Analizando el dataset Census Income

Marta Gómez y Braulio Vargas

3 de junio de 2016

Índice

1 Definición del problema a resolver y enfoque elegido 1
1.1 Definición del problema . . . 1
1.2 Enfoque elegido . . . 3
2 Codificación de los datos de entrada para hacerlos útiles a los algoritmos 3
2.1 Valores perdidos . . . 3
2.2 Asignación de valores numéricos a los valores categóricos . . . 5
2.3 Eliminando la variable `education.num` . . . 6
3 Ajustando un modelo lineal 6
3.1 Idoneidad del modelo lineal . . . 6
3.2 Dimensión del modelo lineal . . . 6
3.3 Parámetros del modelo . . . 6
4 Ajustando un modelo *Random Forest* 8
4.1 Idoneidad del modelo *Random Forest* para los datos del problema . . . 8
4.2 Parámetros del modelo . . . 9
4.3 Hiperparámetros del modelo . . . 9
4.4 Complejidad del modelo . . . 9
4.5 Aplicación del modelo *Random Forest* . . . 9
5 Ajustando un modelo de base radial 12
5.1 Support Vector Machine . . . 12
5.2 Vecino más cercano con base radial . . . 15
6 Ajustando un modelo de *Red Neuronal* 18
6.1 Idoneidad del modelo de *Red Neuronal* para los datos del problema . . . 18
6.2 Normalización de los datos . . . 19
6.3 Parámetros e Hiperparámetros del modelo . . . 19
6.4 Aplicación del modelo . . . 19
7 Comparación de los modelos obtenidos 20
7.1 Curvas ROC de los modelos obtenidos . . . 21
8 Selección del mejor modelo 22
9 Ajustando un modelo de red neuronal recuperando los datos perdidos 23
9.1 Recuperando datos perdidos . . . 23
9.2 Ajustando una red neuronal con el dataset completo . . . 24
Referencias 25

1 Definición del problema a resolver y enfoque elegido

1.1 Definición del problema

La base de datos escogida se denomina *Census Income*, aunque también es conocida como *Adult* [1]. Se puede descargar aquí. Esta base de datos tiene 48842 instancias, cada una con 14 atributos. El problema es que también tiene valores perdidos.

El problema consiste en predecir cuando una persona ingresa más de 50.000 dólares al año, basándonos en los datos del censo. Por tanto, estamos ante un problema de *clasificación binaria*.

Los atributos de los que disponemos son:

- ***age***: atributo numérico que expresa la edad de la persona.
- ***workclass***: atributo categórico que expresa el tipo de empleo que tiene la persona. Los valores que puede tomar son:
  - *Private*: persona contratada en una empresa privada.
  - *Self-emp-not-inc*: autónomo.
  - *Self-emp-inc*: persona que tiene una empresa grande.
  - *Federal-gov*: funcionario del gobierno federal.
  - *Local-gov*: funcionario del gobierno local.
  - *State-gov*: funcionario del gobierto estatal.
  - *Without-pay*: persona en paro.
  - *Never-worked*: persona que nunca ha trabajado.
- ***fnlwgt (final weight)***: variable numérica que representa un peso, cada peso corresponde a una característica socio-económica de la población, por tanto, personas con características demográficas parecidas deben tener un peso parecido. [2]
- ***education***: variable categórica que representa el nivel de estudios de la persona. Los valores que puede tomar son: *Bachelors*, *Some-college*, *11th*, *HS-grad*, *Prof-school*, *Assoc-acdm*, *Assoc-voc*, *9th*, *7th-8th*, *12th*, *Masters*, *1st-4th*, *10th*, *Doctorate*, *5th-6th*, *Preschool*.
- ***education-num***: variable númerica que representa a la anterior.
- ***marital-status***: variable categórica que expresa el estado civil de la persona. Los valores que puede tomar son:
  - *Married-civ-spouse*: la persona está casada con un civil.
  - *Divorced*: la persona está divorciada.
  - *Never-married*: la persona nunca ha estado casada.
  - *Separated*: la persona está separada.
  - *Widowed*: la persona es viuda.
  - *Married-spouse-absent*: la persona aparece como casada en el registro, pero no se encuentra a ninguna pareja [3].
  - *Married-AF-spouse*: la persona está casada con alguien de las fuerzas armadas.
- ***occupation***: variable categórica que describe el tipo de profesión que tiene la persona. Puede tomar los valores *Tech-support*, *Craft-repair*, *Other-service*, *Sales*, *Exec-managerial*, *Prof-specialty*, *Handlers-cleaners*, *Machine-op-inspct*, *Adm-clerical*, *Farming-fishing*, *Transport-moving*, *Priv-house-serv*, *Protective-serv*, *Armed-Forces*.
- ***relationship***: variable que contiene el valor que puede tomar la relación de una persona con respecto a otra dentro de una familia. Contiene solo un valor por instancia del dato. Estos valores pueden ser *Wife* (Esposa), *Own-child* (hijo propio), *Husband* (marido), *Not-in-family* (sin familia), *Other-relative* (otro tipo de familiar) y *Unmarried* (soltero).

- ***race***: valor que describe la raza de la persona. Puede ser *White* (Blanco), *Asian-Pac-Islander* (asíatico o de las islas del pacífico), *Amer-Indian-Eskimo* (indio americano o esquimal), *Other* (otro) y *Black* (negro).
- ***sex***: sexo de la persona, *Female* (mujer) o *Male* (hombre).
- ***capital-gain***: registro de la ganancia de capital de la persona.
- ***capital-loss***: registro de la pérdida de capital de la persona.
- ***hours-per-week***: corresponde a las horas de trabajo a la semana.
- ***country***: país de pertenencia. En este caso, hay una mayor variedad de valores, siendo los siguientes: *United-States*, *Cambodia*, *England*, *Puerto-Rico*, *Canada*, *Germany*, *Outlying-US(Guam-USVI-etc)*, *India*, *Japan*, *Greece*, *South*, *China*, *Cuba*, *Iran*, *Honduras*, *Philippines*, *Italy*, *Poland*, *Jamaica*, *Vietnam*, *Mexico*, *Portugal*, *Ireland*, *France*, *Dominican-Republic*, *Laos*, *Ecuador*, *Taiwan*, *Haiti*, *Columbia*, *Hungary*, *Guatemala*, *Nicaragua*, *Scotland*, *Thailand*, *Yugoslavia*, *El-Salvador*, *Trinadad&Tobago*, *Peru*, *Hong*, *Holand-Netherlands*, o lo que es lo mismo, Estados-Unidos, Colombia, Inglaterra, Puerto-Rico, Canada, Alemania, Estados supervisados de los Estados Unidos, India, Japón, Grecia, Sudáfrica, China, Cuba, Irán, Honduras, Filipinas, Italia, Polonia, Jamaica, Vietnam, México, Portugal, Irlanda, Francia, República Dominicana, Laos, Ecuador, Taiwn, Haití, Colombia, Hungría, Guatemala, Nicaragua, Escocia, Tailandia, Yugoslavia, El Salvador y Trinidad-Tobago.
- ***income***: corresponde al valor que queremos predecir, puede tomar los valores "<=50K" o ">50K"

1.2 Enfoque elegido

El enfoque que se usará en este trabajo consiste en realizar una serie de modelos ***M'*** usando distintos algoritmos. Cada uno de ellos tendrá un error de train y test diferentes, una complejidad distinta y comportamientos distintos a la hora de generalizar. Tras analizar cada uno de los distintos modelos, ver su complejidad, su idoneidad para el conjunto de datos, el error de test... se escogerá un modelo ***M*** que será el "mejor" del conjunto de modelos desarrollados. A lo largo del trabajo se desarrollarán distintos modelos usando los siguientes algoritmos:

- **Modelo lineal**: se usará como modelo lineal la *regresión logística*.
- **Modelo Random Forest**
- **Modelos de Base Radial**: en este caso, usaremos dos modelos de base radial diferentes:
  - **Support Vector Machine**
  - **KNN**
- **Red Neuornal**

2 Codificación de los datos de entrada para hacerlos útiles a los algoritmos

2.1 Valores perdidos

Respecto a los valores perdidos, podríamos tomar tres enfoques diferentes: sustituirlos por la media del resto de valores para ese dato predictor o predecirlo con un modelo, eliminar todas las filas que contengan algún valor perdido o no usar las variables que contengan valores perdidos. Para tomar esta decisión, primero hemos de estudiar el porcentaje de valores perdidos que presentan los datos y dónde se pierden estos datos.

Para ello, en primer lugar leemos los datos.

```
leer_datos <- function(fichero = "./Data/adult.data") {
    adult.train <- read.csv(fichero, header=FALSE, col.names = c("age","workclass",
            "fnlwgt","education","education-num","marital-status","occupation","relationship",
            "race","sex","capital-gain","capital-loss","hours-per-week","country","income"),
            na.strings = c(" ?", "?", ""), stringsAsFactors = F)
```

```
}

adult.train <- leer_datos()
adult.test <- leer_datos(fichero = "./Data/adult.test")
```

Para evitar problemas con las funciones de R, forzaremos a que todos los datos categóricos, tenga el mismo $factor$. Para ello, realizaremos lo siguiente:

```
# Añadimos una columna a los datos para indicar
# cuales pertenecen a datos de train y cuales a
# datos de test

adult.test$trainTest = rep(1,nrow(adult.test))
adult.train$trainTest = rep(0,nrow(adult.train))

# Reconstruimos el conjunto de datos al completo,
# uniendo los datos de train y test

fullSet <- rbind(adult.test,adult.train)

# Cada una de las variables categóricas, pasarán
# de ser cadenas de caracteres a tener un valor
# numérico o un factor, con lo que
# tanto datos de train como datos de test,
# obtendrán el mismo factor

fullSet$workclass = as.factor(fullSet$workclass)
fullSet$country = as.factor(fullSet$country)
fullSet$education = as.factor(fullSet$education)
fullSet$marital.status = as.factor(fullSet$marital.status)
fullSet$sex = as.factor(fullSet$sex)
fullSet$relationship = as.factor(fullSet$relationship)
fullSet$occupation = as.factor(fullSet$occupation)
fullSet$income = as.factor(fullSet$income)
fullSet$race = as.factor(fullSet$race)

# Reconstruimos los datos de train y test originales

adult.train = data.frame(fullSet[fullSet$trainTest == 0,])
adult.test = data.frame(fullSet[fullSet$trainTest == 1,])

# Y eliminamos la columna auxiliar

adult.test$trainTest = NULL
adult.train$trainTest = NULL
```

Una vez leídos, calculamos el número de variables perdidas en cada columna:

```
apply(X=adult.train, MARGIN=2, FUN=function(columna) length(is.na(columna)[is.na(columna)==T]))
```

```
##            age      workclass         fnlwgt      education  education.num
##              0           1836              0              0              0
## marital.status     occupation   relationship           race            sex
##              0           1843              0              0              0
##   capital.gain   capital.loss hours.per.week        country         income
##              0              0              0            583              0
```

Sólo tienen valores perdidos los atributos *workclass*, *occupation* y *country*.

También, vamos a comprobar el número de instancias que contienen algún dato perdido. Como ya hemos visto que los datos perdidos sólo se encuentran en las columnas *workclass*, *occupation* y *country*, nos fijaremos sólo en esas. Para ello, vamos a realizar lo siguiente:

```
getRowsNA <- function(datos = adult.train) {
    aux = is.na(datos)*1
    rowsMissingValues = apply(X=aux, MARGIN=1,
             FUN = function(fila) sum(fila))
}

rowsMissingValues.train = getRowsNA()
length(rowsMissingValues.train[rowsMissingValues.train > 0])
```

```
## [1] 2399
```

En total, sólo hay 2399 filas con valores perdidos. Al tener en total 32561 datos de entrenamiento, perder 2399 no supone una gran diferencia, sólo perdemos un $7,37\,\%$ de los datos. Por tanto, lo más sencillo es eliminar las filas que contengan algún dato perdido. Este mismo planteamiento es usado con los datos de test, donde tenemos 16282 filas en total, de las cuales 1222 presentan datos perdidos, un $7,505\,\%$ de los datos.

```
adult.train.clean = adult.train[rowsMissingValues.train == 0,]
adult.test.clean = adult.test[getRowsNA(datos = adult.test) == 0,]
fullSet.clean = data.frame(fullSet[getRowsNA(datos = fullSet) == 0,])
```

2.2 Asignación de valores numéricos a los valores categóricos

Al cargar el fichero de datos hemos usado la opción `stringsAsFactors`. Esta función sirve para convertir todas las cadenas de carácteres en factores. Los factores, tal y como se explica en [4], asignan a cada posible valor que puede tomar una variable categórica un número. Los factores representan una forma de almacenamiento muy eficiente, ya que las cadenas de carácteres correspondientes sólo se almacenan una vez y los valores se guardan como enteros.

Por ejemplo, los números asociados a la variable `workclass` corresponden con el orden mostrado por la función `levels`. Así, al primer elemento se le asigna el número uno, al siguiente el dos, etc.

```
levels(adult.train.clean$workclass)
```

```
## [1] "Federal-gov"      "Local-gov"        "Never-worked"
## [4] "Private"          "Self-emp-inc"     "Self-emp-not-inc"
## [7] "State-gov"        "Without-pay"
```

Para ver el vector numérico almacenado por `R` en vez de el vector con las cadenas de caracteres usamos la función `c`:

```
head(c(adult.train.clean$workclass))
```

```
## [1] 7 6 4 4 4 4
```

```
head(adult.train.clean$workclass)
```

```
## [1] State-gov        Self-emp-not-inc Private          Private
## [5] Private          Private
## 8 Levels: Federal-gov Local-gov Never-worked Private ... Without-pay
```

Como vemos, los números asignados a los primeros elementos coinciden con el orden mostrado anteriormente.

2.3 Eliminando la variable `education.num`

Aunque cada problema tenga su propio método para escoger y normalizar variables, hemos considerado que la variable `education.num` no nos va a servir, ya que es únicamente una asignación numérica de los valores categóricos de la variable `education`. Al haber almacenado la variable `education` como factor, ya tiene una asignación numérica hecha, y por tanto, eliminamos la variable `education.num`:

```
adult.train.clean[,"education.num"] = NULL
adult.test.clean[,"education.num"] = NULL
adult.train[,"education.num"] = NULL
adult.test[,"education.num"] = NULL
```

3 Ajustando un modelo lineal

3.1 Idoneidad del modelo lineal

Antes de pasar a modelos más complejos, vamos a comprobar cómo se comporta un modelo lineal. Esto es así, ya que la complejidad de un modelo lineal es bastante menor que a la de los otros modelos que se usarán a continuación, y si es capaz de ajustar bastante mejor los datos que el resto de modelos, se escogería este.

Para ello, nos vamos a basar en un modelo que utiliza la regresión logística para este problema. Este modelo, nos devuelve la probabilidad de pertenecer a una clase u otra. Usando el concepto de *maximum likelihood* o máxima probabilidad, asigna a los datos la clase a la que es más probable que pertenezca, de acuerdo a la probabilidad que calcula el modelo.

Estas probabilidades se obtienen a partir de la siguiente expresión:

$$P(y|x) = \theta(yw^T x)$$

donde θ es la función logística, y es la etiqueta del modelo, w es nuestro vector de pesos asociado, y x los datos predictores. Esta función es la que usaremos para crear un modelo de estimación del error, y trataremos de minimizar ese error, que lo calcularemos de la siguiente forma [5]:

$$E_{in} = \frac{1}{N}\sum_{n=1}^{N} \ln\left(\frac{1}{\theta(y_n w^T x_n)}\right) = \frac{1}{N}\sum_{n=1}^{N} \ln\left(1 + e^{-y_n w^T x_n}\right)$$

3.2 Dimensión del modelo lineal

La dimensión del modelo lineal la mediremos en función de la dimensión de **Vapnik–Chervonenkis** o dimensión VC, denotada como d_{VC} que expresa la complejidad del modelo, y el máximo valor de N, donde N es el número de datos disponibles, que el modelo puede separar. En el caso de los modelos lineales, como es el caso, $d_{VC} = d + 1$, donde d es el número de variables predictoras que tenemos.

Con esto vemos, que la dimensión d_{VC} es basntante competitiva y puede comportarse bien en el problema.

3.3 Parámetros del modelo

En el modelo de regresión logística, el principal parámetro a configurar es el vector de pesos w que utiliza la regresión. Este vector de pesos se puede inicializar a un valor, fijo o un valor aleatorio, que de algún modo marca el punto de inicio del algoritmo de regresión. Este parámetro se irá actualizando en la regresión logística conforme se vayan comprobando nuevos puntos.

```
predictorGLM <- function(model, pintar = T){
  ypred = predict(model, adult.test.clean, type="response")
  ypred[ypred <= 0.5] = ">50K"
  ypred[ypred > 0.5] = "<=50K"
```

```
  if(pintar){
    print("Matriz de confusión datos de test")
    print(table(predict=ypred, truth=(adult.test.clean$income)))
  }
  cat("Eout = ",mean((ypred != adult.test.clean$income)*1))
  ypred
}
trainingIndex=which(fullSet.clean$trainTest==0)
fullSet.clean$trainTest = NULL
fullSet.clean$education.num = NULL
set.seed(1)
glmModel = glm(income ~ ., data = fullSet.clean,
    subset = trainingIndex, family = binomial(logit))
glmPred = predictorGLM(glmModel, pintar=T)
```

```
## [1] "Matriz de confusión datos de test"
##        truth
## predict <=50K  >50K
##   <=50K 11360  3700
## Eout =  0.2456839
```

Como podemos ver, el error que genera la regresión lineal, tomando todas las variables como predictoras, genera un error de aproximadamente el 24 %. Como vemos, el error generado por la regresión logística es bastante alto y por ello, vamos a comprobar otros modelos. El principal problema de la regresión logística, es que se ve muy afectado por las clases desbalanceadas como es el caso, donde la clase perteneciente al conjunto `>50K` tiene muchas menos instancias que la clase `<=50K` por lo que el modelo puede ajustar muy bien la clase mayoritaria, mientras que la clase minoritaria prácticamente la obvia.

```
cat("Error obtenido en la clase <=50K: ",
    0/length(adult.test.clean$income[adult.test.clean$income=="<=50K"]))
```

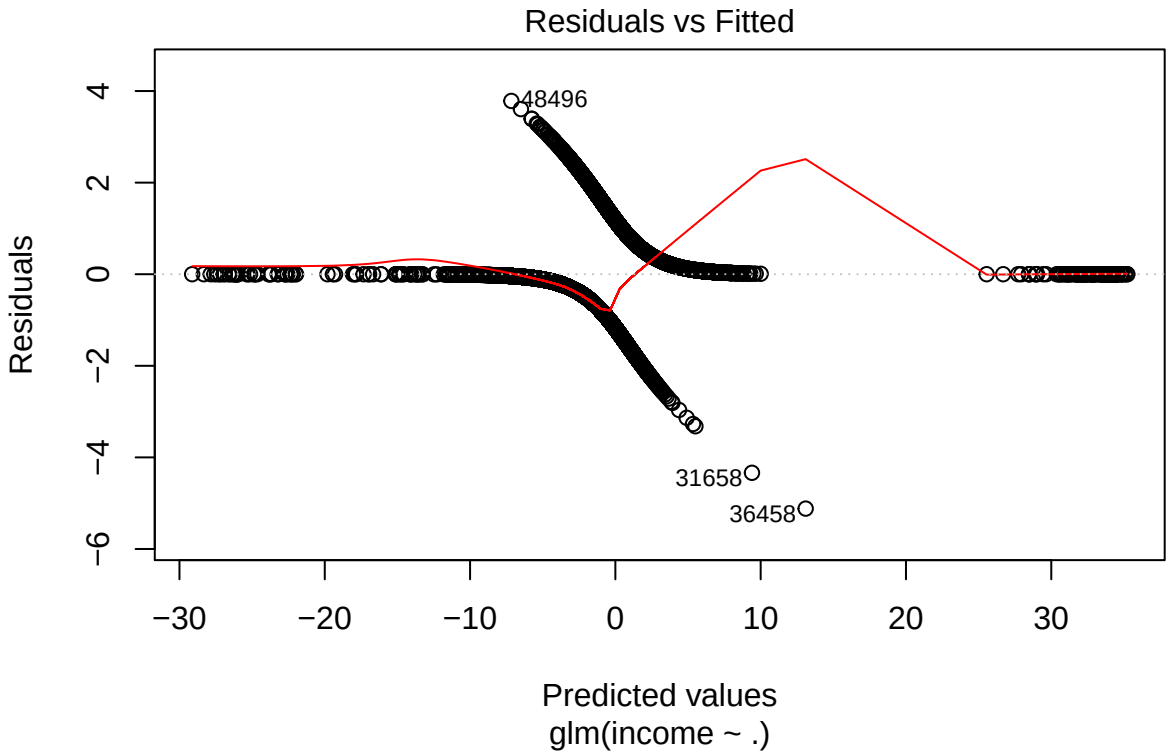
```
## Error obtenido en la clase <=50K:  0
```

```
cat("Error obtenido en la clase >50K: ",
    370000/length(adult.test.clean$income[adult.test.clean$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  100
```

Como vemos, el modelo no ha fallado ni un solo caso de la clase `<=50K`, pero ha fallado todos los casos `>50K`. Esto se debe a que todas las instancias de test han sido clasificadas como `<=50K`.

```
plot(glmModel, which=c(1))
```



En esta gráfica vemos cómo el error del modelo sigue una tendencia no lineal, esto nos indica que debemos ajustar un modelo más complejo para poder ajustar bien los datos.

4 Ajustando un modelo *Random Forest*

4.1 Idoneidad del modelo *Random Forest* para los datos del problema

Un modelo usando un árbol como base, es capaz de clasificar los datos con cierta precisión y tiene la ventaja de que por sí solo, es capaz de explicar bastante bien la clasificación que ha hecho. Pero, utilizar un sólo árbol no ofrece la potencia necesaria para ajustarse lo suficientemente bien a los datos, además de que sufre de tener una alta variabilidad, ya que si particionamos los datos aleatoriamente en subconjuntos, podemos obtener resultados muy diferentes si aprendemos el modelo con un subconjunto u otro, y por ello usaremos el modelo *Random Forest*, que aunque pierda las facilidades de comprensión que tiene un árbol, nos ofrece un poder de ajuste muchísimo mayor, y un poder de generalización mayor.

El porqué de usar *Random Forest* frente a un modelo *Bagging* es que con este último, tenemos que aprender el modelo usando las p variables predictoras, mientras que con *Random Forest* podemos usar un subconjunto de tamaño m. *Bagging*, al hacer uso de los p predictores, en el caso de que haya una variable predictora muy fuerte en los datos, todos los árboles que se acaben generando, tendrán esta variable predictora en los primeros nodos del árbol, mientras que al usar varios subconjuntos de los p predictores de tamaño m escogidos de forma aleatoria en *Random Forest*, seremos capaces de detectar otras relaciones más débiles entre las variables predictoras pero que también son capaces de predecir los datos con exactitud.

Respecto a los datos que tenemos (30162 datos de entrenamiento y 15060 datos de test)[1] tenemos suficientes datos como para que *Random Forest* trabaje de forma adecuada, y disponemos de 14 datos predictores, ya que la variable `income` es aquella que queremos predecir.

[1] Eliminando las instancias con valores perdidos.

4.2 Parámetros del modelo

El parámetro que utiliza el modelo *Random Forest*, es el parámetro m, que indica el número de variables que se escogeran de forma aleatoria para crear un modelo de árbol para ajustar los datos cada vez que el modelo vaya a ajustar un nuevo árbol. Al no ser un modelo paramétrico, no tiene mucho más modificar, ya que los pesos no se pueden establecer desde un principio.

4.3 Hiperparámetros del modelo

En este caso, como hiperparámetro se utiliza el **número de árboles** que generará el modelo para ajustar los datos. En cierto modo, es similar a la regularización, ya que a mayor número de árboles, más sobreajustaremos el modelo, mientras que con un menor número de árboles, menos se sobreajusta el modelo.

4.4 Complejidad del modelo

La dimensión d_{VC} para un modelo de árbol, es infinita, ya que para un conjunto de datos, podemos añadir al árbol tantas variables como queramos para clasificar de forma correcta todos los datos, y si tenemos infinitos datos, podemos añadir infinitas variables al árbol para separar hasta el último punto del espacio, con lo que $d_{VC} = \infty$.

Si esto lo aplicamos a *Random Forest* tenemos que la dimensión $d_{VC} = \infty$ para este modelo también, al estar construido sobre árboles.

4.5 Aplicación del modelo *Random Forest*

Para ajustar un modelo *Random Forest*, haremos uso del paquete *randomForest* de *R*.

En primer lugar, vamos a ajustar un modelo *Random Forest* con el número de predictores por defecto de `R`: $m = \sqrt{p}$. Siendo p el número de variables predictoras totales. Este número por defecto es el recomendado para problemas de clasificación como éste. En este caso, al tener $p = 13$, el número de predictores a usar será $m = 3$.

```
library(randomForest)

outOfSampleError <- function(model, newdata = adult.test.clean,  printTable = T, ...){
  set.seed(1)
  ypred = predict(object = model, newdata = newdata, ...)
  if(printTable)
    print(table(predict=ypred, truth=newdata$income))
  cat("Eout = ",mean((ypred != newdata$income)*1))
}
set.seed(1)
rf.clean = randomForest(income ~ ., data = adult.train.clean, importance = T)
print(rf.clean)
```

```
##
## Call:
##  randomForest(formula = income ~ ., data = adult.train.clean,      importance = T)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 18.17%
## Confusion matrix:
##       <=50K >50K class.error
## <=50K 22600   54 0.002383685
```

```
## >50K   5426 2082 0.722695791
```

```
outOfSampleError(rf.clean)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 11326  2706
##   >50K     34   994
## Eout =  0.1819389
```

Con $m = 3$ obtenemos un modelo con un $18,18\,\%$ de error *Out Of Bag*. Al ajustar el modelo de Random Forest, cada uno de los árboles usa un subconjunto de los datos, y los no usados para calcular dicho árbol se quedan como *Out Of Bag*. Una vez generados los árboles, se predice el valor de cada dato usando los árboles en los que dicho dato ha sido *Out Of Bag*. En el caso de clasificación se hace mediante voto mayoritario. El error de clasificación obtenido para cada dato es válido, porque para predecir el dato se han usado árboles en los que el dato en cuestión no "ha participado" [6].

Respecto a la matriz de confusión generada, vemos que hay muchos más aciertos para la clase `<=50K` que para la clase `>50K` y que la tasa de error en la primera clase es mucho menor que la de la segunda, $0,002$ frente a $0,72$. Esto puede deberse a que el número de instancias para el la clase `<=50K` sea mucho mayor y, por tanto, el modelo tienda a clasificar los datos como `<=50K`:

```
length(adult.train.clean$income[adult.train.clean$income == "<=50K"])
```

```
## [1] 22654
```

```
length(adult.train.clean$income[adult.train.clean$income == ">50K"])
```

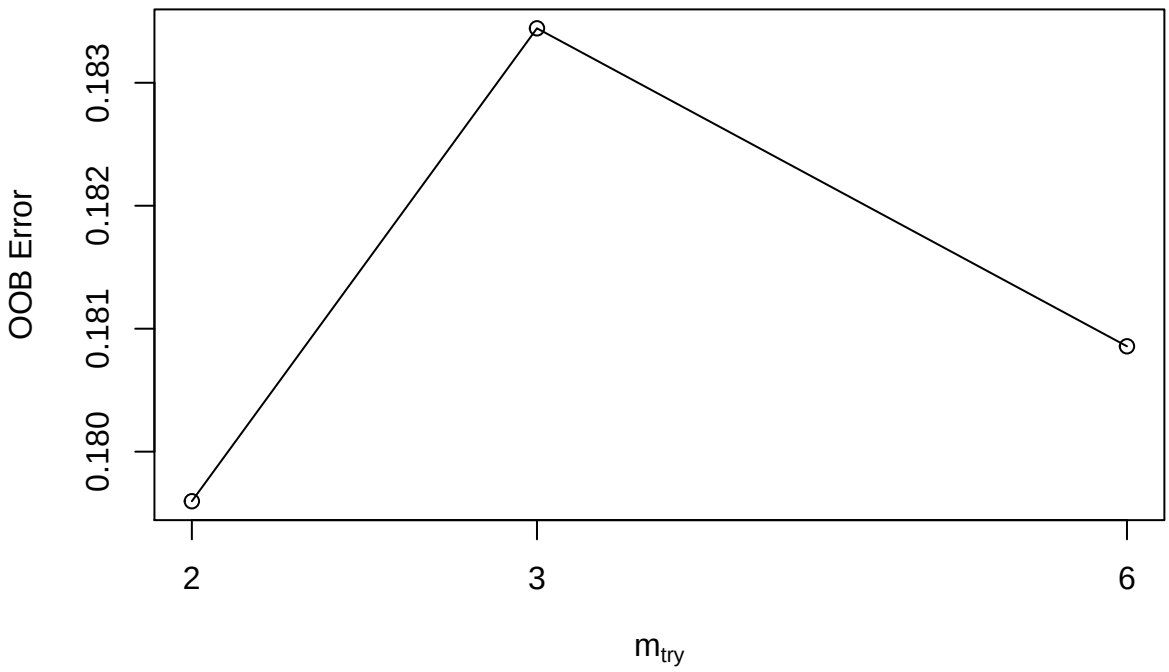
```
## [1] 7508
```

Como hemos dicho antes, hay una gran diferencia entre la proporción de datos para un clase y otra, esto hace que el modelo tienda a clasificar los datos como `<=50K`. De hecho, si nos fijamos en la matriz de confusión, los fallos de clasificación con `<=50K` son mucho mayores a los de `>50K`: 5446 frente a 54.

Ahora bien, ¿podemos llegar a obtener un error más bajo cambiando éste parámetro? Para ello, usamos la función `tuneRF`, que empieza en el valor por defecto de `R` y, a partir de éste, prueba con otros valores hasta encontrar el óptimo. En cada iteración, incrementa o decrementa el valor de m según un parámetro `stepFactor` y además, si la mejora de error obtenida no es mayor a lo indicado por el parámetro `improve`, se para la búsqueda.

```
set.seed(1)
rf.tuned = tuneRF(x=subset(adult.train.clean, select=-income),
      y=adult.train.clean$income, doBest=T)
```

```
## mtry = 3  OOB error = 18.34%
## Searching left ...
## mtry = 2     OOB error = 17.96%
## 0.02096512 0.05
## Searching right ...
## mtry = 6     OOB error = 18.09%
## 0.01409723 0.05
```



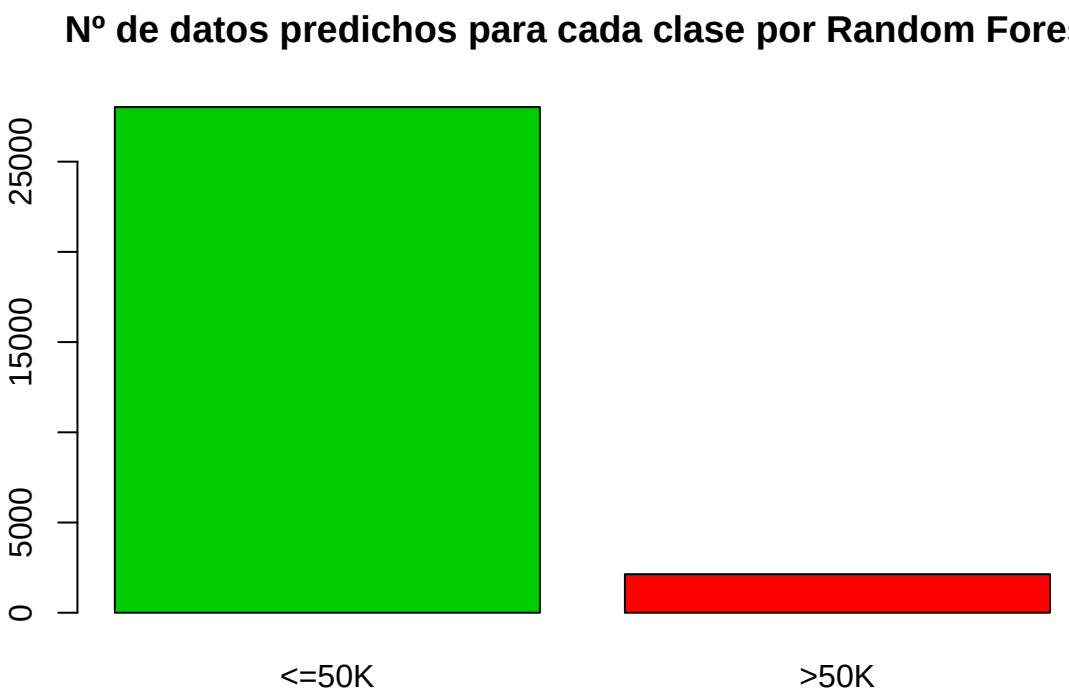
Como vemos en la gráfica, con $m = 2$ obtenemos el modelo óptimo con menor error, por tanto, en vez de quedarnos con el modelo generado anteriormente con $m = 3$, nos quedamos con el mejor obtenido según el error *Out-Of-Bag*, junto con la matriz de confusión para los datos de train, donde vemos que el error para la clase `>50K` es del 72 %.

Lo último que podemos preguntarnos es, ¿cómo se comporta el modelo con los datos de test? Para saberlo, generamos la tabla de confusión:

```
outOfSampleError(rf.tuned)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 11333  2701
##   >50K     27   999
## Eout =  0.1811421
```

```
plot(rf.clean$predicted, col = c(3,2), main = "Nº de datos predichos para cada clase por Random Forest")
```




```
cat("Error obtenido en la clase <=50K: ",
    2700/length(adult.test.clean$income[adult.test.clean$income=="<=50K"]))
```

```
## Error obtenido en la clase <=50K:  0.2376761
```

```
cat("Error obtenido en la clase >50K: ",
    269900/length(adult.test.clean$income[adult.test.clean$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  72.94595
```

El E_{out} obtenido es muy parecido al *Out-Of-Bag* aunque algo mayor (17,9 en train frente a 18,1 en test), ya que éste es una buen estimador del error fuera de la muestra tal y como hemos compobado.

En esta matriz de confusión se aprecia lo desbalanceadas que están las clases en este problema: en la clase `<=50K` hemos obtenido un error del $0,2\,\%$, mientras que en la clase `>50K` el error ha sido del $72\,\%$.

En los datos de test también se aprecia lo desbalanceadas que están las clases en el problema y esto influye también en los resultados obtenidos:

```
length(adult.test.clean$income[adult.test.clean$income == "<=50K"])
```

```
## [1] 11360
```

```
length(adult.test.clean$income[adult.test.clean$income == ">50K"])
```

```
## [1] 3700
```

Al haber esta diferencia en los datos de entrenamiento y test, el modelo se ve muy afectado.

5 Ajustando un modelo de base radial

5.1 Support Vector Machine

5.1.1 Idoneidad del modelo *Support Vector Machine* para los datos del problema

El uso de los modelos de *Support Vector Machine* o *SVM* es muy apropiado para el problema, ya que funciona bastante bien con los problemas de clasificación binaria, como es el caso, ya que ofrece una aproximación muy

natural al problema, y un buen comportamiento en problemas con grandes dimensiones, gracias al uso del *Kernel*.

Además, los modelos SVM, tienen un rendimiento muy bueno frente a conjuntos de datos con clases desbalanceadas, como es nuestro caso, donde la clase con un valor de $income$ = "$<= 50K$" representa un subconjunto mucho mayor que la clase con $income$ = "$> 50K$""$, como podemos ver en [7].

5.1.2 Parámetros del modelo

Los parámetros a ajustar del modelo son:

- *Kernel del SVM*: será el kernel que vamos a usar, es decir, cómo de importantes serán el resto de instancias de los datos para ajustar un dato, en función de la distancia entre los datos. En este caso, se usará un modelo de con un kernel de base radial, que tiene como base la siguiente función:

$$w(u, v) = e^{-\gamma \cdot |u-v|^2}$$

- γ: valor que tomará γ para la función de base radial. Normalmente, se suele hacer que $\gamma = \frac{1}{d}$, donde d es la dimensión del problema.
- ϵ: valor de ϵ para la función de pérdida de SVM. En este caso valdrá 0.1.
- *Ponderización de la clase*: en caso de que sea necesario, podemos ponderar las clases si queremos para balancear el conjunto de datos, dando más peso a la clase minoritaria que a la mayoritaria. En un principio, las clases tendrán el mismo peso, es decir 1.

5.1.3 Hiperparámetros del modelo

En este caso, como hiperparámetro del modelo tenemos el parámetro λ o *cost*, que controla la regularización del problema. Esta regularización de los datos, se realiza haciendo uso de la formulación de Lagrange.

5.1.4 Complejidad del modelo

La complejidad de los modelos SVM varía en función del $Kernel$ escogido para la función. En el caso de los modelos de base radial, como es el caso, esta dimensión d_{VC}, ya que si nuestro espacio de búsqueda se establece en $\mathbf{R}^d$, dado un punto x_1, podemos coger un punto $x_2 \in R^d$ y medir la distancia entre estos dos puntos para ver lo que aporta el modelo. Como esta distancia puede ser infinita, $d_{VC} = \infty$.[8]

5.1.5 Ajuste del SVM al problema

5.1.6 Normalización de los datos

Para ver como se comporta *Support Vector Machine* al problema, vamos

```
library(e1071)
set.seed(1)
svmModel = svm(income ~ ., data = adult.train.clean, kernel = "radial")
outOfSampleError(svmModel)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 10637  1519
##   >50K    723  2181
## Eout =  0.1488712
```

Como vemos, el error fuera de la muestra supone un 15 % de E_{out} aproximadamente con un kernel de base radial, un valor de $\gamma = \frac{1}{14}$, $\epsilon = 0,1$, con el mismo peso para las dos clases, y con el valor $cost = 1$. En la matriz de confusión, vemos como este modelo se comporta mejor que otros modelos frente a la clase minoritaria, reduciendo mucho la tasa de error en esta.

A continuación, vamos a realizar tres nuevos modelos con SVM, añadiendo regularziación al modelo. Este valor de regularización tomará los valores de $\lambda = 0,001$, $\lambda = 0,01$, $\lambda = 0,1$ y $\lambda = 0,9$.

```
set.seed(1)
svmModelReg_0001 = svm(income ~ ., data = adult.train.clean, kernel = "radial", cost = 0.001)
outOfSampleError(svmModelReg_0001)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 11360  3700
##   >50K      0     0
## Eout =  0.2456839
```

```
set.seed(1)
svmModelReg_001 = svm(income ~ ., data = adult.train.clean, kernel = "radial", cost = 0.01)
outOfSampleError(svmModelReg_001)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 11357  3535
##   >50K      3   165
## Eout =  0.234927
```

```
set.seed(1)
svmModelReg_01 = svm(income ~ ., data = adult.train.clean, kernel = "radial", cost = 0.1)
outOfSampleError(svmModelReg_01)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 10918  1975
##   >50K    442  1725
## Eout =  0.1604914
```

```
set.seed(1)
svmModelReg_09 = svm(income ~ ., data = adult.train.clean, kernel = "radial", cost = 0.9)
outOfSampleError(svmModelReg_09)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 10646  1513
##   >50K    714  2187
## Eout =  0.1478752
```

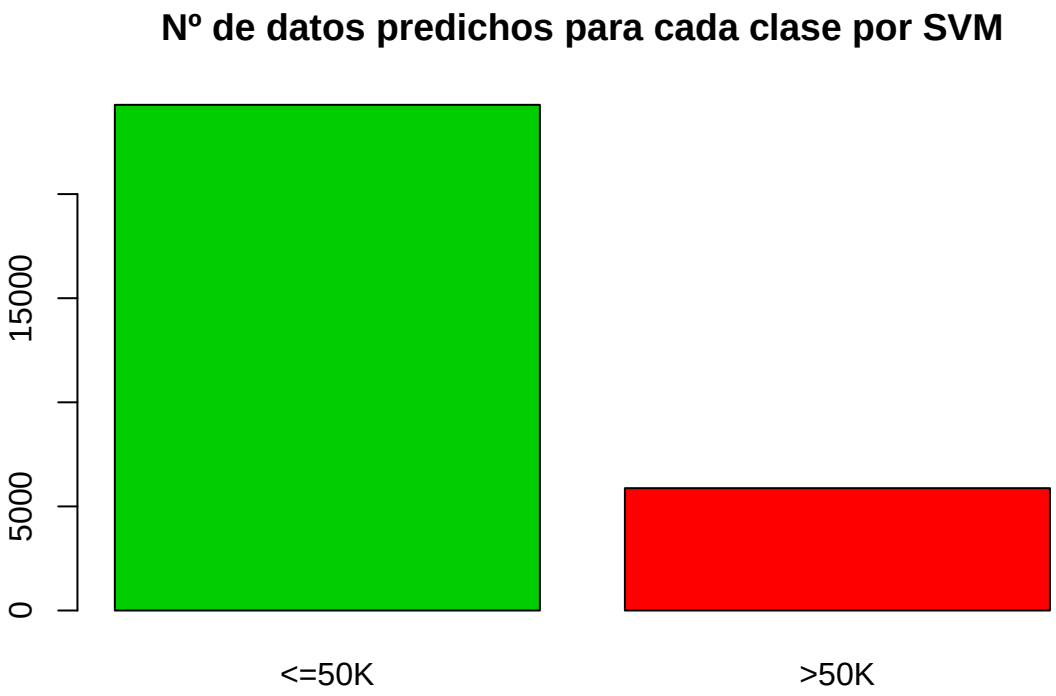
```
set.seed(1)
svmModelReg_25 = svm(income ~ ., data = adult.train.clean, kernel = "radial", cost = 2.5)
outOfSampleError(svmModelReg_25)
```

```
##         truth
## predict <=50K  >50K
##   <=50K 10623  1499
##   >50K    737  2201
## Eout =  0.1484728
```

Como podemos ver, los resultados cambian dependiendo del valor de λ que tome el modelo. Frente al valor de λ del primer modelo SVM, que era 1, el resto de modelos empeoran al cambiar el valor de la regularización, pasando de una gran deterioro del modelo en valores pequeños de λ ($0,001$ y $0,01$) a un deterioro más pequeño con $\lambda = 0,1$. En todos estos casos, estábamos sobreaustando el modelo. Sin embargo, con $\lambda = 0,9$ el modelo es capaz de mejorar un poco al primer modelo y un con un λ mayor, podíamos llegar a subajustar el modelo.

También podemos ver como si hacemos el valor de λ mayor, el modelo no realiza mejoras significativas, y en este caso, incluso empeora un poco.

```
plot(svmModelReg_09$fitted, col = c(3,2), main = "Nº de datos predichos para cada clase por SVM")
```




```
cat("Error obtenido en la clase <=50K: ",
    71400/length(adult.test.clean$income[adult.test.clean$income=="<=50K"]))
```

```
## Error obtenido en la clase <=50K:  6.285211
```

```
cat("Error obtenido en la clase >50K: ",
    151300/length(adult.test.clean$income[adult.test.clean$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  40.89189
```

5.2 Vecino más cercano con base radial

5.2.1 Idoneidad del modelo de *Vecino más cercano* para los datos del problema

La razón principal para usar el vecino más cercano es que tenemos una gran cantidad de datos y además, tenemos una dimensión $d = 15$. Por tanto, tenemos instancias suficientes como para que el modelo del vecino más cercano actúe de forma correcta.

Otro punto a favor para el modelo del *Vecino más cercano* es que funciona bien con clases desbalanceadas, tal y como se indica en [7].

Además, al usar una aproximación con base radial, obtenemos una clasificación mucho mejor dando a cada vecino un peso dependiende de su *distancia de Minkowski* [9], que es una generalización de la *distancia Euclídea* y la *distancia de Manhattan*:

$$d(x_i, x_j) = \left(\sum_{s=1}^{p} |x_{is} - x_{js}|^q \right)^{\frac{1}{q}}$$

Hay varios kernel distintos para actualizar los pesos, como por ejemplo una *gaussiana* o la inversa de la distancia:

$$Gauss\ kernel = \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{d^2}{2}\right) \qquad Inversion\ kernel = \frac{1}{|d|}$$

5.2.2 Dimensión VC del modelo del Vecino más cercano

Al igual que *Random Forest*, la d_{VC} de éste modelo es $d_{VC} = \infty$. Esto se cumple tanto para la técnica $1-NN$ como para la técnica $k-NN$ y se debe a que el conjunto de hipótesis $\mathcal{H}$ que genera el modelo ajusta con $E_{in} = 0$ cualquier dataset, sin importar el tamaño tal y como se indica en [5].

5.2.3 Normalización de los datos

En primer lugar, vamos a normalizar las variables numéricas de nuestro modelo: `age`, `fnlwgt`, `capital.gain`, `capital.loss` y `hours.per.week`.

Para ello, he desarrollado una función que normaliza los datos en el intervalo $[0, 1]$ aplicando la siguiente fórmula a cada uno de los datos:

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

donde x representa el vector de datos originales y z el vector de datos normalizados. Este método busca normalizar en base al mínimo (que representaría el 0) y el máximo (que representaría el 1) todos los datos. En `R`, la forma de implementar este método es mediante la función `scale`:

```
normalizar_maxmin <- function(data=adult.train.clean, numbercols = c("age","fnlwgt","capital.gain",
                            "capital.loss","hours.per.week"), data_test = adult.test.clean) {
    # nos quedamos sólo con las columnas numéricas
    cols_numericas = subset(data, select=numbercols)
    colstest_numericas = subset(data_test, select=numbercols)
    # calculamos el máximo y el mínimo de cada columna de los datos de train
    maxs = apply(X=cols_numericas, MARGIN=2, FUN=max)
    mins = apply(X=cols_numericas, MARGIN=2, FUN=min)
    # aplicamos el escalado a los datos de train
    datos_normalizados_numericos = scale(x=cols_numericas, center=mins, scale=maxs)
    # aplicamos los valores de normalización de train sobre los de test
    test_normalizado = as.data.frame(scale(x = colstest_numericas,
                    center = attr(datos_normalizados_numericos, "scaled:center"),
                    scale = attr(datos_normalizados_numericos, "scaled:scale")))
    # juntamos los valores normalizados con el resto de columnas
    datos_normalizados_numericos = as.data.frame(datos_normalizados_numericos)
    for (c in numbercols) {
        data[c] = datos_normalizados_numericos[c]
        data_test[c] = test_normalizado[c]
    }
    list(data, data_test)
}

norm = normalizar_maxmin()
adult.train.clean.norm = norm[[1]]
adult.test.clean.norm = norm[[2]]
norm = NULL
```

5.2.4 Aplicación del modelo de *Vecino más cercano con base radial*

Para elegir el k a usar, vamos a entrenar el modelo con *Validación cruzada leave-one-out*. Como `kmax` vamos a poner el doble de la dimensión del dataset, para que la función tenga flexibilidad en cuanto al número de vecinos entre los que escoger. Como kernel, vamos a elegir entre uno gaussiano y otro inversamente proporcional a la distancia, ya que elegir entre alguno más incrementa muchísimo el tiempo de ejecución.

```
library(kknn)
set.seed(1)
best_model_knn <- train.kknn(formula = income ~ ., data = adult.train.clean.norm,
  kmax = 2*ncol(adult.train), kernel = c("gaussian", "inversion"))
best_model_knn
```

```
##
## Call:
## train.kknn(formula = income ~ ., data = adult.train.clean.norm,     kmax = 2 * ncol(adult.train), ke
##
## Type of response variable: nominal
## Minimal misclassification: 0.1633512
## Best kernel: gaussian
## Best k: 23
```

El error obtenido con este modelo ha sido del 16 %. A pesar de que este error haya sido con los datos de entrenamiento, es una buena medida debido a que ha sido calculado mediante validación cruzada *leave-one-out* y la estimación se ha hecho en base a ese dato que se ha dejado fuera y no ha participado en el modelo. Como esto se ha repetido una vez por cada dato que tenemos, obtenemos una buena estimación del error del modelo.

Una vez obtenidos los mejores parámetros, vamos a ajustar el modelo para obtener también un vector con predicciones y poder hacer la matriz de confusión:

```
set.seed(1)
model_knn <- kknn(formula = income ~ ., train = adult.train.clean.norm,
    test = adult.test.clean.norm, k = best_model_knn$best.parameters$k,
    kernel = best_model_knn$best.parameters$kernel)
print(table(predict = model_knn$fitted.values, truth = adult.test.clean$income))
```

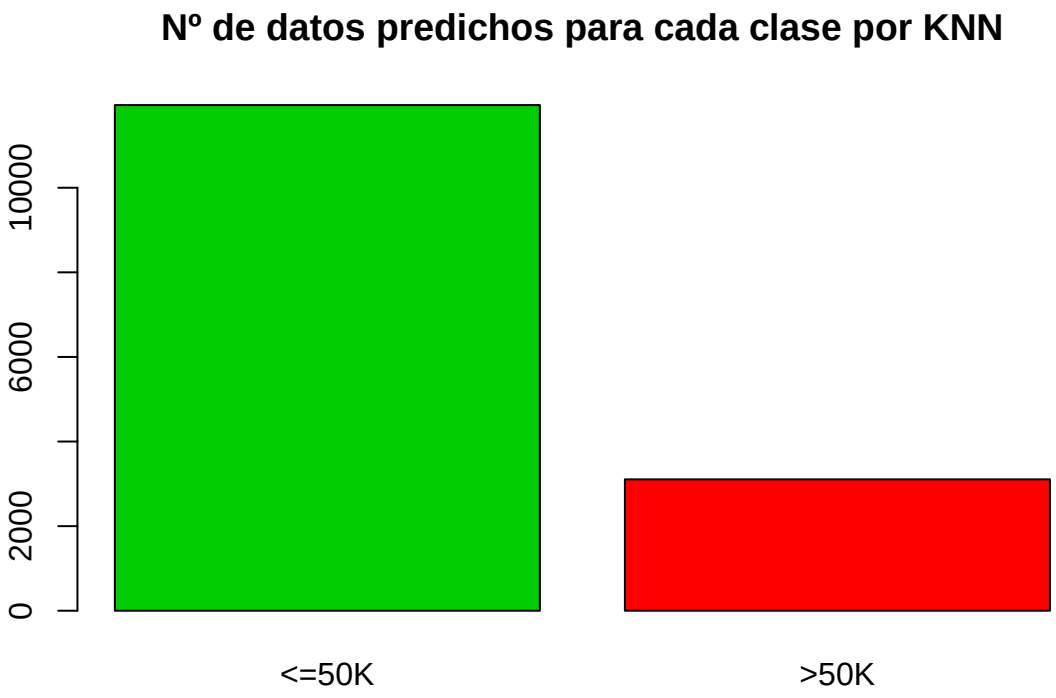
```
##        truth
## predict <=50K  >50K
##   <=50K 10413  1541
##   >50K    947  2159
```

```
cat("Eout = ",mean((model_knn$fitted.values != adult.test.clean$income)*1))
```

```
## Eout =  0.1652058
```

El error de test obtenido ha sido del 16 %, al igual que el error de validación cruzada. En este caso el modelo ha conseguido corregir las clases desbalanceadas que había, equilibrando el error obtenido en una y en otra:

```
plot(model_knn$fitted.values, col = c(3,2),
     main = "Nº de datos predichos para cada clase por KNN")
```




```
cat("Error obtenido en la clase <=50K: ",
    94700/length(adult.test.clean$income[adult.test.clean$income=="<=50K"]))
```

```
## Error obtenido en la clase <=50K:  8.336268
```

```
cat("Error obtenido en la clase >50K: ",
    154100/length(adult.test.clean$income[adult.test.clean$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  41.64865
```

A pesar de que el error en la clase `>50K` sigue siendo bastante alto, ha disminuido bastante y el error en la clase `<=50K` ha aumentado pero aún sigue siendo bajo. El error en ambas clases se ha equilibrado un poco y por eso, hemos obtenido un mejor error en el modelo que con `RandomForest`.

6 Ajustando un modelo de *Red Neuronal*

6.1 Idoneidad del modelo de *Red Neuronal* para los datos del problema

La *Red Neuronal* es un modelo muy extendido y con mucha flexibilidad en cuanto a la parametrización del modelo. Es tal la flexibilidad que ofrece, que tiene una gran potencia para ajustar los datos y capacidad para crear un modelo muy competitivo, pero tienen un alto coste a pagar, y es que el sobreajuste a los datos está practicamente servido de entrada. Este sobreajuste se puede mejorar usando regularización con *early stopping* o *weight decay*.

Uno de los parámetros que utiliza la *Red Neuronal* es el número de *unidades ocultas*, m. Este parámetro controla el número de capas internas de las que dispone nuestra red. Éste parámetro afecta mucho a su comportamiento, ya que tal y como se indica en [5], con éste parámetro suficientemente alto podemos asegurar $E_{in} \approx E_{out}$. A mayor número de capas, las transformaciones que existen entre capa y capa, dan potencia y flexibilidad al modelo para generar una función g que ajuste a f, con la contrapartida de que aumentamos la posibilidad de sobreajustar los datos. Además, como también tenemos un número alto de instancias, N, incrementamos la posibilidad de obtener un buen rendimiento en el modelo obtenido. La mejor forma de escoger este parámetro es mediante validación cruzada.

Al tener un problema de dimensión media/alta ($d = 15$) y tantas instancias ($N = 30162$), conviene usar un modelo lo suficientemente flexible para poder obtener una buena aproximación de forma sencilla.

Por último, al igual que otros modelos usados en este proyecto, las *Redes Neuronales* son buenos modelos para clases desbalanceadas, tal y como se indica en [7].

6.1.1 Dimensión VC del modelo

La dimensión d_{VC} para una red neuronal sigmoidal, generalmente, puede ser infinita. Pero, para el sigmoide $tanh(\cdot)$, donde el nodo salida es el $sign(\cdot)$, se puede ver que

$$d_{VC} = O(VQ)$$

donde V es el número de capas ocultas y Q el número de pesos.

6.2 Normalización de los datos

Para normalizar los datos, vamos a usar el mismo resultado usado en el modelo del *Vecino más cercano*.

6.3 Parámetros e Hiperparámetros del modelo

Para elegir los parámetros, vamos a seguir los consejos dados en [10]:

- ***Pesos iniciales***: los pesos iniciales serán números aleatorios cercanos a cero. Así, el modelo empezará de forma lineal pero se transformará en uno no lineal conforme los pesos aumenten. Si se empezase con pesos exactamente cero, el algoritmo no sería capaz de moverse, ya que estaríamos haciendo derivadas por cero. Empezar con pesos muy grandes suele llevar a soluciones malas.
- ***Regularización***: al tener tantos pesos, las redes neuronales pueden sobreajustar los datos en un mínimo local. Para evitar esto, como hemos dicho antes, podemos usar *early stopping* o *weight decay*. En este caso vamos a usar *weight decay*, ya que la función que vamos a utilizar, `nnet`, nos permite ajustar este hiperparámetro de forma sencilla. Debido a la gran cantidad de instancias de las que disponemos, la función `tune.nnet` necesitaba un gran tiempo de cálculo y, en vez de realizar validación cruzada, hemos usado $\lambda = 0,1$ para aplicar regularización, ya que de entre todos los parámetros que hemos probado era el que mejor resultado obtenía.
- ***Número de capas ocultas***: es recomendable tener entre 5 y 100 capas, para que el modelo pueda tener flexibilidad suficiente. La regularización se encarga de dejar a 0 aquellos pesos que lleven a sobreajusar el modelo. El problema es que, debido a que el paquete sólo permite usar una única capa oculta, no vamos a poder configurar este parámetro y obtendremos unos resultados peores que si usáramos un mayor número de capas ocultas. Hay otros paquetes que sí permiten configurar el número de capas ocultas, pero no permiten usar datos categóricos.
- ***Número de unidades en la capa oculta***: lo que sí podemos configurar es el número de unidades en la capa oculta, aunque de forma limitada: si establecemos un número mayor a 15, la función devuelve un error diciendo que el modelo tiene demasiadas unidades.

6.4 Aplicación del modelo

Hemos establecido a 10000 el número máximo de iteraciones, para que el algoritmo pudiese converger antes de llegar a dicho límite. Esto no es un problema ya que estamos aplicando regularización con *weight decay*. El parámetro `trace = F` sólo sirve para que no se imprima cada paso que da el algoritmo en el PDF.

```
library(nnet)
set.seed(1)
model.nnet = nnet(formula = income ~ ., maxit=10000,
    data = adult.train.clean.norm, size = 10, decay=0.1,trace=F)
outOfSampleError(model.nnet, adult.test.clean.norm, type="class")
```

```
##         truth
## predict <=50K  >50K
##   <=50K 10388  1391
##   >50K    972  2309
## Eout =  0.1569057
```

A pesar de disponer de una única capa oculta, el modelo ha sido capaz de obtener un resultado muy competitivo: $E_{out} = 15\,\%$. Además, el número de fallos en cada clase está bastante equilibrado, casi tanto como en el modelo KNN con base radial:

```
cat("Error obtenido en la clase <=50K: ",
    94700/length(adult.test.clean$income[adult.test.clean$income=="<=50K"]))
```

```
## Error obtenido en la clase <=50K:  8.336268
```

```
cat("Error obtenido en la clase >50K: ",
    154100/length(adult.test.clean$income[adult.test.clean$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  41.64865
```

Por tanto, la conclusión obtenida sobre éste modelo es, que a pesar de haber estado limitado por la implementación del paquete de `R`, hemos obtenido un resultado muy competitivos y de una alta calidad, tanto en cuanto a error fuera de la muestra como error en cada clase y equilibrio en clases desbalanceadas.

7 Comparación de los modelos obtenidos

Para concluir con los resultados obtenidos, vamos a hacer un estudio comparativo de todos los modelos que hemos ajustado en este proyecto. Para ello, vamos a representar la curva ROC de cada uno y realizar una tabla comparativa.

Cuadro 1: Comparativa de los resultados de los modelos ajustados

	Regresión logística	Random forest	Support Vector Machine	Vecino más cercano	Red Neuronal
E_{out}	24,56 %	18,1 %	14,78 %	16,52 %	15,69 %
$E_{out>50K}$	100 %	72, 95 %	40,9 %	41,65 %	37,59 %
$E_{out<=50K}$	0 %	0, 24 %	6,29 %	8,34 %	8,56 %

En la Tabla 1 vemos una comparativa de los modelos obtenidos. El modelo con menor E_{out} ha sido *Support Vector Machine* utilizando un $Kernel$ de base radial. Es el modelo con menor tasa de error en la clase minoritaria, ya que a diferencia de otros modelos, no ha "obviado" la clase minoritaria, aunque ha sido superado por la *Red Neuronal*, que lo ha conseguido con un error menor al 40 %. Este modelo ha conseguido balancear mejor el error entre ambas clases, con lo que puede generalizar un poco mejor que el modelo SVM, con el coste de aumentar un poco el error fuera de la muestra. Ahora bien, el modelo *Red Neuronal* ha jugado con desventaja, al no poder ajustar uno de los parámetros que más flexibilidad da al modelo: el número de capas ocultas, pero esto se debe a una limitación de la función del lenguaje. A pesar de esta desventaja, ha logrado ser el segundo mejor modelo.

El peor modelo ha sido la *Regresión logística*, ya que ha obviado la clase minoritaria completamente. Éste modelo ha obtenido un 24 % de error, ya que ha clasificado de forma correcta todos los ejemplos de la clase `<=50K` y sólo ha fallado en los de la clase `>50K`. Al obviar la clase minoritaria, el error fuera de la muestra no se ve demasiado afectado, pero, puede darse que esta clase minoritaria, fuera de nuestro conjunto de train y test tenga una representación mucho mayor y deje de ser minoritaria, con lo que el modelo estaría fallando enormemente. El modelo *Random Forest* también ha hecho una clasificación bastante mala, ya que el error obtenido en la clase `>50K` ha sido demasiado alto. Por tanto, al no haber podido trabajar correctamente con

datos desbalanceados estos dos modelos han obtenidos los peores resultados.

Por último, el modelo del *Vecino más cercano* con base radial ha obtenido buenos resultados, pero en comparación con *Support Vector Machine* o *Red Neuronal* ha obtenido un error mayor fuera de la muestra, debido a un mayor error en la clase minoritaria. Ha obtenido un error menor en `<=50K` que *Red Neuronal*, 8,34 frente a 8,65, pero esta diferencia no es significativa.

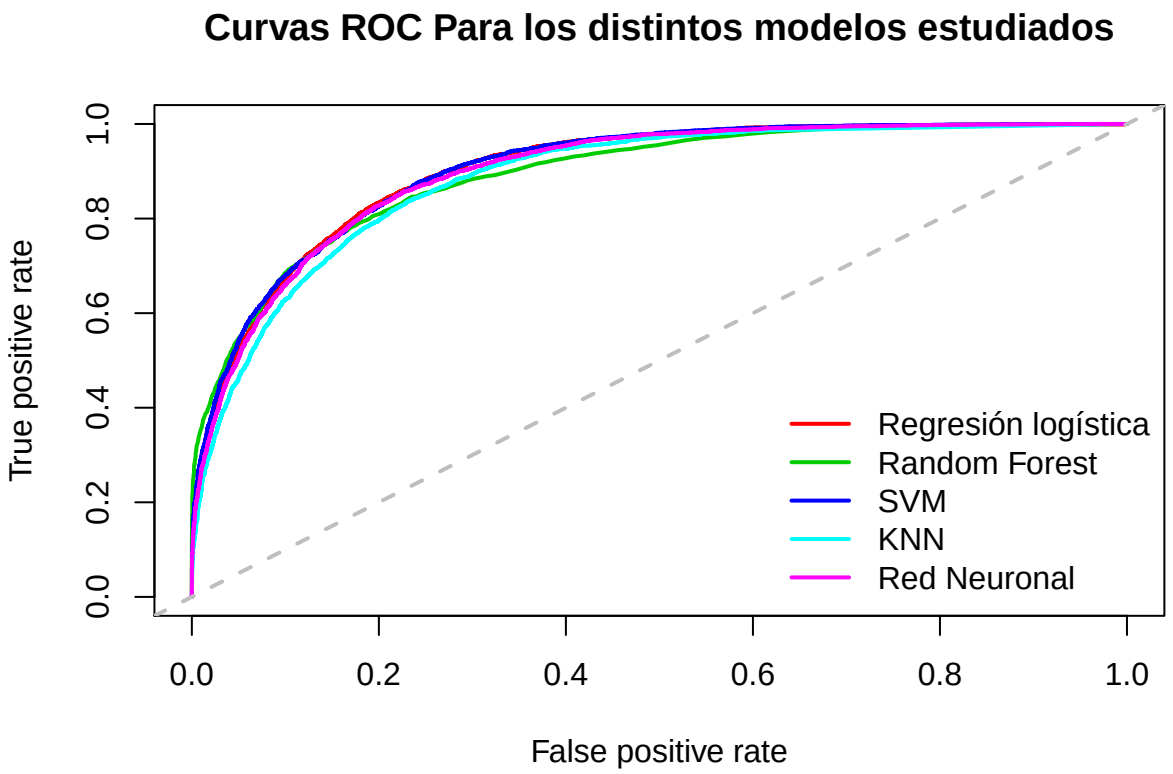
7.1 Curvas ROC de los modelos obtenidos

A continuación, podemos ver las curvas *ROC* asociadas a cada uno de los modelos estudiados:

```
library(ROCR)

getPerfomance <- function(model, newdata = adult.test.clean, svmPred = F,
                          nnetOrGlm=F, calculatePred = T,...){
    set.seed(1)
    if(calculatePred) {
        if(!nnetOrGlm && !svmPred)
            preds = predict(object = model, newdata = newdata, ...)[,2]
        else if (!svmPred && nnetOrGlm)
            preds = predict(object = model, newdata = newdata, ...)
        else{
            preds = attributes(predict(model, newdata, decision.values=T))$decision.values
            preds = preds*-1
        }
    } else
        preds = as.numeric(model)

    pred = prediction(preds, newdata$income)
    # tpr --> True Positive Rate
    # fpr --> False Positive Rate
    performance(pred, "tpr", "fpr")
}
plot(getPerfomance(model = glmModel, type="response", nnetOrGlm = T),
     col=2,lwd=2,main="Curvas ROC Para los distintos modelos estudiados")
plot(getPerfomance(model = rf.tuned, type = "prob"),col=3,lwd=2,add=T)
plot(getPerfomance(model = svmModelReg_09, svmPred = T),lwd=2,col=4,add=T)
plot(getPerfomance(model_knn$prob[,2], newdata = adult.test.clean.norm,
                   calculatePred = F),lwd=2,col=5,add=T)
plot(getPerfomance(model = model.nnet, newdata = adult.test.clean.norm,
  type="raw", nnetOrGlm = T),lwd=2,col=6,add=T)
abline(a=0,b=1,lwd=2,lty=2,col="gray")
legend("bottomright",col=c(2:6),lwd=2,legend=c("Regresión logística",
            "Random Forest","SVM","KNN","Red Neuronal"),bty='n')
```



Como podemos ver, las curvas que más destacan, a pesar de estar todas muy igualadas, son las curvas del modelo SVM y de la red neuronal frente al resto, lo que nos indica, que el ratio de aciertos es mayor en estos que en el resto de modelos.

Respecto a la curva del modelo de regresión logística, la tasa de aciertos de este modelo es "bastante alta" ya que el modelo, al obviar la clase minoritaria, ha conseguido acertar la clase mayoritaria. Al ser clase mayoritaria, la tasa de aciertos es muy alta frente a los otros modelos, aunque esto es "virtual".

8 Selección del mejor modelo

En base a los resultados, y el análisis de los errores que obtiene cada modelo, junto con el estudio de la curva ROC para cada modelo, podemos elegir como mejores modelos al modelo SVM y el modelo de *Red Neuronal*, ya que son los que mejores resultados ofrecen.

Respecto a la hora de elegir entre estos dos modelos, podemos elegir el modelo más sencillo, ya que, si seguimos el criterio de la *Navaja de Ockham*, si tenemos dos modelos que se comportan de forma similar, se debe elegir el modelo más "simple". Tal y como hemos visto, la dimensión d_{VC} de los modelos SVM puede llegar a ser infinita, mientras que para los modelos de redes neuronales, esta dimensión se establece como $d_{VC} = O(VQ)$, por lo que, la red neuronal es un modelo un poco más sencillo en este caso que el modelo SVM, a pesar de que el error es un poco peor. Otro aspecto a tomar en cuenta en la elección del modelo es la facilidad para poder explicar lo que hace este modelo para predecir los datos, y por ello en este aspecto gana el modelo SVM.

Por todo esto, el mejor modelo elegido es **el modelo de *Red Neuronal***.

9 Ajustando un modelo de red neuronal recuperando los datos perdidos

9.1 Recuperando datos perdidos

La idea de esto es recuperar todas las instancias de datos que hemos eliminado al contener valores perdidos. La forma de recuperar estos datos va a consistir en tratar de predecir esos datos con un modelo, y una vez predichos, ver cómo se comporta el nuevo modelo con todo el conjunto de datos al completo frente al que teníamos sin los datos perdidos.

Como hemos dicho antes, las variables que tienen valores perdidos son *Workclass*, *Occupation* y *Country*. Vamos a hacer tres modelos diferentes de red neuronal, esta vez con regularización *early stopping* (usando un límite de cien iteraciones).

```
set.seed(1)
# normalizamos todos los datos de la clase
norm = normalizar_maxmin(data = adult.train, data_test = adult.test)
adult.train.norm = norm[[1]]
adult.test.norm = norm[[2]]
norm = NULL

# debemos quitar las variables con NA para poder predecir correctamente
predice_na <- function(attr, noselectvars, formula, test_dataset = adult.test.norm,
                       train_dataset = adult.train.norm, dataset = adult.train.clean.norm) {
    # el vector no selectvars siempre tendrá dos variables string
    model <- nnet(formula = formula, maxit = 100,
                  data = subset(dataset, select=c(-which(colnames(dataset) == noselectvars[1]),
                                                  -which(colnames(dataset) == noselectvars[2]))),
                  size = 8, decay = 1, trace = F)
    pred_train <- predict(object = model, newdata = subset(train_dataset[is.na(train_dataset[,attr]),],
                          select=c(-which(colnames(train_dataset) == noselectvars[1]),
                                   -which(colnames(train_dataset) == noselectvars[2]))), type = "class")
    pred_test <- predict(object = model, newdata = subset(test_dataset[is.na(test_dataset[,attr]),],
                          select=-which(colnames(test_dataset) == noselectvars)), type = "class")
    list(pred_train, pred_test)
}

workclass = predice_na(attr = "workclass", noselectvars = c("country", "occupation"),
                       formula = workclass ~ .)
adult.train$workclass[is.na(adult.train$workclass)] = workclass[[1]]
adult.train.norm$workclass[is.na(adult.train.norm$workclass)] = workclass[[1]]
adult.test$workclass[is.na(adult.test$workclass)] = workclass[[2]]
adult.test.norm$workclass[is.na(adult.test.norm$workclass)] = workclass[[2]]

occupation = predice_na(attr = "occupation", noselectvars = c("country", "workclass"),
                        formula = occupation ~ .)
adult.train$occupation[is.na(adult.train$occupation)] = occupation[[1]]
adult.train.norm$occupation[is.na(adult.train.norm$occupation)] = occupation[[1]]
adult.test$occupation[is.na(adult.test$occupation)] = occupation[[2]]
adult.test.norm$occupation[is.na(adult.test.norm$occupation)] = occupation[[2]]

country = predice_na(attr = "country", noselectvars = c("occupation", "workclass"),
                     formula = country ~ .)
adult.train$country[is.na(adult.train$country)] = country[[1]]
adult.train.norm$country[is.na(adult.train.norm$country)] = country[[1]]
adult.test$country[is.na(adult.test$country)] = country[[2]]
```

```
adult.test.norm$country[is.na(adult.test.norm$country)] = country[[2]]
```

Una vez hecho esto, vemos que ya no tenemos ningún valor perdido en los datos:

```
apply(X=adult.train, MARGIN=2, FUN=function(columna) length(is.na(columna)[is.na(columna)==T]))
```

```
##            age      workclass         fnlwgt      education marital.status
##              0              0              0              0              0
##     occupation   relationship           race            sex   capital.gain
##              0              0              0              0              0
##   capital.loss hours.per.week        country         income
##              0              0              0              0
```

```
apply(X=adult.test, MARGIN=2, FUN=function(columna) length(is.na(columna)[is.na(columna)==T]))
```

```
##            age      workclass         fnlwgt      education marital.status
##              0              0              0              0              0
##     occupation   relationship           race            sex   capital.gain
##              0              0              0              0              0
##   capital.loss hours.per.week        country         income
##              0              0              0              0
```

9.2 Ajustando una red neuronal con el dataset completo

Una vez hemos conseguido predecir todos los valores perdidos, vamos a ajustar el mismo modelo de *Red Neuronal* ajustado anteriormente para ver la diferencia obtenida al usar más datos.

```
library(nnet)
set.seed(1)
model.nnet = nnet(formula = income ~ ., maxit=10000,
    data = adult.train.norm, size = 10, decay=0.1,trace=F)
outOfSampleError(model.nnet, adult.test.norm, type="class")
```

```
##         truth
## predict <=50K  >50K
##   <=50K 11481  1437
##   >50K    954  2409
## Eout =  0.1468583
```

En este caso, el E_{out} ha mejorado algo respecto del modelo anterior, bajando del 15 % de error al 14. Respecto al error obtenido en cada clase, vemos en la matriz de confusión que también se ha mejorado el error en ambas clases:

```
cat("Error obtenido en la clase <=50K: ",
    97200/length(adult.test$income[adult.test$income=="<=50K"]))
```

```
## Error obtenido en la clase <=50K:  7.816647
```

```
cat("Error obtenido en la clase >50K: ",
    143800/length(adult.test$income[adult.test$income==">50K"]))
```

```
## Error obtenido en la clase >50K:  37.3895
```

En ambas hemos reducido el error en un 1 % y se puede comprobar como el error del modelo se reduce al tener más instancias para poder entrenar el modelo y a la vez, el tener más instancias para datos de test nos da una mejor representación del error real que tenemos fuera de la muestra. Aun así, tras haber recuperado los datos perdidos, no existe una mejora sustancial en el modelo, ya que los datos que eliminamos para aprender el modelo inicial sólo suponían un 7.37 % del total de los datos que disponíamos, pero aun así, se puede notar cómo el modelo se comporta mejor que antes.

Referencias

[1] **M. Lichman**, «UCI Machine Learning Repository». University of California, Irvine, School of Information; Computer Sciences, 2013. Disponible en: http://archive.ics.uci.edu/ml

[2] **UCI**, «Census Income Data Set Description». Disponible en: http://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.names

[3] **U.S. Census Bureau**, «Marital Status: 2000». Disponible en: http://www.census.gov/prod/2003pubs/c2kbr-30.pdf

[4] **Universidad De Berkeley**, «Factors in R». 2009. Disponible en: http://www.stat.berkeley.edu/~s133/factors.html

[5] **Yaser S. Abu-Mostafa, Malik Magdon-Ismail, y Hsuan-Tien Lin**, *Learning From Data A Short Course*. AMLbook.com, pp. 89-93.

[6] **Gareth James, Daniela Witten, Trevor Hastie, y Robert Tibshirani**, *An Introduction To Statiscal Learning with Applications in R*. Springer, pp. 317-318.

[7] **Vaishali Ganganwar**, «An overview of classification algorithms for imbalanced datasets». 2012. Disponible en: http://www.ijetae.com/files/Volume2Issue4/IJETAE_0412_07.pdf

[8] **Christopher J.C. Burges**, «A tutorial on Support Vector Machines for Pattern Recognition». 1998. Disponible en: http://www-ai.cs.uni-dortmund.de/LEHRE/VORLESUNGEN/KDD/SS08/burges_svm_tutorial.pdf

[9] **Hechenbichler K. y Schliep K.P.**, «Weighted k-Nearest-Neighbor Techniques and Ordinal Classification». 2004. Disponible en: http://www.stat.uni-muenchen.de/sfb386/papers/dsp/paper399.ps

[10] **Trevor Hastie, Robert Tibshirani, y Jerome Friedman**, *The Elements of Statiscal Learning: Data Mining, Inference, and Prediction*, Second. Springer, 2008, pp. 416-420.