variables que presentan una correlación del 99%, se vuelven a tener 40 variables que lo cumplen. Teniendo en cuenta este hecho, se puede optar por este filtrado en el que se reduciría el 'dataset' de 50 variables predictoras a 10 o, por el contrario, si se quiere conservar más variables aumentar

aun más el filtrado:

```
altamenteCorreladas <- caret::findCorrelation(corrMatrix, cutoff=0.9999)
length(altamenteCorreladas)
```

```
## [1] 28
```

Es necesario una correlación del 99.99% para poder filtrar de 50 variables a 28 que lo cumplan.

Una vez decidido el filtrado, se aplica tanto a los datos de entrenamiento como a los de test:

```
datosTrainFiltrados <- datosTrain[,-altamenteCorreladas]
datosTestFiltrados <- datosTest[,-altamenteCorreladas]
dim(datosTrainFiltrados)
```

```
## [1] 7548   23
```

```
dim(datosTestFiltrados)
```

```
## [1] 3919   22
```

Y así se realizaría este método de selección de variables.

CFS

Otro método de selección de variables es mediante un filtrado CFS, en el que se combinan tanto medidas de correlación como de entropía entre variables:

```
library(FSelector)

subset <- FSelector::cfs(C ~ .,datosTrain)
subset
```

```
## [1] "X16"
```

Sin embargo, dada la alta correlación ya vista que presenta este conjunto de datos, deshecha todas las variables y únicamente selecciona la variable 'X16' por lo que no resulta tan útil en este caso particular.

Consistency

Otra posibilidad es hacer la selección en función de la consistencia que haya entre las variables, algo que realiza el método 'consistency':

```
subset <- consistency(C~.,datosTrain)
```

Las variables finalmente seleccionadas por este método son:

"X1" "X3" "X5" "X7" "X9" "X10" "X13" "X15" "X18" "X20" "X22" "X26" "X28" "X30" "X32" "X38" "X42" "X49"

Una selección interesante para poder reducir la dimensionalidad del conjunto de datos sin perder información sobre él.

Relief

Con esta técnica se asocian pesos a cada una de las variables en función de la distancia que haya entre las instancias. Con estos pesos asociados, se apoya en la función 'cutoff' para seleccionar finalmente las variables:

```
weights <- FSelector::relief(C ~.,datosTrain,neighbours.count = 30,sample.size = 120)
subset <- FSelector::cutoff.k(weights,2)
```

Sin embargo, nuevamente se han eliminado demasiadas variables, seleccionando únicamente las variables 'X3' y 'X16'. Posiblemente, al haber una correlación tan alta entre las variables, se haya detectado una distancia muy baja y se hayan determinado únicamente estas dos variable como relevantes.

PCA

También se puede aplicar uno de los métodos más famosos de selección de atributos, el Análisis de Componentes Principales, en el que se estudia cuánto aporta cada una de las variables a las componentes principales y en función de ello se pueden seleccionar la variables más relevantes para el modelo.

```
library(FactoMineR)
library(factoextra)
limite <- ncol(datosTrain)-1
res.pca <- PCA(datosTrain[,1:limite],graph = FALSE)

#Se seleccionan aquellas componentes que conformen el 98% del total, consiguiendo así las
#componentes más importantes
topComponentes <- unname(which(res.pca$eig[,3]>98)[1])

#Para estas componentes escogidas, se obtiene la contribución de las variables a ellas
contrib <- fviz_contrib(res.pca,choice="var",axes=1:topComponentes)

#Gracias a que se devuelven las variables ordenadas en función de su contribución, se
#escogen los dos primeros tercios, es decir, el 66% de las variables que en función de
#su contribución a las componentes son más importantes
topVars <- as.integer(ncol(datosTrain)/1.5)
subset <- contrib$data$name[order(contrib$data$contrib,decreasing = TRUE)][1:topVars]
subset <- unlist(lapply(subset,toString))
subset
```

```
##  [1] "X28" "X50" "X5"  "X40" "X4"  "X39" "X45" "X10" "X19" "X22" "X8"
## [12] "X6"  "X46" "X25" "X35" "X36" "X29" "X30" "X17" "X9"  "X32" "X11"
## [23] "X14" "X15" "X7"  "X20" "X13" "X33" "X2"  "X1"  "X38" "X47" "X37"
## [34] "X43"
```

Estas son las variables seleccionadas por PCA. Por supuesto, se pueden escoger diferentes proporciones en cuanto a las componentes que escoger y a las contribuciones por parte de las variables, para encontrar otras selecciones de variables.

Una vez determinada la selección de variables, se aplica sobre los datos de entrenamiento y los de test:

```
datosTrain <- datosTrain[,c(subset,"C")]
datosTest <- datosTest[,subset]
datosTrain[,"C"] <- datosTrain[,"C"] - 1
dim(datosTrain)
```

```
## [1] 7548   35
```

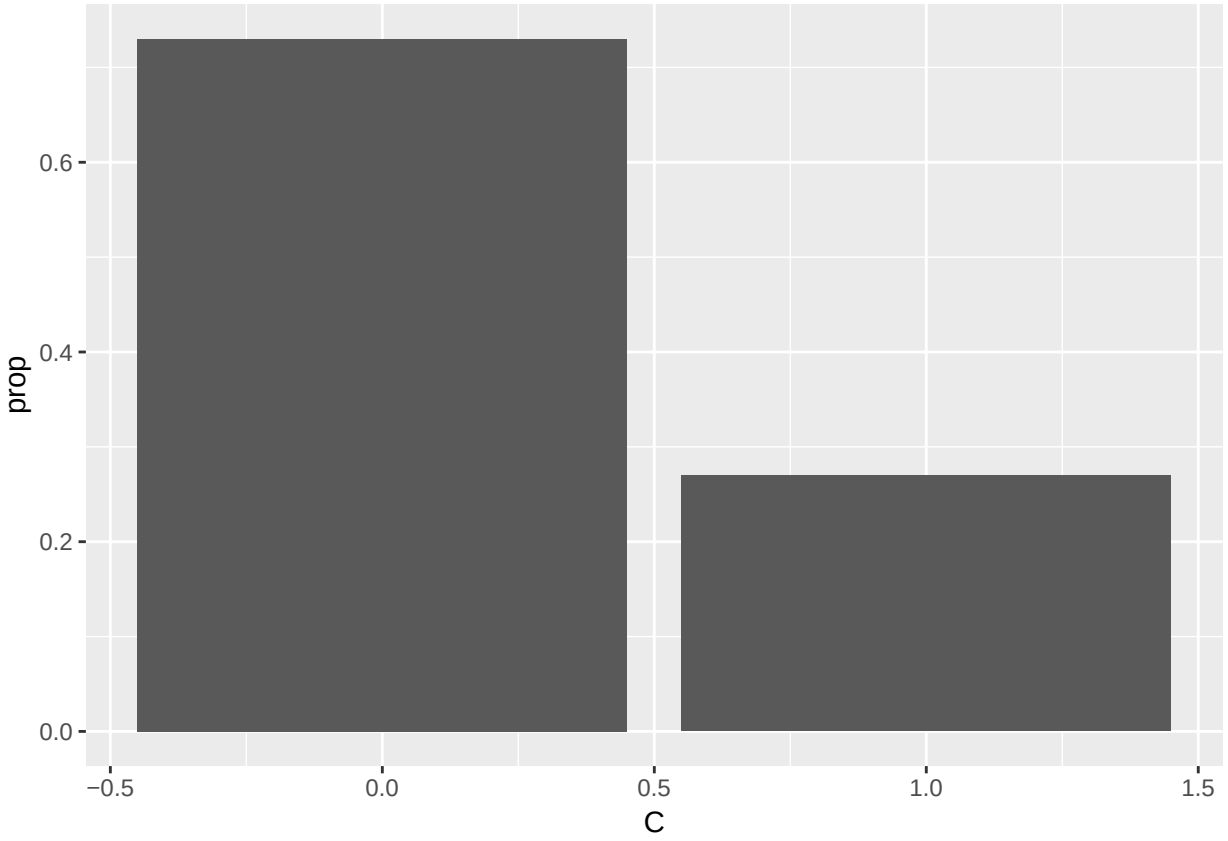
```
dim(datosTest)
```

```
## [1] 3919   34
```

Balanceo de clases

Algo que ya se pudo adelantar anteriormente al analizar el conjunto de datos es que para la variable de clase C, los datos no estaban todo lo bien balanceados que se quisiese. Es decir, hay considerablemente más instancias de una clase que otra, algo que puede resultar en que el modelo no aprenda correctamente para ambas clases:

```
ggplot(data=datosTrain) + geom_bar(mapping =aes(x=C,y=..prop..,group=1))
```




```
resumen <- dplyr::group_by(datosTrain,C) %>% dplyr::summarise(nc=n())
resumen
```

```
## # A tibble: 2 x 2
##       C    nc
##   <dbl> <int>
## 1     0  5512
## 2     1  2036
```

Efectivamente estos datos presentan un desbalanceo considerable, habiendo 5512 instancias para una clase y 2036 instancias para la otra, es decir, una relación de 73% a 27%.

Para solventar este problema se pueden utilizar técnicas de 'undersampling', en las que se disminuye el número de instancias para la clase mayoritaria, técnicas de 'oversampling', en las que se aumenta el número

de instancias para la clase minoritaria, e incluso una hibridación de ambas técnicas.

Uno de los métodos más utilizados es 'SMOTE', en el que a partir de dos instancias se calcula y se crea una nueva instancia para esa clase minoritaria:

```
library(DMwR)
```

```
## Loading required package: grid
```

```
datosTrainBalanceados <- datosTrain
datosTrainBalanceados$C <- as.factor(datosTrainBalanceados$C)
datosTrainBalanceados <- SMOTE(C~., as.data.frame(datosTrainBalanceados), perc.over=100)
resumen <- dplyr::group_by(datosTrainBalanceados,C) %>% dplyr::summarise(nc=n())
print(resumen)
```

```
## # A tibble: 2 x 2
##   C        nc
##   <fct> <int>
## 1 0      4072
## 2 1      4072
```

Ahora sí se tendrían los datos de entrenamiento balanceados.

Por supuesto, existen muchas otras técnicas para conseguir el balanceo de los datos. Así, haciendo uso de la función 'oversample' se pueden emplear métodos tales como 'MWMOTE' o 'DBSMOTE' entre otros:

```
library(imbalance)
datosTrainBalanceados <- imbalance::oversample(as.data.frame(datosTrain), ratio = 0.85,
                                               method = "MWMOTE", classAttr = "C")
resumen <- dplyr::group_by(datosTrainBalanceados,C) %>% dplyr::summarise(nc=n())
print(resumen)
```

```
## # A tibble: 2 x 2
##       C    nc
##   <dbl> <int>
## 1     0  5512
## 2     1  4686
```

```
datosTrainBalanceados <- imbalance::oversample(as.data.frame(datosTrain), ratio = 0.85,
                                               method = "ADASYN", classAttr = "C")
resumen <- dplyr::group_by(datosTrainBalanceados,C) %>% dplyr::summarise(nc=n())
print(resumen)
```

```
## # A tibble: 2 x 2
##       C    nc
##   <dbl> <int>
## 1     0  5512
## 2     1  4686
```

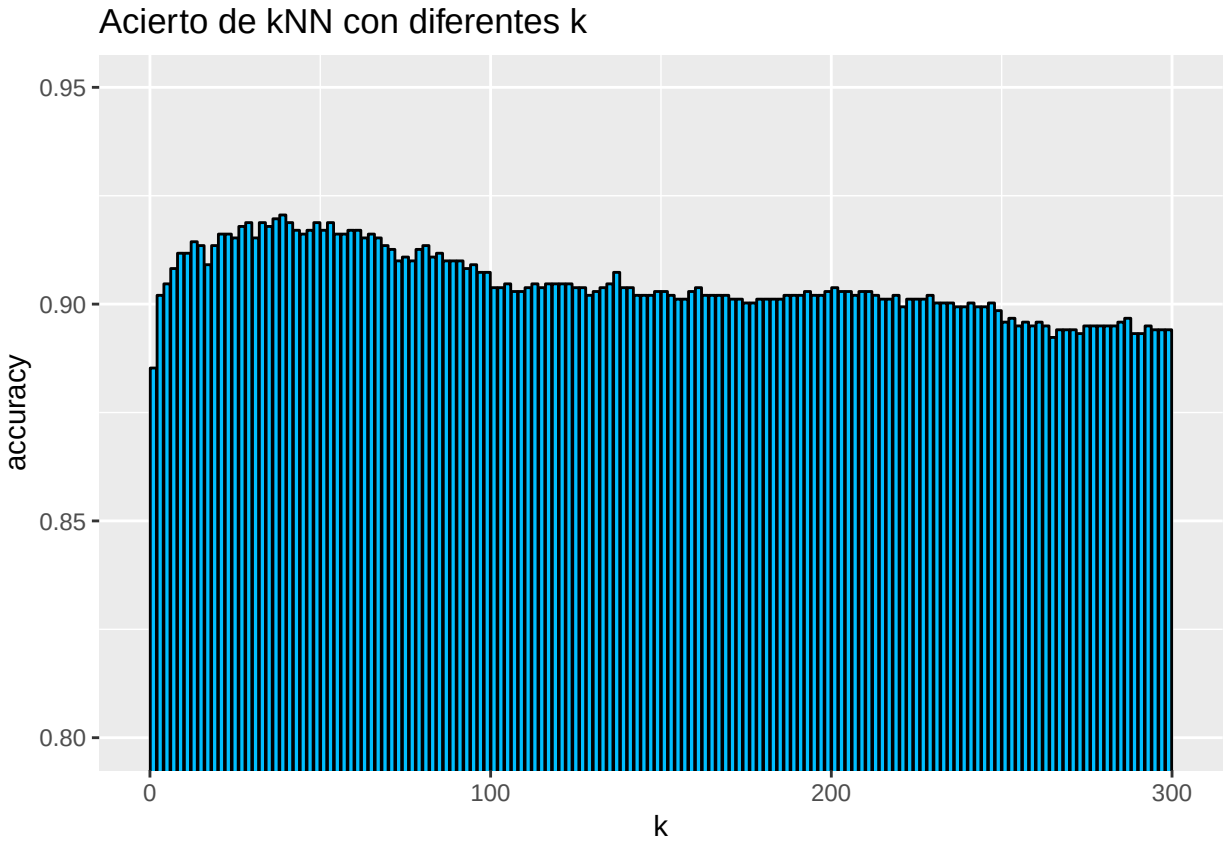
Clasificación

Una vez preprocesados los datos, ya se puede proceder a contruir el modelo que determinará la clase para los datos de test. Para construcción del modelo se ha escogido el algoritmo kNN, que determina la clase de una instancia en función de sus 'k' vecinos más cercanos. Para este algoritmo, el parámetro más significativo es 'k', el número de 'vecinos' más cercanos a tener en cuenta para determianr la clase correspondiente.

La determinación de este parámetro se puede hacer experimentalmente, simplemente entrenando el modelo con diferentes 'k' y observando con cual o cuales se obtienen mejores resultados. Para saber qué acierto se está obteniendo, se realizará validación cruzada sobre los datos de entrenamiento para tener una idea sobre cómo de bien o mal funciona el modelo.

```
#Se divide los datos de entrenamiento en pseudoTrain y pseudoTest a relación de 85%-15%
misDatosTrain <- datosTrain
shuffled <- sample(dim(misDatosTrain)[1])
eightypct <- (dim(misDatosTrain)[1] * 85) %/% 100
datosTrainPseudo <- misDatosTrain[shuffled[1:eightypct], 1:(dim(misDatosTrain)[2]-1)]
datosTestPseudo <- misDatosTrain[shuffled[(eightypct+1):dim(misDatosTrain)[1]],
                                 1:(dim(misDatosTrain)[2]-1)]
datosTrainPseudo_labels <- misDatosTrain[shuffled[1:eightypct], dim(misDatosTrain)[2]]
datosTrainPseudo_labels <-as.data.frame(datosTrainPseudo_labels)[,1]
datosTestPseudo_labels <- misDatosTrain[shuffled[(eightypct+1):dim(misDatosTrain)[1]],
                                        dim(misDatosTrain)[2]]
datosTestPseudo_labels <- as.data.frame(datosTestPseudo_labels)[,1]


getKnn <- function(miK){

  #Se entrena el modelo con la 'k' indicada
  test_pred <- knn(train = datosTrainPseudo, test = datosTestPseudo,
                   cl = datosTrainPseudo_labels, k=miK)

  #Se estructura la predicción realizada
  test_pred <- as.vector(test_pred)
  test_pred[which(is.na(test_pred))] <- 2
  test_pred <- as.numeric(test_pred)

  #Se contruye la matriz de confusión y se devuelve el acierto obtenido por el modelo
  u <- union(test_pred, datosTestPseudo_labels)
  t <- table(factor(test_pred, u), factor(datosTestPseudo_labels, u))
  cm<-confusionMatrix(t)
  cm$overall["Accuracy"]

}


result1 <- lapply(1:150*2-1,getKnn)
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



Así, rapidamente se descubre que los mejores resultados se obtienen con un 'k' en torno 20-100. Con una 'k' menor que 20 los resultados son peores y a partir de 100 se aprecia una pérdida progresiva del acierto. Si se observa detenidamente los 30 mejores resultados:

```
(1:150*2-1)[order(r1,decreasing=TRUE)[1:30]]
```

```
##  [1] 39 37 29 33 41 49 53 27 35 43 47 51 59 61 21 23 45 55 57 65 25 31 63
## [24] 67 13 15 19 69 81 71
```

En terminos generales, los mejores aciertos se obtienen con una 'k' en torno a 30, por lo que serán los valores más interesantes para construir el modelo final que realice la predicción de los datos de test.

Otro parámetro del algoritmo kNN es el parámetro 'l', que indica el número de votos mínimos por parte de los vecinos para tomar una decisión sobre la instancia a clasificar. En caso de que no se consiga el mínimo de votos, la instancia se clasifica como 'duda'; algo interesante pero que en este caso particular sólo se quiere una clasificación binaria, por lo que este parámetro se mantendrá en 0.

Resultados obtenidos

Aquí se presentan los aciertos obtenidos por los diferentes modelos construidos sobre los datos de test, es decir, los resultados obtenidos en la plataforma de Kaggle:

Amelia	IPF	Supr. Anomalias	Normalizacion	Filtrado Corr.	Seleccion	Balanceo	k	Acierto
folds=1				0.9999			7	84.07%
folds=1				0.99999		MWMOTE	7	81.62%
folds=1		X (5%)		0.99999		MWMOTE	7	78.05%
folds=1		X (5%)	X	0.999999		MWMOTE	7	78.66%
folds=1			X	0.9999			7	85.29%
folds=1		X (5%)	X	0.9999			7	85.04%
folds=1			X	0.99			7	85.34%
folds=1			X	0.85			7	84.89%
folds=1			X	0.999999			7	85.29%
folds=1			X	0.9999			50	87.59%
folds=1			X	0.9999			23	88%
folds=1			X	0.999999			23	88.1%
folds=1			X	0.999999		MWMOTE	23	83.97%
folds=1			X	0.999999		ADASYN	23	82.08%
folds=3			X	0.9999			31	85.6%
folds=3			X	0.9999		SMOTE	31	80.8%
folds=3			X	0.9999		DBSMOTE	31	84.63%
folds=1		X (3%)	X	0.999999			23	87.28%
folds=1		X (3%)	X	0.999999			28	87.54%
folds=3			X	0.999999			23	87.79%
folds=1			X		CFS		23	80.55%
folds=1			X	0.999999			293	86.47%
folds=1	X		X	0.999999			23	87.23%
folds=3	X		X	0.9999			23	88.1%
folds=3	X		X	0.9999		DBSMOTE	23	86.83%
folds=3	X		X	0.9999			200	86.57%
folds=3	X		X				23	85.8%
folds=3	X	X (2%)	X	0.9999			23	87.69%
folds=3	X		X		Consistency		23	87.13%
folds=3	X		X	0.9999	Consistency		23	87.9%
folds=3	X		X	0.999999	Consistency		23	85.29%
folds=3	X		X		PCA 0.9		23	87.64%
folds=3	X		X		PCA 0.95		23	87.44%
folds=3			X		PCA 0.9		23	86.98%
folds=3	X		X		PCA 0.9		31	88.46%
folds=3	X		X		PCA 0.97		180	87.08%
folds=3	X		X		PCA 0.97		23	88.56%
folds=3	X		X		PCA 0.98		19	89.38%
folds=3	X		X		PCA 0.98		11	88.71%
folds=3	X		X		PCA 0.98		28	89.53%
folds=3	X		X		PCA 0.95		28	88.71%
folds=3	X		X		PCA 0.98		43	88.76%
folds=5	X		X		PCA 0.98		28	89.38%
folds=3	X		X		PCA 0.98	DBSMOTE	28	87.69%

Los mejores resultados se obtuvieron al realizar imputación con Amelia utilizando 3 'folds', realizando IPF para la eliminación de ruido, normalizando los datos, empleando PCA para seleccionar los atributos que representasen el 98% de las componentes principales y una 'k' entre 20 y 30.