

Flujo de análisis en clasificación supervisada

Métodos supervisados

Laura Rodríguez Navas

Septiembre 2020

Contenido

| | |
|---|-----------|
| Análisis exploratorio de los datos | 2 |
| Variable <i>target</i> | 3 |
| Variable <i>keyword</i> | 5 |
| Variable <i>location</i> | 7 |
| Variable <i>id</i> | 8 |
| Conclusión análisis exploratorio | 9 |
| Procesamiento de texto | 9 |
| Corpus | 9 |
| Limpieza del texto | 10 |
| Creación de un modelo predictivo | 11 |
| Preprocesado de los datos | 11 |
| Feature Extraction mediante SVD | 12 |
| División de los datos en train y test | 13 |
| Selección de variables | 14 |
| Entrenamiento de modelos | 15 |
| Regresión Logística | 17 |
| Random Forest | 18 |
| Gradient Boosting | 19 |
| SVM | 20 |
| Evaluación de modelos mediante resampling | 21 |
| Wilcoxon signed-rank test | 23 |
| Conclusión evaluación de modelos | 24 |
| Predicción | 25 |
| Resultado en Kaggle | 26 |
| Conclusiones | 27 |
| Referencias | 27 |

Empezamos por cargar a nuestro espacio de trabajo los paquetes que usaremos:

- **tidyverse**, engloba otros paquetes (**dplyr**, **tidyr**, **ggplot**, etc.) que facilitan en gran medida el análisis exploratorio de los datos.
- **tm**, específico para minería de textos.
- **irlba**, específico para la *Descomposición de Valores Singulares (SVD)* de matrices muy grandes.
- **caret**, para realizar tareas de clasificación y regresión.
- **doParallel**, proporciona computación paralela.
- **syuzhet**, específico para la extracción de sentimientos (información subjetiva) de textos.
- **ggcorrplot**, muestra visualizaciones gráficas de matrices de correlación usando **ggplot2**.

```
library(tidyverse)
library(tm)
library(irlba)
library(caret)
library(doParallel)
library(syuzhet)
library(ggcorrplot)
```

Análisis exploratorio de los datos

Antes de entrenar un modelo predictivo, o incluso antes de realizar cualquier cálculo con un nuevo conjunto de datos, es muy importante realizar una exploración descriptiva de los datos. Este proceso nos permite entender mejor que información contienen cada variable, detectar posibles errores, etc. además, puede dar pistas sobre qué variables no son adecuadas para predecir un modelo.

Acorde a la realización del ejercicio propuesto se ha elegido la competición en Kaggle: *Real or Not? NLP with Disaster Tweets*. El dataset de la competición se puede encontrar en el siguiente [enlace](#). Este dataset, con 10.876 instancias, contiene 4 variables explicativas: **id**, **keyword**, **location** y **text**, y dos valores en la variable clase **target** (0 y 1). Como observaremos la variable clase es binaria, así que, durante este ejercicio vamos a crear un modelo de *clasificación binaria*. El objetivo de este modelo será predecir si dado un tweet, éste trata sobre un desastre real o no. Si un tweet trata sobre un desastre real, se predice un 1. Si no, se predice un 0.

La **clasificación binaria** es un tipo de clasificación en el que tan solo se pueden asignar dos clases diferentes (0 o 1).

La métrica de evaluación esperada por la competición de Kaggle es **F1**. Para más información consultar el siguiente [enlace](#). Se calcula de la siguiente manera:

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

donde:

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

- True Positive (TP)
- False Positive (FP)
- False Negative (FN)

La partición inicial train-test, no se tiene que realizar, ya que las instancias de train y test ya vienen definidas en el dataset de la competición de Kaggle. Solo tendremos que descargar a nuestro espacio de trabajo los ficheros **train.csv** y **test.csv** des del siguiente [enlace](#)).

Cargamos a nuestro espacio de trabajo los conjuntos de datos de train y test que hemos descargado con la función **read.csv()**, renombrando los valores perdidos como **NA** para que los podamos tratar más adelante.

Además, mostramos las dimensiones de los conjuntos de datos de train y test usando la función **dim()**. Esta función nos mostrará el número total de observaciones y de variables de cada conjunto de datos.

```
train <- read.csv("train.csv", na.strings=c("", "NA"))
test  <- read.csv("test.csv", na.strings=c("", "NA"))
dim(train)
```

```
## [1] 7613    5
```

```
dim(test)
```

```
## [1] 3263    4
```

Vemos que el conjunto de datos de train contiene 7613 instancias y el conjunto de datos de test contiene 3263 instancias. Cada instancia contiene las siguientes variables:

- **id**: un identificador único para cada tweet.
- **keyword**: una palabra clave del tweet.
- **location**: la ubicación desde la que se envió el tweet.
- **text**: el texto del tweet.
- **target**: solo en el conjunto de datos de train porque es la variable clase, la variable a predecir. Indica si un tweet corresponde a un desastre real (1) o no (0).

El conjunto de datos de train:

```
## 'data.frame':    7613 obs. of  5 variables:
## $ id          : int  1 4 5 6 7 8 10 13 14 15 ...
## $ keyword     : chr  NA NA NA NA ...
## $ location    : chr  NA NA NA NA ...
## $ text        : chr  "Our Deeds are the Reason of this #earthquake May ALLAH Forgive "..
## $ target      : int  1 1 1 1 1 1 1 1 1 1 ...
```

El conjunto de datos de test:

```
## 'data.frame':    3263 obs. of  4 variables:
## $ id          : int  0 2 3 9 11 12 21 22 27 29 ...
## $ keyword     : chr  NA NA NA NA ...
## $ location    : chr  NA NA NA NA ...
## $ text        : chr  "Just happened a terrible car crash" "Heard about #earthquake is"..
```

Variable *target*

Como acabamos de comentar en la sección anterior, la variable **target** es la variable clase y la variable a predecir. La variable es de tipo cuantitativa, concretamente de tipo entero, y conviene transformarla para que sea una variable de tipo cualitativa (categórica).

Para transformar la variable a predecir en una variable categórica usamos la función **as.factor()**. Esto lo hacemos para evitar errores futuros, y además se recodifica para que sus dos posibles valores sean “Yes” y “No”.

```
train$target <- as.factor(ifelse(train$target == 0, "No", "Yes"))
ggplot(train, aes(x=target)) + geom_bar(aes(fill=target))
```

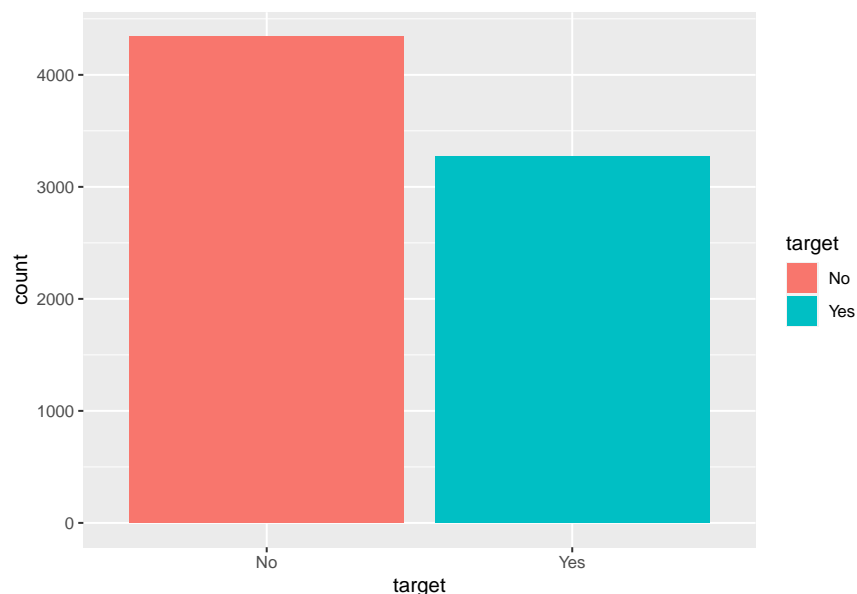


Figura 1: distribución de la variable **target**.

Cuando se quiere crear un modelo de predicción, es muy importante estudiar la distribución de la variable clase, en este caso la variable a predecir, ya que, a fin de cuentas, es lo que nos interesa predecir.

Gráficamente (ver Figura 1) observamos que la distribución de la variable a predecir no está muy sesgada y que está relativamente equilibrada. Aunque si nos fijamos hay menos tweets que se refieren a desastres reales (columna "Yes"). Así, la clase "Yes" representaría a la clase minoritaria y la clase "No", a la clase mayoritaria. Tampoco parece que presente un problema notable de *desequilibrio de clases*, porqué contamos con muchas observaciones de la clase minoritaria.

Al enfrentarse a la situación de crear un modelo de clasificación binaria es habitual que las clases no se encuentren balanceadas. Esto es, el número de observaciones para una de las clases es inferior a la otra. Cuando el desequilibrio es pequeño, esto no supone un problema, pero cuando el desequilibrio es grande es un problema para la mayoría de los modelos de clasificación. Esta situación se conoce como el **problema del desequilibrio de clases**.

```
sum(train$target == "Yes") / dim(train)[1] * 100
```

```
## [1] 42.96598
```

```
sum(train$target == "No") / dim(train)[1] * 100
```

```
## [1] 57.03402
```

Concretamente, el 57% de los tweets corresponden a desastres no reales, y aproximadamente el 43% corresponden a desastres reales. Para que el modelo predictivo que crearemos nos sea útil, tendremos que intentar superar ese porcentaje mínimo del 57% de la clase mayoritaria.

Como el objetivo del ejercicio es predecir que tweets pertenecen o no a un desastre real, lo que haremos a continuación, es analizar las variables explicativas del conjunto de datos de train en relación con a la variable a predecir **target**. De este análisis se podrán extraer ideas sobre que variables están más relacionadas con los desastres reales.

Variable *keyword*

La variable explicativa **keyword** representa una palabra clave en cada tweet. Vemos las 10 primeras del conjunto de datos de train usando las funciones **select()**, **unique()** para no ver las variables repetidas, y **head()** donde le decimos que cantidad de variables queremos ver.

```
train %>% select(keyword) %>% unique() %>% head(10)
```

```
##           keyword
## 1             <NA>
## 32          ablaze
## 68          accident
## 103         aftershock
## 137 airplane%20accident
## 172          ambulance
## 210        annihilated
## 244        annihilation
## 273          apocalypse
## 305         armageddon
```

Nuestro interés en la variable **keyword**, dentro del análisis exploratorio, es ver si existen correlaciones entre ella y la variable a predecir **target**. Para ello, y como estamos delante un ejercicio de [Procesamiento del Lenguaje Natural](#), realizaremos un **análisis de sentimientos**.

El **análisis de sentimientos** es una técnica de [Machine Learning](#), basada en el Procesamiento del Lenguaje Natural, que pretende obtener información subjetiva de una serie de textos.

Para el análisis de sentimientos usamos los paquetes: **syuzhet**, **ggcorrplot** y **doParallel**.

- **syuzhet** cuenta con la función **get_nrc_sentiment()** que calculará la presencia de los diferentes sentimientos dado un conjunto de palabras clave (variable **keyword**).

Los argumentos de esta función son:

- **char_v**: un vector de caracteres que en este caso contendrá todas las palabras clave.
- **language**: define el lenguaje. Como los tweets están en inglés, el lenguaje será inglés.
- **cl**: para el análisis en paralelo. Es un argumento opcional, pero en este caso se usará porque hay muchas palabras clave.

- **doParallel** cuenta con las funciones:

- **makePSOCKcluster()**: crea un clúster de sockets paralelos y inicia la computación paralela con estrategia secuencial.
- **registerDoParallel()**: registra el número de *cores* que usará el clúster creado.
- **stopCluster()**: detiene la computación paralela.

La computación paralela la usaremos en muchas de las ejecuciones de este ejercicio ya que nos encontramos delante de un problema de **alta dimensionalidad**. Eso es, que la dimensionalidad de nuestro dataset es tan elevada que puede reducir drásticamente la eficiencia de los modelos de clasificación supervisada que entrenaremos, además de producir un alto coste computacional.

Para solucionar el problema de la alta dimensionalidad de los datos, se realiza una **reducción de la dimensionalidad** (reducción del número de variables). La aplicaremos en este ejercicio, pero más adelante. Durante este proceso usaremos técnicas de **selección de variables** y **extracción de características**.

El análisis de sentimientos de nuestro ejercicio, consiste en extraer los sentimientos de cada palabra clave, con la función **get_nrc_sentiment()**. Guardar los sentimientos en un nuevo conjunto de datos (**emotion.df**). Y finalmente calcular la matriz de correlaciones (ver Figura 2). Para calcular i visualizar la matriz de

correlaciones usamos los paquetes **cor** y **ggcorrplot**. Es muy importante volver a transformar la variable a predecir **target**, a variable de tipo cuantitativa, porque sino no podremos crear la matriz.

Veremos que se usa la función **gsub()** para eliminar los guiones bajos de las palabras clave y el argumento **lab** en la función **ggcorrplot()** para visualizar los coeficientes de correlación.

```
cl <- makePSOCKcluster(4, setup_strategy="sequential")
registerDoParallel(cl)

emotion.df <- get_nrc_sentiment(char_v = gsub("_", " ", train$keyword),
                              language = "english", cl=cl)

emotion.df <- emotion.df %>% data.frame(target = train$target)

emotion.df$target <- as.numeric(emotion.df$target)

cor(emotion.df) %>%
  ggcorrplot(lab = TRUE,
            title = "Matriz de correlación entre \nkeyword y target",
            legend.title = "correlation")
```

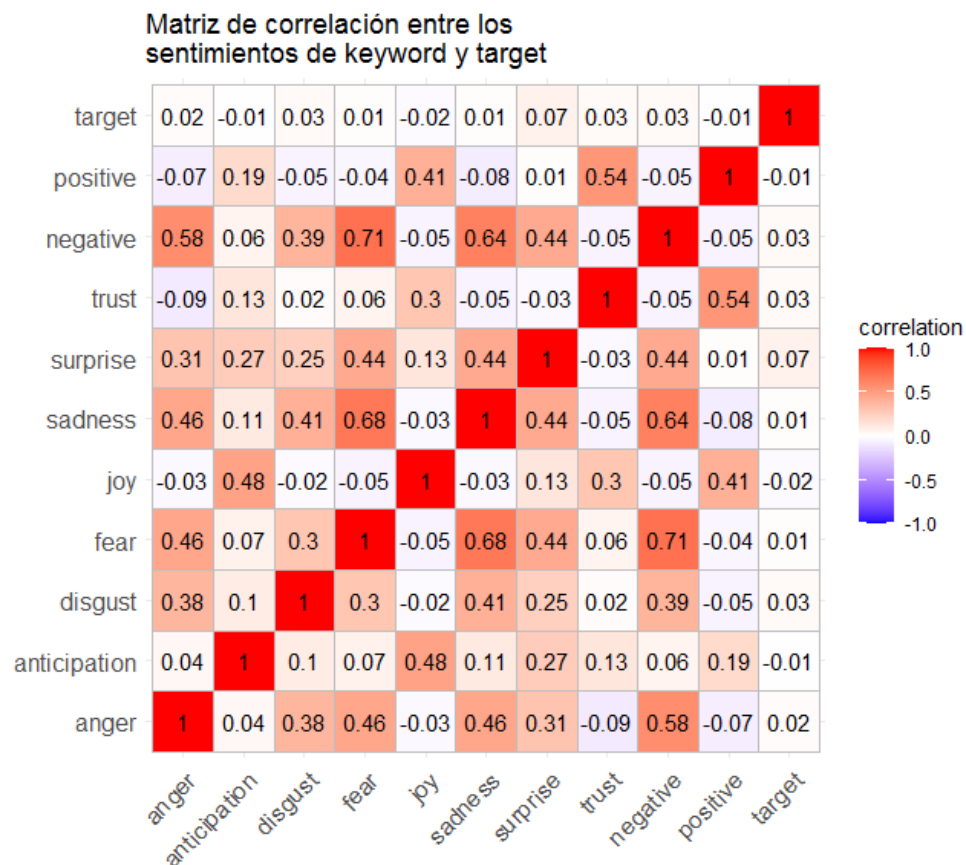


Figura 2: matriz de correlaciones.

```
stopCluster(cl)
```

Parece que, al observar la matriz de correlaciones, existe una correlación nula entre las variables **keyword** y **target**. Si lo revisamos con mayor detalle, podemos observar que la mayoría de las palabras clave no tienen un sentimiento positivo asociado, muchos de sus coeficientes se encuentran dentro del umbral: (-1.0, 0.5). Las palabras clave asociadas a un sentimiento se asocian negativamente (p. ej. miedo o tristeza), lo cual es bastante consistente con el ejercicio, ya que intentemos predecir desastres reales. Teniendo en cuenta el análisis exploratorio de la variable **keyword** y acorde a nuestro criterio, no se considera buena para hacer predicciones ya que no está realmente asociada con la variable a predecir. Así que la excluirémos en el procesamiento de texto.

Variable *location*

La variable explicativa **location** representa a las ubicaciones donde se generaron los tweets. Vemos las 8 primeras y el número total de ubicaciones diferentes del conjunto de datos de train (3342 ubicaciones sin repeticiones).

```
train %>% select(location) %>% unique() %>% head(10)
```

```
##              location
## 1              <NA>
## 32            Birmingham
## 33 Est. September 2012 - Bristol
## 34              AFRICA
## 35      Philadelphia, PA
## 36              London, UK
## 37              Pretoria
## 38      World Wide!!
```

```
count(train %>% select(location) %>% unique())
```

```
##      n
## 1 3342
```

A continuación, vemos las ubicaciones que se repiten más de 10 veces en el conjunto de datos de train. Con las funciones **table**, **unlist** y **select** creamos una tabla a partir de una lista única de ubicaciones. Con la función **which** seleccionamos las que se repiten más de 10 veces y con la función **barplot** las visualizamos.

```
location.freq <- table(unlist(train %>% select(location)))
location.freq[which(location.freq > 10)]
```

```
##
##      Australia      California      California, USA      Canada
##           18           17           15           29
##      Chicago      Chicago, IL           Earth      Everywhere
##           11           18           11           15
##      Florida           India      Indonesia      Ireland
##           14           24           13           12
##           Kenya      London      Los Angeles      Los Angeles, CA
##           20           45           13           26
##           Mumbai      New York      New York, NY      Nigeria
##           22           71           15           28
##           NYC      San Francisco      San Francisco, CA      Seattle
##           12           14           11           11
##           Toronto           UK      United Kingdom      United States
##           12           27           14           50
##           USA      Washington, D.C.      Washington, DC      Worldwide
##           104           13           21           19
```

```
barplot(location.freq[which(location.freq>10)], las = 2,
        ylab = "frequency")
```

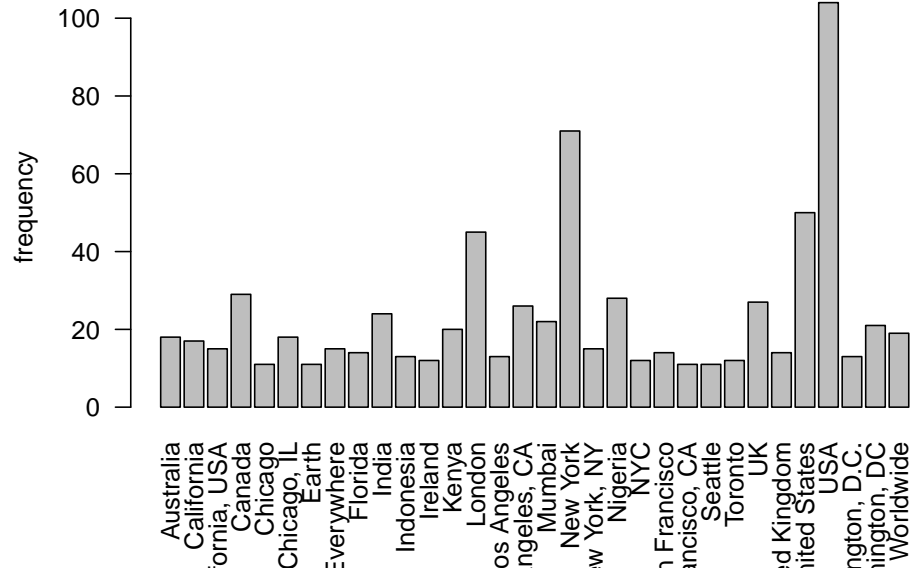


Figura 3: ubicaciones que se repiten más de 10 veces.

El argumento **las** de la función **barplot** se usa para poner horizontalmente los nombres de las ubicaciones en el *eje-x* de la Figura 3.

Del total de ubicaciones, 3342, la mayoría de ellas cuenta con menos de 10 observaciones. Teniendo en cuenta el análisis exploratorio de la variable **location** y acorde a nuestro criterio, no se considera buena para hacer predicciones, ya que la variable consta de muy pocas observaciones dentro del conjunto de datos (no es representativa), y puede ocurrir que, durante la realización de la validación cruzada o *bootstrapping*, algunas de las muestras no contengan ninguna observación de dicha variable, lo que puede dar lugar a errores.

Una muestra *bootstrap* es una muestra obtenida a partir de la muestra original por muestreo aleatorio con reposición, y del mismo tamaño que la muestra original. Muestreo aleatorio con reposición (resampling with replacement) significa que, después de que una observación sea extraída, se vuelve a poner a disposición para las siguientes extracciones. Como resultado de este tipo de muestreo, algunas observaciones aparecerán múltiples veces en la muestra *bootstrap* y en otras ninguna.

La variable **location** también la excluirémos en el procesamiento de texto.

Variable *id*

La variable **id** es solo un identificador único, así que, no la analizaremos y procederemos a eliminarla de los conjuntos de datos de train y test.

```
train$id <- NULL
test$id <- NULL
```


Conclusión análisis exploratorio

Llegados a este punto, después de un análisis exploratorio de las variables explicativas **keyword**, **location** y **id**, y el estudio de la distribución de la variable a predecir con sus posibles relaciones con estas variables, nuestro criterio es descartarlas para las futuras predicciones y el procesamiento de texto. Consecuentemente nos centraremos en la variable **text** en la siguiente sección.

Procesamiento de texto

Combinamos los conjuntos de datos de train y test para ahorrar esfuerzos en el preprocesado de datos. Para ello, usamos la función **bind_rows()** del paquete **dplyr**, que nos permitirá enlazar de forma eficiente los dos conjuntos de datos por fila y columna. Podremos comprobar que la combinación se hace correctamente, sumando los elementos del conjunto de datos de train (7613) y los elementos del conjunto de datos de test (3263), el nuevo conjunto de datos (**complete_df**) tendrá 10876 observaciones, 3 variables explicativas (**keyword**, **location**, **text**) y la variable de clase **target**.

```
complete_df <- bind_rows(train, test)
str(complete_df, width = 85, strict.width = "cut")

## 'data.frame':    10876 obs. of  4 variables:
## $ keyword : chr  NA NA NA NA ...
## $ location: chr  NA NA NA NA ...
## $ text    : chr  "Our Deeds are the Reason of this #earthquake May ALLAH Forgive "..
## $ target  : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 2 2 2 2 2 ...
```

El preprocesado de datos englobará las transformaciones de los textos de la variable **text**, como, por ejemplo la imputación de los valores ausentes o la reducción de dimensionalidad del conjunto de datos **complete_df**.

Ahora, vamos a mirar cuantos valores perdidos contiene nuestro conjunto de datos **complete_df**. La función **colSums()** sumará los valores que la función **sapply()** del paquete **base** encuentre, en este caso, los valores perdidos.

```
colSums(sapply(complete_df, is.na))

## keyword location      text      target
##      87      3638         0      3263
```

Identificamos que las variables explicativas **keyword** y **location** contienen valores perdidos. La variable explicativa **text** no contienen valores perdidos. Además, hay una gran cantidad de tweets para los cuales falta su ubicación. Los 3263 valores perdidos de la variable a predecir provienen del conjunto de datos de test.

Nos ocuparemos de los valores perdidos más adelante.

Corpus

Un corpus lingüístico se define como “un conjunto de textos de un mismo origen” y que tiene por función recopilar conjuntos de textos. El uso de un corpus lingüístico nos permitirá obtener información de las palabras utilizadas con más o menor frecuencia dentro del conjunto de datos.

Con el nuevo conjunto de datos, **complete_df**, procedemos a crear nuestro Corpus, es decir, un conjunto de textos a partir de los textos de la variable **text**. El Corpus que se compondrá de todos los textos de los tweets, lo asignaremos al objeto *myCorpus* usando las funciones **VectorSource()** y **Corpus()**. La función **Corpus()** creará el Corpus a partir de un vector de textos que contiene los textos de todos los tweets. La función **VectorSource()** interpretará cada mensaje de texto de los tweets como un elemento de ese vector de textos.

```
myCorpus <- Corpus(VectorSource(complete_df$text))
myCorpus
```

```
## <<SimpleCorpus>>
## Metadata:  corpus specific: 1, document level (indexed): 0
## Content:   documents: 10876
```

Nuestro Corpus está compuesto por 10876 conjuntos de textos.

Limpieza del texto

Como tenemos 10876 conjuntos de textos, necesitamos limpiarlos de caracteres que son de poca utilidad. Eso nos ayudará a reducir su dimensionalidad. Para ello, mayoritariamente usaremos las funciones `gsub()` y `tm_map()` del paquete `tm`. La función `gsub()` buscará y reemplazará desde la primera hasta la última de las coincidencias de un patrón (representado por una *regular expression*). La función `tm_map()` será la encargada de aplicar las diferentes transformaciones en los conjuntos de textos.

Una expresión regular (o en inglés *regular expression*) es una representación, según unas reglas sintácticas de un lenguaje formal, de una porción de texto genérico a buscar dentro de otro texto, como por ejemplo caracteres, palabras o patrones de texto concretos.

Empezamos la limpieza eliminando los enlaces.

```
removeURL <- function(x) gsub("http[[:space:]]*", "", x)
myCorpus <- tm_map(myCorpus, content_transformer(removeURL))
```

Convertimos todo a minúsculas.

```
myCorpus <- tm_map(myCorpus, content_transformer(tolower))
```

Eliminamos los nombres de usuario.

```
removeUsername <- function(x) gsub("@[[:space:]]*", "", x)
myCorpus <- tm_map(myCorpus, content_transformer(removeUsername))
```

Nos deshacemos de la puntuación, puesto que por ejemplo “fin” y “fin.” son identificadas como palabras diferentes, lo cual no deseamos.

```
removeNumPunct <- function(x) gsub("[^[:alpha:]][[:space:]]*", "", x)
myCorpus <- tm_map(myCorpus, content_transformer(removeNumPunct))
```

Usamos la función `stopwords("english")`, recordemos que los textos de los tweets están en inglés, y cada idioma tiene sus propias palabras vacías; para eliminar las palabras vacías, es decir, aquellas que consideramos que tienen poco valor de análisis para nuestro ejercicio, que carecen de un significado por sí solas, tales como artículos, preposiciones, conjunciones, pronombres, etc.

```
myStopWords <- c((stopwords('english')),
  c("really", "tweets", "saw", "just", "feel", "may", "us", "rt", "every", "one",
    "amp", "like", "will", "got", "new", "can", "still", "back", "top", "much",
    "near", "im", "see", "via", "get", "now", "come", "oil", "let", "god", "want",
    "pm", "last", "hope", "since", "everyone", "food", "content", "always", "th",
    "full", "found", "dont", "look", "cant", "mh", "lol", "set", "old", "service",
    "city", "home", "live", "night", "news", "say", "video", "people", "ill",
    "way", "please", "years", "take", "homes", "read", "man", "next", "cross",
    "boy", "bad", "ass"))
```

```
head(myStopWords, 30)
```

```
## [1] "i"          "me"          "my"          "myself"      "we"
## [6] "our"        "ours"        "ourselves"   "you"         "your"
## [11] "yours"      "yourself"    "yourselves" "he"          "him"
## [16] "his"        "himself"     "she"         "her"         "hers"
## [21] "herself"    "it"          "its"         "itself"      "they"
## [26] "them"       "their"       "theirs"      "themselves" "what"
```

```
myCorpus <- tm_map(myCorpus, removeWords, myStopWords)
```

Podemos observar que se han añadido (de manera aleatoria) algunas palabras vacías de más como: “really”, “tweets” y “saw”. Porqué son de las más usadas en los mensajes de texto de los tweets en Twitter (ver el siguiente [enlace](#) para ver las 500 palabras más usadas en esta red social). Si son palabras muy utilizadas contendrán poco valor de análisis para nuestro ejercicio.

Eliminamos las palabras de una sola letra.

```
removeSingle <- function(x) gsub(" . ", " ", x)
myCorpus <- tm_map(myCorpus, content_transformer(removeSingle))
```

Por último, eliminamos los espacios vacíos excesivos. Muchos de ellos son introducidos por las transformaciones anteriores.

```
myCorpus <- tm_map(myCorpus, content_transformer(stripWhitespace))
```

La limpieza del texto, es un proceso básico dentro del Procesamiento del Lenguaje Natural (PLN), que reduce de manera significativa la longitud del conjunto de textos del Corpus.

Creación de un modelo predictivo

Para la creación de un modelo predictivo necesitamos construir una **Term Document Matrix** del conjunto de textos de nuestro Corpus, donde cada fila representará un texto y cada palabra (única) de un texto estará representada por una columna. Con esta matriz podremos empezar el preprocesado de datos y crear nuestro modelo predictivo más adelante.

Una **Term Document Matrix** es una matriz matemática que describe la frecuencia con la que se repiten una serie de palabras en una colección de textos.

Preprocesado de los datos

Sabemos que el preprocesado de datos engloba aquellas transformaciones de los datos con la finalidad de mejorar los resultados de la clasificación supervisada. Todo preprocesado de datos debe aprenderse de las observaciones del conjunto de datos de train y luego aplicarse a los conjuntos de datos de train y de test. Esto es muy importante para no violar la condición de que ninguna información procedente de las observaciones del conjunto de datos de test influya en el ajuste del modelo predictivo.

Comenzaremos mapeando nuestro Corpus, indicando que es una Term Document Matrix. Para ello, utilizamos la función **TermDocumentMatrix()** del paquete **tm** y asignaremos el resultado al objeto **complete.tdm**.

Con el argumento **control** de la función **TermDocumentMatrix()**, indicamos que evaluamos y incluimos todos los textos que tengan una longitud superior a 4 caracteres hasta *Inf*, donde *Inf* es el valor Infinito. De esta manera, conseguimos incluir la mayoría de los textos del Corpus en la Term Document Matrix. Por defecto la función **TermDocumentMatrix()** usa el valor numérico *tf-idf*, que mide la importancia relativa de cada palabra de los textos, es decir, mide su frecuencia de ocurrencia. El valor *tf-idf* aumenta según el número de veces que una palabra aparece en un texto y no en otros, lo que permite averiguar qué palabras son más comunes en este texto respecto a los otros.

```
complete.tdm <- TermDocumentMatrix(myCorpus, control=list(wordLengths= c(4, Inf)))
complete.tdm
```

```
## <<TermDocumentMatrix (terms: 16880, documents: 10876)>>
## Non-/sparse entries: 76219/183510661
## Sparsity          : 100%
## Maximal term length: 49
## Weighting         : term frequency (tf)
```

Podemos observar que tenemos 16880 *terms* en 10876 textos, esto quiere decir que tenemos 16880 palabras diferentes en 10876 textos. Lo cual es una cantidad considerable de palabras de la Term Document Matrix, pero no esperaríamos otra cosa de una red social como Twitter. La palabra más larga contiene 49 caracteres.

Por ese motivo, usaremos la función **removeSparseItems()** del paquete **tm** para depurar nuestra Term Document Matrix de aquellas palabras que aparecen con muy poca frecuencia, es decir, que son dispersas. Con demasiadas palabras sería posible que no podamos entrenar nuestro modelo debido a grandes restricciones computacionales.

La función **removeSparseItems()** requiere que especifiquemos el argumento **sparse**, que puede asumir valores de 0 a 1. Este valor representa la dispersión de las palabras que queremos conservar. Si lo fijamos muy alto (cerca de 1, pero no 1), conservaremos muchas palabras, casi todas, pues estamos indicando que queremos conservar palabras, aunque sean muy dispersas. Naturalmente, ocurre lo opuesto si fijamos este valor muy bajo (cerca de 0, pero no 0), pudiendo incluso quedarnos sin ninguna palabra, si las palabras en nuestros textos son bastante dispersas en general.

Para nuestro ejercicio, se decide fijar el valor del argumento **sparse** a *.9975*, con la intención de conservar las palabras que aparecen al menos en el 0.25% de las observaciones. Lo vemos a continuación.

```
complete.tdm <- removeSparseTerms(complete.tdm, sparse = .9975)
complete.tdm
```

```
## <<TermDocumentMatrix (terms: 582, documents: 10876)>>
## Non-/sparse entries: 31214/6298618
## Sparsity          : 100%
## Maximal term length: 17
## Weighting         : term frequency (tf)
```

De las 16880 palabras que teníamos, nos hemos quedado con 582, lo cual reduce en gran medida la dificultad y complejidad del problema de *alta dimensionalidad*, lo cual es muy deseable. La palabra más larga ahora contiene 17 caracteres.

Feature Extraction mediante SVD

La descomposición de los datos originales en un nuevo conjunto de datos, sin necesidad de pérdida de información relevante y sacando a la luz la información existente, es un proceso de vital importancia para implementar la parte más computacionalmente intensa correspondiente a la [minería de textos](#) del ejercicio.

Buscando una intuitiva separabilidad de las clases de los datos aplicaremos la técnica denominada como *Descomposición en Valores Singulares (SVD)*, en nuestra Term Document Matrix.

La [Descomposición de Valores Singulares](#) (en inglés Singular Value Decomposition (SVD)) es una técnica de reducción de la dimensionalidad en minería de textos, que puede utilizarse para descubrir las dimensiones existentes que determinan similitudes semánticas entre las palabras (unidades léxicas) o entre los textos (unidades de contexto).

Para aplicar la técnica denominada como *Descomposición en Valores Singulares (SVD)* tenemos que transformar nuestra Term Document Matrix en una **matriz transpuesta** que llamaremos **complete.term.matrix** para la factorización de la misma.

Para crear la matriz transpuesta usamos la función **t()** del paquete **base** y la convertimos en un objeto de tipo **matrix**. Nos aseguraremos que la matriz transpuesta no contiene valores perdidos, usando la función **which()**.

```
complete.term.matrix <- as.matrix(t(complete.tdm))
which(!complete.cases(complete.term.matrix))
```

```
## integer(0)
```

La matriz transpuesta no contiene valores perdidos, podemos comenzar con la factorización de ésta. Para ello usaremos la función **irlba()**, que descompondrá la matriz transpuesta en los 150 vectores singulares más importantes, después de buscar un máximo de 600 iteraciones. El resultado se almacenará en el objeto **complete_irlba**.

```
c1 <- makePSOCKcluster(4, setup_strategy="sequential")
registerDoParallel(c1)

complete_irlba <- irlba(t(complete.term.matrix), nv = 150, maxit = 600)
complete_irlba$v[1:10, 1:5]
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] -0.0003162281 -2.109638e-04 -3.318231e-04 -2.035137e-05 0.0008660149
## [2,] -0.0431207327 1.408993e-02 5.124787e-03 3.922969e-03 -0.0021251369
## [3,] -0.0020297768 2.205792e-04 -5.231801e-05 1.014223e-04 0.0008899277
## [4,] -0.0054444920 1.259299e-04 -3.441743e-03 1.669137e-04 0.0094345444
## [5,] -0.0021975777 8.533917e-05 7.910354e-05 -7.809264e-05 0.0017043646
## [6,] -0.0449020384 1.303972e-02 1.677561e-03 3.883599e-03 -0.0005116573
## [7,] -0.0060579030 -8.521515e-03 -3.239198e-02 6.490016e-04 -0.0023784289
## [8,] -0.0387247287 1.297783e-02 5.140001e-03 3.764856e-03 -0.0112714576
## [9,] -0.0079142623 -5.649745e-04 -6.691751e-04 -4.606357e-04 0.0135358450
## [10,] -0.0014005942 -2.762898e-04 -3.440679e-04 -4.573951e-05 0.0016280077
```

```
stopCluster(c1)
```

Lo que vemos arriba es una pequeña parte de la matriz transpuesta factorizada, que usaremos como característica principal para la clasificación supervisada. Porqué con el rango de factorización, que indica las similitudes de las palabras en los textos, nos permitirá agrupar las palabras para su clasificación.

División de los datos en train y test

Evaluar la capacidad predictiva de un modelo consiste en comprobar cómo de próximas son sus predicciones a los verdaderos valores de la variable a predecir. Para poder cuantificar lo de forma correcta, se necesita disponer de un conjunto de observaciones, de las que se conozca la variable a predecir, pero que el modelo no haya “visto”, es decir, que no hayan participado en su ajuste. Con esta finalidad, se dividen los datos que tenemos en **complete_df** en un conjunto de datos de train y un conjunto de datos de test.

Con el conjunto de datos, del que se conoce la variable a predecir (**complete_df**) y la matriz transpuesta que hemos creado y factorizado en la sección anterior (**complete_irlba**) creamos un nuevo objeto de tipo dataframe (**complete.svd**) que a continuación dividiremos en un conjunto de datos de train (**train.df**) y en un conjunto de datos de test (**test.df**), con el mismo tamaño de la partición inicial proporcionada por la competición de Kaggle que hemos elegido para nuestro ejercicio. En esta partición inicial el conjunto de datos de train cuenta con 7613 instancias y el conjunto de datos de test con 3263 instancias. Ver sección del análisis exploratorio de los datos para comprobarlo.

```
complete.svd <- data.frame(target=complete_df$target, complete_irlba$v)
train.df <- complete.svd[1:7613, ]
test.df <- complete.svd[7614:10876, -1]
dim(train.df)
```

```
## [1] 7613 151
```

```
dim(test.df)
```

```
## [1] 3263 150
```

Vemos que usando la función `dim()`, los nuevos conjuntos de datos de train (`train.df`) y de test (`test.df`) tienen el mismo número de observaciones que la partición inicial.

Otra comprobación importante, es verificar que la distribución de la variable a predecir `target`, del nuevo conjunto de datos de train `train.df`, no ha cambiado respecto a la partición inicial `complete_df`. Si transformamos las variables `complete_df$target` y `train.df$target` en un objeto de tipo `table` podremos usar la función `prop.table()` para ver las proporciones de cada clase y compararlas.

```
prop.table(table(complete_df$target))
```

```
##
##      No      Yes
## 0.5703402 0.4296598
```

```
prop.table(table(train.df$target))
```

```
##
##      No      Yes
## 0.5703402 0.4296598
```

La distribución de la variable a predecir no ha cambiado.

Selección de variables

Cuando se entrena un modelo de clasificación supervisada, es importante incluir como variables únicamente aquellas variables que están realmente relacionadas con la variable a predecir, ya que son estas las que contienen información útil para el modelo. Incluir un exceso de variables suele conllevar una reducción de la capacidad de predicción de los modelos. En este punto, es donde la selección de variables para puede suponer la diferencia entre un modelo normal y uno muy bueno.

`caret` posee una colección de herramientas internas para la selección de variables. Estas herramientas contienen sus propias estrategias de selección de variables. Por este motivo los modelos de `caret` como *random forest*, *lasso* o *boosting* son tan versátiles.

Muchos de los modelos a los que se puede acceder mediante la función `train()` de `caret` producen ecuaciones de predicción que no necesariamente utilizan todas las variables, ya que, estos modelos tienen una selección de variables incorporada. El uso de estos modelos con selección de variables incorporada siempre será más eficiente que otras estrategias de selección de variables, como los métodos *wrapper* y los métodos de filtrado.

Suponiendo que la selección de variables incorporada en los modelos de `caret` es la mejor para cada modelo, para este ejercicio escogemos y entrenaremos en la siguiente sección los siguientes algoritmos:

- **glmnet**. Generalized Linear Model with Stepwise Feature Selection.
- **rf**. Random Forest.
- **gbm**. Stochastic Gradient Boosting.

Todos los algoritmos con selección de variables incorporada se encuentran disponibles en el siguiente [enlace](#).

Sabemos que hacer una buena selección de variables puede suponer la diferencia entre un modelo normal y uno muy bueno, pero ninguno de los algoritmos que se han seleccionado se utilizan específicamente en minería de textos. Así que, también entrenaremos el modelo **svmRadial** de **caret**. **svmRadial** es un algoritmo de clasificación supervisada utilizado en minería de textos. La finalidad de esta elección es poner de manifiesto que esto también puede ser un elemento muy relevante para nuestro ejercicio.

Con el modelo **svmRadial** de **caret** no podemos incorporar la selección de variables usando la función **train()**. En este caso tendríamos que utilizar alguno de los métodos de selección de variables que nos ofrece **caret**, como el denominado *recursive feature elimination* con la función **rfe()**, mediante un *algoritmo genético* con la función **gafs.default()**, etc. Pero no utilizaremos ningún método de selección de variables, porque queremos probar si **svmRadial** es mejor clasificador en comparación con los modelos que sí incorporan la selección de variables mediante la función **train()**.

Más información sobre las funciones de **caret** **rfe()** i **gafs.default()**:

- <https://www.rdocumentation.org/packages/caret/versions/6.0-86/topics/rfe>
- <https://www.rdocumentation.org/packages/caret/versions/6.0-86/topics/gafs.default>

Entrenamiento de modelos

En esta sección se entrenan los diferentes modelos elegidos de clasificación supervisada con el objetivo de compararlos e identificar el que mejor resultado obtiene prediciendo. Todos estos modelos incorporados en el paquete **caret** se entrenan con la función **train()**. Entre los argumentos de esta función destacan:

- **method**. El nombre del algoritmo que se desea emplear ([modelos disponibles](#)).
- **metric**. La métrica que se usa para evaluar la capacidad predictiva del modelo. Aunque en la competición de Kaggle en la que se basa nuestro ejercicio se espera **F1** como métrica de evaluación, para cuantificar como de bueno son nuestros modelos utilizaremos la métrica de evaluación **Accuracy**. Porqué la distribución de la variable clase no presenta un problema notable de desbalanceo de clases y cuando la distribución de la variable clase está bastante equilibrada se considera una buena medida de evaluación.

La métrica de evaluación **Accuracy** representa el porcentaje de observaciones correctamente clasificadas respecto al total de predicciones.

- **trControl**. Especificaciones adicionales sobre la forma de llevar a cabo el entrenamiento del modelo.

Para especificar el tipo de validación que se usará durante el entrenamiento, se crea un control de entrenamiento mediante la función **trainControl()**, donde se pasa al argumento **trControl** de la función **train()**. Para nuestro ejercicio, el tipo de validación que se desarrolla es una **repeated K-Fold Cross-Validation**.

Con la función **createMultiFolds()** repartimos las observaciones del conjunto de datos de train en 5 folds (conjuntos) del mismo tamaño ($k=5$), y los repetiremos 3 veces ($times=3$). En total tendremos $5 * 3 = 15$ folds en el objeto **cv.folds**.

Veremos que durante el entrenamiento de los modelos se usan semillas con **set.seed(123)**. Las semillas son necesarias si se quiere asegurar la reproducibilidad de los resultados finales, ya que la validación cruzada implica selección aleatoria.

```
set.seed(123)
cv.folds <- createMultiFolds(train.df$target, k=5, times=3)
cv.cntrl <- trainControl(method="repeatedcv", number=5,
                        index=cv.folds, allowParallel=TRUE,
                        returnResamp="final",
                        verboseIter=FALSE)
```


El argumento **method** de la función **trainControl()** hace posible utilizar distintos tipos de validación. La **repeated K-Fold Cross-Validation** se denomina “repeatecv”. Además de fijar el tipo de validación, la función **trainControl()** permite fijar multitud de argumentos.

A continuación, describimos los argumentos que se han fijado:

- **number**. El número de folds.
Tiene que coincidir con el argumento k de la función **createMultiFolds()**.
- **index**. Lista de los elementos de (**cv.folds**). Cada elemento de la lista es un vector que contiene el fold correspondiente a cada repetición.

```
str(cv.folds)
```

```
## List of 15
## $ Fold1.Rep1: int [1:6090] 1 2 3 4 5 6 7 8 9 11 ...
## $ Fold2.Rep1: int [1:6090] 1 2 3 4 5 7 8 9 10 11 ...
## $ Fold3.Rep1: int [1:6090] 1 2 3 5 6 7 8 9 10 11 ...
## $ Fold4.Rep1: int [1:6091] 2 3 4 6 10 12 13 14 15 16 ...
## $ Fold5.Rep1: int [1:6091] 1 4 5 6 7 8 9 10 11 12 ...
## $ Fold1.Rep2: int [1:6090] 1 4 5 6 8 9 10 11 12 13 ...
## $ Fold2.Rep2: int [1:6090] 2 3 4 5 6 7 8 9 11 12 ...
## $ Fold3.Rep2: int [1:6090] 1 2 3 5 6 7 10 11 12 14 ...
## $ Fold4.Rep2: int [1:6091] 1 2 3 4 5 6 7 8 9 10 ...
## $ Fold5.Rep2: int [1:6091] 1 2 3 4 7 8 9 10 11 12 ...
## $ Fold1.Rep3: int [1:6091] 1 2 3 6 7 8 9 10 11 12 ...
## $ Fold2.Rep3: int [1:6090] 1 2 4 5 6 7 9 10 11 13 ...
## $ Fold3.Rep3: int [1:6090] 1 3 4 5 8 9 10 11 12 13 ...
## $ Fold4.Rep3: int [1:6090] 2 3 4 5 6 7 8 12 13 14 ...
## $ Fold5.Rep3: int [1:6091] 1 2 3 4 5 6 7 8 9 10 ...
```

- **allowParallel**. Habilita la computación paralela.
- **returnResamp**. Indica la cantidad de métricas de evaluación que se deben guardar por cada repetición.
Se especifica que se almacene la información “final” de los modelos generados en todas las repeticiones.
- **verboseIter**. Para mostrar o no el registro del entrenamiento. En nuestro caso no se mostrará.

Muchos modelos contienen parámetros que no pueden aprenderse a partir de los datos de entrenamiento y, por lo tanto, deben de ser establecidos por nosotros. A estos se les conoce como *hiperparámetros*, argumento **tuning** de **caret**. Con la función **expand.grid()** definiremos los valores de los parámetros a tunear

Los resultados de un modelo pueden depender en gran medida del valor que tomen estos *hiperparámetros*, sin embargo, no se pueden conocer de antemano cuáles son los más adecuados. La forma más común de encontrarlos es probando diferentes posibilidades. Para ello se utiliza el argumento de **caret** **tuneGrid** como argumento de la función **train()**. Si no se tiene ninguna idea de qué *hiperparámetros* pueden ser los más adecuados, lo mejor es utilizar la aleatoriedad. **caret** también incorpora esta opción mediante el argumento **tuneLength**, también se lo pasamos a la función **train()**.

Durante el entrenamiento de los modelos usaremos **tuneGrid** y **tuneLength** según nuestro criterio. El uso de **tuneGrid** será mayor, ya que, de esta forma, se evita que se generen automáticamente valores que no tienen sentido o que disparan el tiempo de computación necesario.

k-Fold-Cross-Validation. Las observaciones de entrenamiento se reparten en k folds (conjuntos) del mismo tamaño. El modelo se ajusta con todas las observaciones excepto las del primer fold y se evalúa prediciendo las observaciones del fold que ha quedado excluido, obteniendo así la primera métrica. El proceso se repite k veces, excluyendo un fold distinto en cada iteración. Al final, se generan k valores de la métrica, que se agregan (normalmente con la media y la desviación típica) generando la estimación final de validación.

repeated k-Fold-Cross-Validation. Es exactamente igual al método k-Fold-Cross-Validation pero repitiendo el proceso completo n veces. Por ejemplo, 10-Fold-Cross-Validation con 5 repeticiones implica a un total de 50 iteraciones, pero no equivale a un 50-Fold-Cross-Validation.

A lo largo del proceso de entrenamiento de los modelos es donde más destacarán las funcionalidades ofrecidas por **caret**, permitiendo emplear la misma sintaxis para ajustar, optimizar, evaluar y predecir un amplio abanico de modelos variando únicamente el nombre del algoritmo. Aunque **caret** permite todo esto con apenas unas pocas líneas de código son muchos los argumentos que pueden ser adaptados, cada uno con múltiples posibilidades. Con el objetivo de exponer mejor cada una de las opciones, en lugar de crear directamente un modelo final, se entrenarán los modelos que hemos seleccionado en la sección de selección de variables.

Regresión Logística

Información detallada sobre regresión logística en [Regresión logística simple y múltiple](#).

Entrenamos el modelo **glmnet** de **caret**. Este algoritmo no tiene ningún *hiperparámetro*, por lo tanto, usamos **tuneLength** igual a 15.

```

cl <- makePSOCKcluster(6, setup_strategy="sequential")
registerDoParallel(cl)

set.seed(123)
model_glmnet <- train(target ~ ., data=train.df,
                      method="glmnet",
                      metric="Accuracy",
                      trControl=cv.cntrl,
                      tuneLength=15)

model_glmnet

## glmnet
##
## 7613 samples
## 150 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling:
## Summary of sample sizes: 6090, 6090, 6090, 6091, 6091, 6090, ...
## Resampling results across tuning parameters:
##
##  alpha  lambda      Accuracy  Kappa
##  0.10   0.0001850059  0.7648316  0.5085381
##  0.10   0.0018500586  0.7650941  0.5087571
##  0.10   0.0185005864  0.7635180  0.5023833
##  0.55   0.0001850059  0.7648316  0.5084863
##  0.55   0.0018500586  0.7650943  0.5081484
##  0.55   0.0185005864  0.7487187  0.4642323
##  1.00   0.0001850059  0.7649630  0.5087688
##  1.00   0.0018500586  0.7639123  0.5051295
##  1.00   0.0185005864  0.7211778  0.3970376
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were alpha = 0.55 and lambda = 0.001850059.

```

```
stopCluster(c1)
```

El valor de *Accuracy* promedio estimado mediante validación cruzada repetida es de 0.7648316, el modelo predice correctamente el 76.5% de las veces.

Random Forest

Información detallada sobre Random Forest en [Árboles de predicción](#).

Entrenamos el modelo **rf** de **caret**. Este algoritmo tiene un *hiperparámetro*:

- **mtry**. Número de valores seleccionados aleatoriamente en cada árbol. En este caso hemos escogido: 3, 4, 5 y 7.

```
c1 <- makePSOCKcluster(6, setup_strategy="sequential")
registerDoParallel(c1)
```

```
set.seed(123)
cv.grid <- expand.grid(mtry=c(3, 4, 5, 7))
model_rf <- train(target ~ ., data=train.df,
                  method="rf",
                  metric="Accuracy",
                  trControl=cv.cntrl,
                  tuneGrid=cv.grid)
```

```
model_rf
```

```
## Random Forest
##
## 7613 samples
## 150 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling:
## Summary of sample sizes: 6090, 6090, 6090, 6091, 6091, 6090, ...
## Resampling results across tuning parameters:
##
##  mtry  Accuracy  Kappa
##  3     0.7618968 0.5068252
##  4     0.7623348 0.5078468
##  5     0.7608022 0.5051518
##  7     0.7587883 0.5010024
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 4.
```

```
stopCluster(c1)
```

Empleando un modelo Random Forest con $mtry = 4$, se consigue un valor de *Accuracy* promedio de validación igual a 0.7623348, el modelo predice correctamente el 76.2% de las veces.

Gradient Boosting

Información detallada sobre boosting [Árboles de predicción](#).

Entrenamos el modelo **gbm**. El modelo **gbm** de **caret** emplea el paquete **gbm** que tendremos que cargar antes de entrenar el modelo. Este algoritmo tiene 4 *hiperparámetros*:

- **n.trees**. Número de iteraciones del algoritmo de boosting. Cuanto mayor es este valor, más se reduce el error de entrenamiento, pero mayor es el coste computacional.
- **interaction.depth**. El número total de divisiones que tiene el árbol. Emplear árboles que tienen entre 1 y 6 nodos suele dar buenos resultados.
- **shrinkage**. Controla la complejidad del modelo. Es preferible mejorar un modelo mediante muchos pasos pequeños que mediante unos pocos pasos muy grandes. Por esta razón, se emplea un valor de **shrinkage** tan pequeño como sea posible, teniendo en cuenta que, cuanto menor sea, mayor el número de iteraciones necesarias.
- **n.minobsinnode**. Número mínimo de observaciones que debe tener cada nodo del árbol para poder ser dividido. Al igual que **interaction.depth**, permite controlar la complejidad del modelo.

Además de los *hiperparámetros*, este modelo nos permite controlar otros dos valores más a tener en cuenta. Estos valores son:

- **distribution**. Determina la función de coste (*loss function*). Algunas de las más utilizadas son: *gaussian* (*squared loss*) para regresión, *bernoulli* para clasificaciones binarias, *multinomial* para clasificaciones con más de dos clases y *adaboost* también para clasificaciones binarias, ya que emplea la función exponencial del algoritmo original [AdaBoost](#).
- **bag.fraction** (subsampling fraction). Fracción de observaciones del conjunto de datos train seleccionadas de forma aleatoria. Si su valor es 1, se emplea el algoritmo de *Gradient Boosting*, si es menor que 1, se emplea *Stochastic Gradient Boosting*. Por defecto su valor es 0.5.

En este caso, solo añadimos de más el valor de **distribution**. Como nuestro ejercicio se basa en un problema de clasificación binaria elegimos *adaboost*.

```
library(gbm)

cl <- makePSOCKcluster(6, setup_strategy="sequential")
registerDoParallel(cl)

set.seed(123)
cv.grid <- expand.grid(interaction.depth=c(1, 2),
                      n.trees=100,
                      shrinkage=c(0.001, 0.01, 0.1),
                      n.minobsinnode=c(2, 5, 15))

model_gbm <- train(target ~ ., data=train.df,
                  method="gbm",
                  metric="Accuracy",
                  trControl=cv.cntrl,
                  tuneGrid=cv.grid,
                  distribution="adaboost",
                  verbose=FALSE)

model_gbm
```

```
## Stochastic Gradient Boosting
##
## 7613 samples
## 150 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling:
## Summary of sample sizes: 6090, 6090, 6090, 6091, 6091, 6090, ...
## Resampling results across tuning parameters:
##
## shrinkage interaction.depth n.minobsinnode Accuracy Kappa
## 0.001      1                2              0.5703402 0.0000000
## 0.001      1                5              0.5703402 0.0000000
## 0.001      1               15              0.5703402 0.0000000
## 0.001      2                2              0.5703402 0.0000000
## 0.001      2                5              0.5703402 0.0000000
## 0.001      2               15              0.5703402 0.0000000
## 0.010      1                2              0.6553692 0.2397333
## 0.010      1                5              0.6548878 0.2383772
## 0.010      1               15              0.6544059 0.2374328
## 0.010      2                2              0.6946872 0.3400736
## 0.010      2                5              0.6931547 0.3366365
## 0.010      2               15              0.6932867 0.3367455
## 0.100      1                2              0.7244616 0.4220042
## 0.100      1                5              0.7230157 0.4197413
## 0.100      1               15              0.7235418 0.4199881
## 0.100      2                2              0.7446022 0.4675017
## 0.100      2                5              0.7438146 0.4658317
## 0.100      2               15              0.7421065 0.4621889
##
## Tuning parameter 'n.trees' was held constant at a value of 100
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 100, interaction.depth =
## 2, shrinkage = 0.1 and n.minobsinnode = 2.
```

```
stopCluster(cl)
```

Empleando un modelo Boosting con `n.trees = 100`, `interaction.depth = 2`, `shrinkage = 0.1` y `n.minobsinnode = 2` se consigue un valor de *Accuracy* promedio de validación igual a 0.7446022, el modelo predice correctamente el 74.4% de las veces.

SVM

Información detallada sobre SVM en [Support Vector Machines](#).

Entrenamos el modelo **svmRadial** de **caret**. Este algoritmo tiene 2 *hiperparámetros*:

- **sigma**. Coeficiente del kernel radial.
- **C**. Penalización por violaciones del margen del hiperplano.

```
cl <- makePSOCKcluster(6, setup_strategy="sequential")
registerDoParallel(cl)
```

```

set.seed(123)
cv.grid <- expand.grid(sigma=c(0.001, 0.01, 0.1, 0.5, 1),
                      C=c(1, 20, 50, 100))

model_svm <- train(target ~ ., data=train.df,
                  method="svmRadial",
                  metric="Accuracy",
                  trControl=cv.cntrol,
                  tuneLength=cv.grid)

model_svm

## Support Vector Machines with Radial Basis Function Kernel
##
## 7613 samples
## 150 predictor
## 2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling:
## Summary of sample sizes: 6090, 6090, 6090, 6091, 6091, 6090, ...
## Resampling results across tuning parameters:
##
##  C      Accuracy  Kappa
##  0.25  0.7615472  0.5008012
##  0.50  0.7678960  0.5139026
##  1.00  0.7701726  0.5185027
##
## Tuning parameter 'sigma' was held constant at a value of 0.006839611
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.006839611 and C = 1.

stopCluster(cl)

```

Empleando un modelo SVM con $\sigma = 0.006839611$ y $C = 1$, se consigue un valor de *Accuracy* promedio de validación igual a 0.7701726, el modelo predice correctamente el 77% de las veces.

Evaluación de modelos mediante resampling

Una vez que se han entrenado los distintos modelos, se tiene que identificar cuál de ellos consigue mejores resultados para el problema en cuestión. Con los datos disponibles, existen dos formas de comparar los modelos. Si bien las dos no tienen por qué dar los mismos resultados, son complementarias a la hora de tomar una decisión final. Las dos formas de comparar y evaluar los distintos modelos entrenados son: usando las métricas de evaluación obtenidas y/o con un **test estadístico** empleando la función **extractPrediction()** de **caret**.

El test estadístico se usaría para determinar si un método es mejor que otro, cuando no sea suficiente la comparación de los valores mínimos (o máximos dependiendo de la métrica) que ha conseguido cada uno. Si ese fuera el caso, con el uso de los promedios podríamos determinar si existen evidencias suficientes de superioridad entre los modelos.

La función **resamples()** nos permitirá extraer, dado los modelos entrenados con **train()**, las métricas de evaluación obtenidas para cada repetición del proceso de validación. Es importante tener en cuenta que la función **resamples()** recupera todos los resultados de un entrenamiento, por lo que, si no se ha especificado en el control del entrenamiento, en la función **trainControl()** el argumento **returnResamp = "final"**, se devolverán todas las métricas, no solo las del modelo final.

```
models <- list(logistic=model_glmnet,
               RF=model_rf,
               boosting=model_gbm,
               SVMRadial=model_svm)
```

```
resamps <- resamples(models)
summary(resamps)
```

```
##
## Call:
## summary.resamples(object = resamps)
##
## Models: logistic, RF, boosting, SVMRadial
## Number of resamples: 15
##
## Accuracy
##           Min.   1st Qu.   Median     Mean   3rd Qu.     Max. NA's
## logistic 0.7424442 0.7592784 0.7667543 0.7650943 0.7728168 0.7820092    0
## RF       0.7399869 0.7545992 0.7608410 0.7623348 0.7669074 0.7944846    0
## boosting 0.7170059 0.7398160 0.7439265 0.7446022 0.7488510 0.7754432    0
## SVMRadial 0.7491793 0.7641261 0.7667543 0.7701726 0.7731451 0.7990808    0
##
## Kappa
##           Min.   1st Qu.   Median     Mean   3rd Qu.     Max. NA's
## logistic 0.4573146 0.4949223 0.5129929 0.5081484 0.5257726 0.5456174    0
## RF       0.4637190 0.4919726 0.5020328 0.5078468 0.5179187 0.5780024    0
## boosting 0.4089761 0.4589374 0.4640576 0.4675017 0.4759897 0.5330039    0
## SVMRadial 0.4748475 0.5041853 0.5092062 0.5185027 0.5261318 0.5834342    0
```

Observamos que junto con la estimación del porcentaje de bien clasificados (Accuracy), se nos muestra el valor de la métrica **Kappa**, el cual es una medida que compara el Accuracy observado respecto al Accuracy esperado (de una predicción al azar). Esto es, cuanto mejor estima un modelo respecto a otro que predice al azar. Un ameno ejemplo para entenderla mejor se encuentra en este [enlace](#).

Kappa o Cohen's Kappa es el valor de Accuracy normalizado respecto del porcentaje de acierto esperado por azar. A diferencia de Accuracy, cuyo rango de valores puede ser [0, 1], el de Kappa es [-1, 1]. En problemas con clases desbalanceadas, donde el grupo mayoritario supera por mucho al grupo minoritario, Kappa es más útil porque evita caer en la ilusión de creer que un modelo es bueno cuando realmente solo supera por poco lo esperado por azar.

Todos los modelos tienen un promedio de aciertos en la predicción superior al nivel basal, Accuracy superior a 0.57 (distribución de la clase mayoritaria, ver Figura 1). El modelo **SVMRadial** es el modelo que consigue el valor de Accuracy promedio más alto, seguido muy de cerca por el modelo **logistic**. Para determinar si las diferencias entre ellos son significativas, se recurre al test estadístico.

El test estadístico chequeará las diferencias de los valores de la métrica Accuracy, entre todos los pares de modelos y interpretará mediante el *p-value* asociado, si estas diferencias son **estadísticamente significativas** o no, usando el clásico umbral de 0.05. Si las diferencias de los modelos **SVMRadial** y **logistic** son estadísticamente diferentes, el mejor modelo será **SVMRadial** porque su valor de Accuracy es mayor.

El test estadístico en **R** que se ha elegido es el **Wilcoxon signed-rank test**. Para lanzar-lo usaremos la función **pairwise.wilcox.test()** del paquete **stats**.

Wilcoxon signed-rank test

Para desarrollar el test, primero tenemos que transformar los valores de las métricas devueltos por la función `resamples()`, para separar cada modelo con sus métricas. Para ello, se usan las funciones:

- `gather()` para juntar las columnas en pares modelo-valor.
- `separate()` para separar la columna modelo en las columnas modelo-métrica.

Para mejorar la separación, se usan las funciones:

- `group_by()` para agrupar los resultados modelo-métrica.
- `summarise()` para resumir el promedio de los valores de las métricas.
- `spread()` para distribuir los pares métrica-media en varias columnas.
- `arrange()` para ordenar los resultados de forma descendente.

Todas estas funciones se pueden encontrar en los paquetes **tidyr** y **dplyr** del paquete **tidyverse**. Y se recomienda visualizar el valor de la variable `metricas_resamples` durante las diferentes transformaciones para entender que hace cada función.

```
metricas_resamples <- resamps$values %>%  
  gather(key = "modelo", value = "valor", -Resample) %>%  
  separate(col = "modelo", into = c("modelo", "metrica"),  
           sep = "~", remove = TRUE)  
  
metricas_resamples %>%  
  group_by(modelo, metrica) %>%  
  summarise(media = mean(valor)) %>%  
  spread(key = metrica, value = media) %>%  
  arrange(desc(Accuracy))
```

Una vez tenemos los valores de las métricas separados por modelo, seleccionamos solo los valores de la métrica Accuracy. El test comparará estos valores entre todos los pares de modelos y calculará el *p-value* resultante de la comparación. La función `pairwise.wilcox.test()` es la que calculará estas comparaciones múltiples y su *p-values* resultantes. Esta función contiene los siguientes argumentos:

- **x**. Vector que contiene todos los valores de la métrica *Accuracy*.
- **g**. Vector en *x* agrupado por modelo.
- **paired**. Indicador de si el test será pareado o no. En nuestro caso, sí que lo es.
- **p.adjust.method**. Método para ajustar los *p-values*. En este caso se ajusta con el método de [Holm's](#), que es de los más utilizados para este tipo de test estadístico.

```
metricas_accuracy <- metricas_resamples %>% filter(metrica == "Accuracy")  
comparaciones <- pairwise.wilcox.test(x = metricas_accuracy$valor,  
                                     g = metricas_accuracy$modelo,  
                                     paired = TRUE,  
                                     p.adjust.method = "holm")
```

El *p-value* ayuda a diferenciar resultados que son producto del azar en el muestreo, de resultados que son estadísticamente significativos.

Se almacenan los *p-values* en forma de dataframe y aplicamos otra serie de transformaciones sobre este para mejorar su interpretación. Además de las funciones que ya hemos usado anteriormente como `gather()` y `arrange()`, añadimos otras dos:

- `rownames_to_column()` que nos agrega una columna al inicio del dataframe.
- `na.omit()` que excluye los valores perdidos, en el caso de que existieran en el dataframe.

```
comparaciones <- comparaciones$p.value %>%
  as.data.frame() %>%
  rownames_to_column(var = "modeloA") %>%
  gather(key = "modeloB", value = "p_value", -modeloA) %>%
  na.omit() %>%
  arrange(modeloA)
```

```
comparaciones
```

```
##      modeloA modeloB      p_value
## 1  logistic boosting 0.0007324219
## 2          RF boosting 0.0036056278
## 3          RF logistic 0.3894042969
## 4 SVMRadial boosting 0.0036056278
## 5 SVMRadial logistic 0.1801641152
## 6 SVMRadial          RF 0.0036056278
```

Acorde a las comparaciones por pares, existen evidencias suficientes para considerar que la capacidad predictiva de los modelos **SVMRadial** y **logistic** es distinta. Ya que el *p-value* es superior 0.05, el umbral fijado inicialmente. Lo mismo ocurre entre los modelos **Random Forest** y **logistic**, pero no interesa el resultado, porque el modelo **Random Forest** ya ha sido descartado para hacer la predicción.

También es recomendable visualizar el valor de la variable **comparaciones** durante las diferentes transformaciones para entender que hace cada función.

Conclusión evaluación de modelos

Hemos podido ver que el modelo basado en **SVM** es el que mejores resultados obtiene, y el modelo basado en **Boosting** es el que peores resultados obtiene, acorde a la métrica Accuracy.

Los modelos basados en **Random Forest** y **Regresión Logística** consiguen valores un poco similares con los resultados de la validación, sin embargo, acorde a los resultados del test estadístico, el modelo basado en **Regresión Logística** es significativamente inferior a **SVMRadial**. Recordemos que en la comparación por pares entre los modelos **SVMRadial** y **logistic** existen evidencias suficientes para considerar que la capacidad predictiva de los modelos es distinta ($p\text{-value} = 0.1801641152 > 0.05$).

Llegados a este punto, sabemos que la predicción sobre el conjunto de datos de test la haremos con el modelo **SVMRadial**. Pero antes de esa predicción, vamos a realizar una predicción, con la función **predict()** de **caret**, sobre el conjunto de datos de train, y calcularemos el valor de la métrica **F1**, la métrica que usa la competición de Kaggle escogida para nuestro ejercicio.

Con eso queremos averiguar si tendremos un buen resultado en la competición de Kaggle. También nos servirá para ver si la predicción sobre el conjunto de test contendrá algunos errores, siendo la predicción peor de la esperada en comparación a la predicción sobre el conjunto de datos de train (valor de **F1** mayor); o que no contendrá errores porque la predicción sobre el conjunto de test será mejor de la esperada en comparación a la predicción sobre el conjunto de datos de train (valor de **F1** menor). Cuanto más próximo a 1 se encuentre el valor de **F1**, mejor resultado obtendremos en Kaggle.

Así pues, empezaremos con predecir sobre el conjunto de datos de train usando la función **predict()**. Nos quedamos con la opción `probs = "raw"` para poder calcular la matriz de confusión.

Con los resultados de la predicción calcularemos la matriz de confusión y finalmente calcularemos el valor de **F1** a partir de la matriz de confusión, porque la matriz de confusión nos devolverá los valores necesarios para el cálculo del valor de **F1**: true positives (TP), false positives (FP) y false negatives (FN). Ir a la sección análisis exploratorio de los datos para ver cómo se calcula.


```
pre_pred <- predict(model_svm, train.df, type="raw")
confusionMatrix(pre_pred, train.df$target)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   No  Yes
##           No 4092 963
##           Yes 250 2308
##
##           Accuracy : 0.8407
##           95% CI : (0.8323, 0.8488)
##           No Information Rate : 0.5703
##           P-Value [Acc > NIR] : < 2.2e-16
##
##           Kappa : 0.6659
##
## Mcnemar's Test P-Value : < 2.2e-16
##
##           Sensitivity : 0.9424
##           Specificity : 0.7056
##           Pos Pred Value : 0.8095
##           Neg Pred Value : 0.9023
##           Prevalence : 0.5703
##           Detection Rate : 0.5375
##           Detection Prevalence : 0.6640
##           Balanced Accuracy : 0.8240
##
##           'Positive' Class : No
##
```

```
true_positives <- 2308
false_positives <- 250
false_negatives <- 963

precision <- true_positives / (true_positives + false_positives)
recall <- true_positives / (true_positives + false_negatives)

F1 <- 2 * (precision * recall) / (precision + recall)
F1 <- round(F1, digits = 4)

F1
```

```
## [1] 0.7919
```

El valor de **Accuracy** es 0.8407 y el valor de **F1** es 0.7919. No son valores muy buenos, pero tampoco son muy malos. El error de entrenamiento (*training error*) es $1 - 0.8407 = 0.1593$. Este error se corresponde con el error que comete el modelo al predecir, un 15.93% en este caso. En la sección posterior veremos si estos valores mejoran o no en la predicción sobre el conjunto de datos de test.

Predicción

Una vez que el modelo ha sido ajustado, se pueden predecir nuevos datos, sobre el conjunto de datos de test. Usaremos la función **predict()** de **caret**.

Los argumentos de la función **predict()** son:

- **models**: el modelo o lista de modelos creados mediante la función **train()**.
- **newdata**: el dataframe con nuevas observaciones (conjunto de datos de test).
- **type**: tipo de predicción. En modelos de clasificación puede ser “raw” para obtener directamente la clase predicha o “prob” para obtener la probabilidad de cada clase. Algunos algoritmos no incorporan de forma nativa un cálculo de probabilidades, para asegurar que textbfcaret las calcula, se debe indicar en el control de entrenamiento `classProbs = TRUE`.

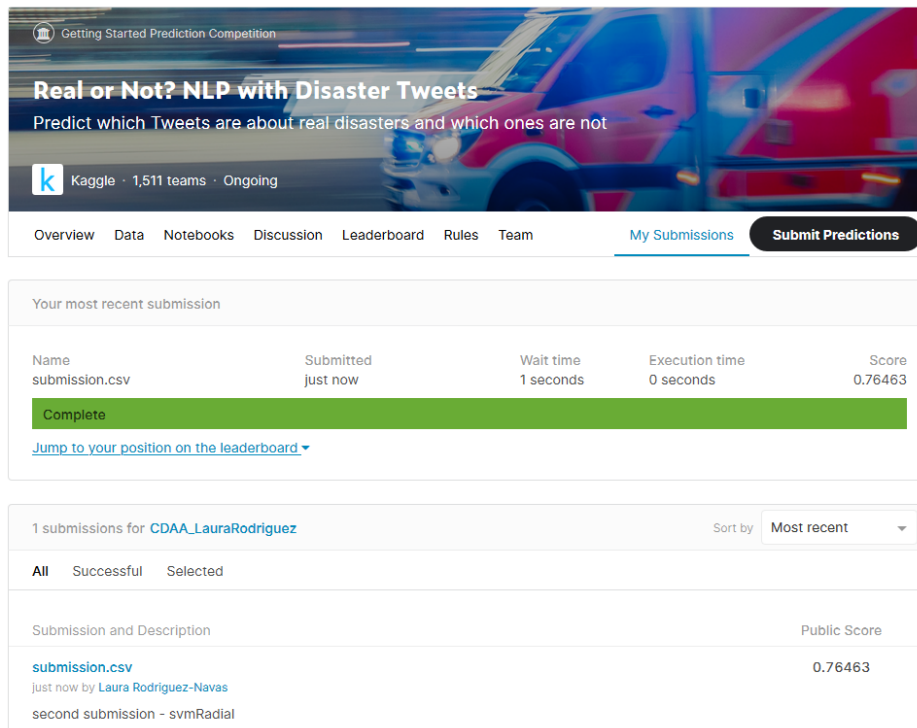
```
pred_svm <- predict(model_svm, test.df, type="raw")
```

Finalmente preparamos el archivo que subiremos a la plataforma de Kaggle con los resultados de nuestra predicción. Veremos las 7 primeras líneas del archivo.

```
submission <- read.csv("sample_submission.csv")
submission$target <- ifelse(pred_svm=="No", 0, 1)
write.csv(submission, "submission.csv", row.names=FALSE)
head(submission, 10)
```

```
##      id target
## 1     0      0
## 2     2      1
## 3     3      1
## 4     9      0
## 5    11      1
## 6    12      1
## 7    21      0
```

Resultado en Kaggle



Getting Started Prediction Competition

Real or Not? NLP with Disaster Tweets

Predict which Tweets are about real disasters and which ones are not

Kaggle · 1,511 teams · Ongoing

Overview Data Notebooks Discussion Leaderboard Rules Team My Submissions Submit Predictions

Your most recent submission

| Name | Submitted | Wait time | Execution time | Score |
|----------------|-----------|-----------|----------------|---------|
| submission.csv | just now | 1 seconds | 0 seconds | 0.76463 |

Complete

[Jump to your position on the leaderboard](#)

1 submissions for CDAA_LauraRodriguez

Sort by Most recent

All Successful Selected

| Submission and Description | Public Score |
|--|--------------|
| submission.csv just now by Laura Rodriguez-Navas second submission - svmRadial | 0.76463 |

Conclusiones

En este documento se revisa cómo importar y preparar documentos de texto para realizar distintas operaciones de minería de textos con ellos, tales como obtener palabras frecuentes, asociaciones de palabras y análisis de sentimientos.

Si comparamos nuestro resultado con los de los otros usuarios de la competición de Kaggle, vemos que el modelo creado no es muy bueno. La predicción ha sido peor de la esperada. Aunque esa no era la finalidad de este ejercicio, esto nos demuestra que el ejercicio se podría mejorar en algunos aspectos. Pero sí que hemos podido comprobar una de las hipótesis que se ha planteado durante este ejercicio. Un modelo sin selección de variables incorporada puede predecir mejor que un modelo que sí que incorpora selección de variables, porque el modelo es un modelo específico de clasificación supervisada en minería de textos? La respuesta es sí. Es una conclusión muy interesante que se extrae de este ejercicio.

Personalmente escogí esta competición ya que nunca había combinado la clasificación supervisada con Procesamiento del Lenguaje Natural (PLN). El Procesamiento del Lenguaje Natural dentro de la Inteligencia Artificial es un campo que encuentro muy interesante.

El problema más importante para la realización de este ejercicio ha sido el coste computacional. Llegado el punto en que la ejecución de todo el flujo de trabajo tardaba unas cuantas horas en finalizar, se pensó en utilizar la computación paralela. Fue una buena idea, se redujo drásticamente el coste computacional y además aprendí a hacerlo en **R**.

Durante la asignatura de Procesamiento del Lenguaje Natural del Máster no se realiza ninguna práctica o trabajo de programación, y me pareció una buena oportunidad de aprender cómo aplicar los conocimientos adquiridos de ella con los conocimientos adquiridos durante esta asignatura.

Por supuesto la minería de textos es un área de análisis extensa. Las siguientes referencias han sido de gran ayuda para realizar este documento.

Referencias

- Machine Learning con R y caret https://www.cienciadedatos.net/documentos/41_machine_learning_con_r_y_caret
- Notebook en Kaggle de barun2104 <https://www.kaggle.com/barun2104/nlp-with-disaster-eda-dfm-svd-ensemble>
- Notebook en Kaggle de sambitmukherjee <https://www.kaggle.com/sambitmukherjee/bag-of-words-stack-knn-tree-lasso#test-set-predictions-submission>