



INFORMATIKA  
FAKULTATEA  
FACULTAD  
DE INFORMÁTICA

**Ingeniería Informática  
UPV/EHU**

Laboratorios Minería de Datos

**Mikel Molina**

## Índice

<b>Octavo laboratorio (8-9 de noviembre)</b>	<b>2</b>
<b>Noveno Laboratorio (15-16 de noviembre)</b>	<b>15</b>
<b>Décimo laboratorio (22-23 de noviembre)</b>	<b>42</b>
<b>Undécimo laboratorio (4-5 de diciembre)</b>	<b>54</b>
<b>Duodécimo laboratorio (13-14 de diciembre)</b>	<b>77</b>

## Octavo laboratorio (8-9 de noviembre)

Este laboratorio está dedicado al tema 8 de clustering, que se aleja clasificación supervisada. Ni caso a la matriz de confusión, ni accuracy, TPR... No. Nada de eso, puesto que ahora el problema no es clasificar variables, sino agruparlas en conjuntos llamados **clústeres**.

Dentro de cada clúster, todos guardan una relación entre sí, y es que se ha utilizado un criterio para agruparlos. En éste caso, se usará euclídea. Sin embargo, sea cual sea la fórmula se emplee para la clusterización, el fin debe ser el mismo: conseguir agrupar los casos que tengan la menor homogeneidad, y evitar lo contrario.

### K-Means Clustering.

Hay varios algoritmos que permiten particionar los conjunto de datos, en este caso el foco de atención será k-means. Su funcionamiento es el siguiente:

- **Paso 1:** Se escogen  $k$  puntos iniciales entre todos los casos del conjunto de datos. Éstos van a formar un clúster de un único elemento.
- **Paso 2:** Se asignan los casos al clúster cuyo *centroide* tengan más cercano. Como se ha dicho previamente, la vara de medir ésto será la distancia euclídea.
- **Paso 3:** Se calcula el *centroide* nuevo de cada clúster. Más tarde se profundizará en éste paso y lo que significa *centroide*.
- **Paso 4:** Reasignación de puntos, por clúster nuevo, habrá un nuevo centroide.
- **Paso 5:** Hasta que se llegue a un número máximo de iteraciones o la diferencia entre dos clusterizaciones consecutivas sea ínfima, se seguirá iterando.

Una vez conseguidos dichos clústeres, es común que se etiqueten. A este proceso se le denomina “class discovery”. Éste paso es útil, por ejemplo, en plataformas para recomendar productos, series, o películas: si se clasifica un perfil en un clúster concreto donde los integrantes comparten gustos, un algoritmo de recomendaciones tendrá una mayor garantía de que si un producto le ha gustado a un miembro del grupo, le gustará al resto también, resultando en un mejor “engagement”, que se traduce en más dinero.

A continuación se va a analizar “*food.arff*” [Hartigan, 1975]. No es más que una colección de datos juguete, pero sirve para entender mejor cómo funciona la clusterización en WEKA. Nada más abrir el dataset, se ve la primera gran diferencia con los anteriores laboratorios, y es que *no* hay clase deseada a predecir. Se debe a que no es el enfoque de este problema: lo que realmente interesa es agrupar los alimentos en función de sus atributos:

1. **Name:** El nombre del producto, puede tomar veintisiete valores distintos.
2. **Energy:** Atributo numérico. Se trata de la energía que dicho alimento aporta al ser humano, su usan varias formas de medirlo, pero la más común y conocida son las calorías.
3. **Protein:** Atributo numérico. Las proteínas que el alimento tiene.
4. **Fat:** Atributo numérico. Las grasas del alimento.
5. **Calcium:** Atributo numérico. La cantidad en calcio del alimento.
6. **Iron:** Atributo numérico. La cantidad en hierro del alimento.

Una vez vistos los atributos, se intuye la utilidad que tiene agrupar los casos. Desde el punto de vista de un nutricionista, si agrupásemos la comida en función de cuáles tienen más energía, proteínas, calorías, etc. estaríamos facilitándole el trabajo, ya que en el futuro podría elaborar una dieta bastante más completa a sus clientes en función de sus necesidades. Por otra parte, el encargado de una tienda podría organizar las baldas en función de esta agrupación, poniendo productos similares juntos, para aligerar el

trabajo del cliente... O todo lo contrario, para que el comprador vea todos los productos de la tienda en el proceso. A la clusterización se le puede sacar tanto provecho como ingenio tenga el usuario.

WEKA cuenta con su propia pestaña para la clusterización llamada *cluster*, que es parecida en formato a las demás pestañas que se han visto en previos laboratorios. Entre todos los métodos para clusterizar, se trabajará con el dado en clase: *SimpleKMeans*. Tiene varios parámetros, pero son cuatro los que hay que examinar:

1. **maxIterations:** Si bien está garantizado que k-means converge, hay situaciones en los que lleva un tiempo exponencial. Es decir, puede que requiera tantas iteraciones que puede que sea mejor cortar por lo sano, antes de que el coste sea demasiado grande. Para evitar este tipo de casos WEKA nos ofrece un límite de iteraciones. Por defecto se encuentra en quinientos, aunque podemos tanto bajarlo como subirlo; en este caso, al tener una colección de datos de unos meros veinte casos, este límite no debería afectar lo más mínimo.
2. **distanceFunction:** Hay varios métodos para calcular la distancia entre dos casos. Pese a que se vaya a emplear la distancia euclídea; saber que hay más funciones viene bien para ver las distintas fórmulas que se pueden usar. Algo curioso pasa, y es que si se trata de escoger otras, no va a dejar. Para visualizarlo mejor, aquí tenemos ambas fórmulas:

#### Distancia euclidiana:

$$d(a, b) = \sqrt{(b_1 - a_1)^2 + (b_2 - a_2)^2 + \dots + (b_n - a_n)^2} \quad (1)$$

#### Distancia Manhattan:

$$d_1(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|_1 = \sum_{i=1}^n |p_i - q_i|,$$

donde  $\mathbf{p} = (p_1, p_2, \dots, p_n)$  y  $\mathbf{q} = (q_1, q_2, \dots, q_n)$  son vectores.

3. **numClusters:** Aquí está la famosa K que le da el nombre al algoritmo. No es más que el número de clústeres se desean generar mediante el algoritmo. Aquí surge la gran duda: ¿cuántos clústeres deben generar? No se sabe a ciencia cierta. Quizá con dos o tres dimensiones, situaciones en los que podemos visualizar la distribución de puntos, la mejor opción es que una persona decida el hiperparámetro manualmente. Sin embargo, y como ocurre aquí, las dimensiones no van a ser pocas, por lo que lo normal es usar varias k's y quedarse con la mejor clusterización. Para estos casos, lo más común es generar un grid con distintos valores de k. A este proceso se le llama "tuning".
4. **displayStdDevs:** En la ventana more se explica bien lo que hace este parámetro, que ahora tenemos en False. Si puesto a true, mostraría la desviación estándar de toda variable numérica, es decir, en qué rango caen la mayoría de los valores de cada variable. Con las variables nominales devuelve una lista con las apariciones de cada nombre.

Antes de ejecutar el clustering, se ignorará atributo del nombre de las variables, no tiene sentido clusterizar una variable de tipo string; hay que tener en cuenta que la distancia euclídea requiere de valores numéricos. El output se encuentra en la figura 1:

1. **Number of iterations:** SimpleKMeans ha requerido de un total de cinco iteraciones para elaborar el clúster. Esto es de esperar: para un dataset pequeño, es necesario un número de iteraciones pequeño para agruparlo.
2. **Initial starting points:** Esto influye en los clústeres que se obtienen al final. Como se deciden aleatoriamente, es posible que los puntos iniciales estén cerca, por lo que los clústeres también lo estarán. Como ya se ha mencionado previamente, para paliar estos problemas se encuentra el tuneado de hiperparámetros.

3. **Missing values replaced with mean/mode:** Si hubiera habido casos en los que no se recolectó uno de sus valores, WEKA nos los habría reemplazado con la media o la moda. Para éste conjunto de datos no ha sido necesario.
4. **Final Cluster Centroids:** En el algoritmo k-means, a cada clúster le va a corresponder un centroide, que es la media de todas variables que han sido asignadas al mismo. Ésto nos va a resultar en un punto, —algo así como el baricentro—, que se usa para calcular la proximidad con respecto a los otros puntos y reasignarlos. Al añadir nuevos puntos al clúster, es necesario recalcular el valor del centroide. Se usa la siguiente fórmula:

$$C_j = \frac{1}{|S_j|} \sum_{\mathbf{x} \in S_j} \mathbf{x}$$

```

kMeans
=====
Number of iterations: 3
Within cluster sum of squared errors: 4.077107847192327

Initial starting points (random):

Cluster 0: 340,20,28,9,2.6
Cluster 1: 170,25,7,12,1.5
Cluster 2: 90,14,2,38,0.8

Missing values globally replaced with mean/mode

Final cluster centroids:
  Cluster#
Attribute  Full Data      0        1        2
              (27.0)   (8.0)   (12.0)   (7.0)
-----
Energy       207.4074   341.875   171.25   115.7143
Protein       19         18.75    22.1667   13.8571
Fat          13.4815   28.875     8.25    4.8571
Calcium      43.963    8.75     48.1667    77
Iron          2.3815   2.4375    2.3583   2.3571

Time taken to build model (full training data) : 0.01 seconds
== Model and evaluation on training set ==
Clustered Instances

0      8 ( 30%)
1      12 ( 44%)
2       7 ( 26%)

```

Figure 1: Output Resultante del Clustering.

WEKA también permite “visualizar” el clustering, se puede ver en la figura 2. Como hay más de dos dimensiones, no se va a poder visualizar el gráfico con todas las variables. Es por esto que WEKA permite ver un gráfico de los casos con respecto a dos variables que nosotros decidamos. Esta vez se escoge como *x* la energía, y como eje *y* las proteínas. También se coloreará cada variable en función del clúster al que pertenece (así evitamos establecer un eje para el clúster) en el parámetro Colour. Si bien en este gráfico las variables se han agrupado correctamente, es decir, cada caso se encuentra dentro del clúster más cercano, en otros gráficos, como el de la figura 3, el clústering se complica: los casos del primer y segundo clúster se mezclan, hay tres casos del clúster rojo que deberían pertenecer al verde... Esto es todo un dilema: al ser una agrupación de tantas dimensiones, si cambiásemos éstas podríamos

estar causando que se agrupasen mal en otro gráfico.

Finalmente, se ejecuta SimpleKMeans para dos clústeres y cuatro clústeres, para luego visualizarlo en el mismo gráfico que en la figura 2. Los resultados se encuentran en las figuras cuatro y cinco, respectivamente. En el caso de  $k = 2$  hay una agrupación clara y de hecho no tan mala. No hay ningún caso mal coloreado ni nada del estilo. El problema viene con  $k = 4$ , y es que no sale ningún caso clasificado como clúster 4; será que con más de tres clústeres es demasiado.

### **Escoge la asignación de clústeres que más te convenza, y explica por qué.**

Es nuestro turno de hacer de **evaluadores externos**, y decidir cuál es el mejor clústering de todos. Cabe destacar que ésto es lo ideal: que una persona decida entre todos los K el mejor parámetro (quizá no nosotros, sino una persona más especializada), así como se hace en el clústering jerárquico. En este caso nos lo podemos permitir porque son tan solo tres Ks las que tenemos que comparar, pero en la vida real ésto no es tan fácil, ni habrá ningún interés en agrupar un conjunto de datos de tan solo veinte instancias: se hará a mayor escala, y en estos casos tendremos que escoger entre cientos de clusterizaciones distintas, suma a esto que un experto requiere de bastante tiempo para llegar a una conclusión y... Se vuelve una tarea imposible, por lo que no hay otro remedio que recurrir a una evaluación interna o automática (Índice Silhouette, SSE...) para tomar estas decisiones, con quizás en la última etapa usando un experto para tomar la decisión final.

La que más convence es SimpleKMeans con  $k = 3$ . Para empezar, como hemos visto previamente, con cuatro grupos no estamos aportando nada; y es que el output de dicha clusterización nos indica que tan solo ha habido un caso clasificado en el cuarto clúster. Tampoco queremos con  $k = 2$ , porque como hemos visto con  $k = 3$ , es más que posible repartir los casos equitativamente en tres grupos distintos; recordemos que en la figura uno hemos acabado con ocho casos en el primer clúster, doce casos en el segundo clúster y siete casos en el tercer clúster.

### **Analiza los centroides de cada clúster.**

Los centroides de la clusterización con  $k = 3$  se encuentran en la figura 1. Ninguno de ellos es parecido al otro, con excepción del hierro, que parece ser que todos comparten un valor igual. El primer cluster tiene el mayor valor de energía, es por esto que la mayoría de variables con la misma característica se encuentra dentro de ese clúster. Entre ellos debemos destacar el alimento que se encuentra en la tercera fila del dataset “Roast Beef”, que cuenta con un 420 de energía, y superando con creces el valor medio. Para el segundo clúster tenemos un gran número de proteínas, una vez más, destacamos la variable que se encuentra en la novena fila del dataset “Beef Heart”, que cuenta con 26 de proteína. Para terminar, en el último clúster, el imperante es el calcio, y la variable de la décimo octava fila “Raw Clams” con 82 de calcio es de destacar. Ahora vamos con la que hay alguna que otra incógnita. Para ello, vamos a estar buscando variables que sean lo más equilibradas posibles, con valores cercanos a la media, entre ellas tenemos “Beef Tongue”, un alimento cuyas variables se encuentran muy cerca de la media, que ha sido agrupada en el cluster uno.

### **Encuéntrale sentido a los clústeres aprendidos.**

Ahora nosotros vamos a hacer de expertos en el tema y sacar una estructura de las agrupaciones hechas por SimpleKMeans. Si nos fijamos en el output de los centroides, se puede ver cómo cada centroide supera con creces en el valor de una variable al resto, vamos a valernos de estos para darles una etiqueta:

1. **Clúster 0:** Un gran índice calórico y de grasa. Éste tipo de alimentos se recomiendan cuando el objetivo es ganar peso, como por ejemplo en el caso de gente que tiene un índice de masa corporal inferior al que deberían tener. Esto es por lo que hemos decidido etiquetar el clúster como “Alimentos Muy Calóricos”.
2. **Clúster 1:** Índice alto de proteínas y calcio. En los casos que el objetivo sea ganar masa muscular, se recomienda comer un índice de proteínas superior al normal. Una dieta compuesta de los alimentos de este cluster se le denomina **dieta hiperproteica**. Por esto mismo vamos a etiquetar el grupo como “Alimentos Hiperproteicos”.

**3. Clúster 2:** Alimentos ricos en calcio. Son los tipos de alimentos que se recomiendan en niños y jóvenes, puesto que es la etapa en la que sus huesos más crecen, los dientes se generan, etc. También es crucial para la gente embarazada, y gente en la vejez. Vamos a llamar a ésta última partición “Alimentos Altos en Calcio”.

Ésto último que acabamos de hacer después de haber obtenido los clústeres es la verdadera utilidad del algoritmo. Es decir, lo que buscamos con la agrupación es encontrar una estructura en todo el montón de datos que se nos ha servido. Ahora que tenemos estas agrupaciones etiquetadas, cada vez que añadamos un nuevo alimento a la base de datos, sabremos de una forma intuitiva qué tipo de alimento se trata, facilitando el trabajo al nutricionista, biólogo, ...

Sin embargo, al ser un conjunto de datos tan pequeño, se están dejando algunos grupos útiles de lado. Por ejemplo, ¿dónde están los alimentos ricos en hierro? ¿Y si deseamos alimentos altos en calorías, pero bajos en grasa? Existen limitaciones.

### ¿Hay casos lejos de su clúster?

Fijándonos en *outliers*, lo que estamos haciendo es indirectamente comprobar si cada clúster tiene un SSE alto o no. Recordemos que éste índice se calcula en la **evaluación interna**, cuando deseamos obtener una métrica de nuestra partición para saber si las descartamos o no. Ésta métrica devolverá un número pequeño si la diferencia entre el centroide de un clúster y sus elementos son cercanos, por lo que si hay muchos o *outliers*, la calidad de dicho clúster será peor. En el caso de nuestra partición, si bien es cierto que algunos gráficos nos brindan algunos casos así, (sin ir más lejos, en la figura 2, hay un caso despegado de cualquier cluster a la derecha del todo), no es la norma. De hecho, por cada gráfica que vemos solo hay como máximo tres casos mal clasificados, que quizás puede que sea algo malo en un conjunto tan pequeño, pero que tan poco es decisivo.

Por no mencionar que estamos viendo los gráficos a cachos, no sería justo decidir si nuestra partición es buena basados en estos gráficos. Puede que una mejor manera sea obtener los componentes principales y ver cómo se forman los clústeres, de tal forma que los *outliers* serían más fáciles de encontrar; ocurre que en algunos gráficos algunos casos que pensamos que están alejados del conjunto, son cercanos en otros.

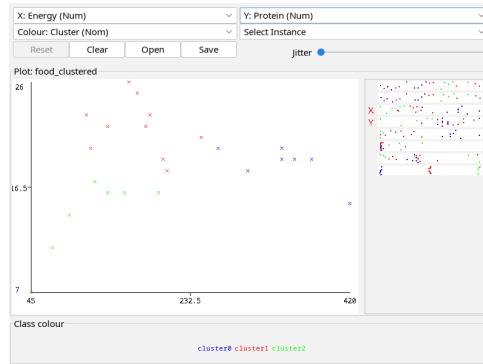


Figure 2: Clustering de la energía con respecto a las proteínas.

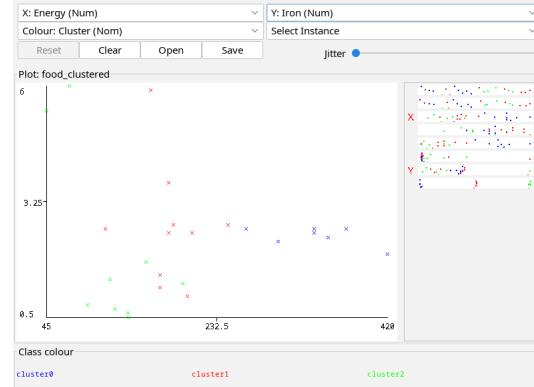


Figure 3: Clustering de la energía con respecto al hierro.

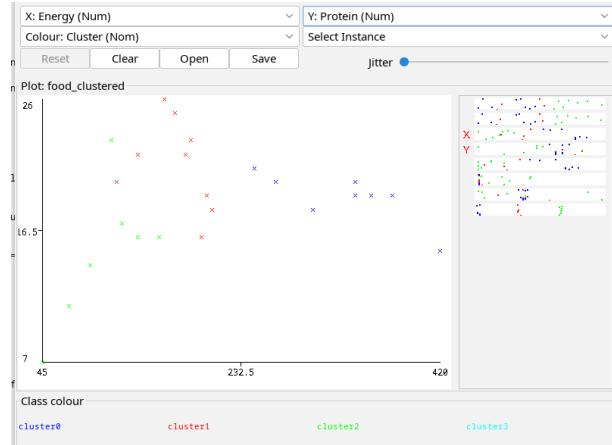


Figure 4: SimpleKMeans con  $k = 4$ .

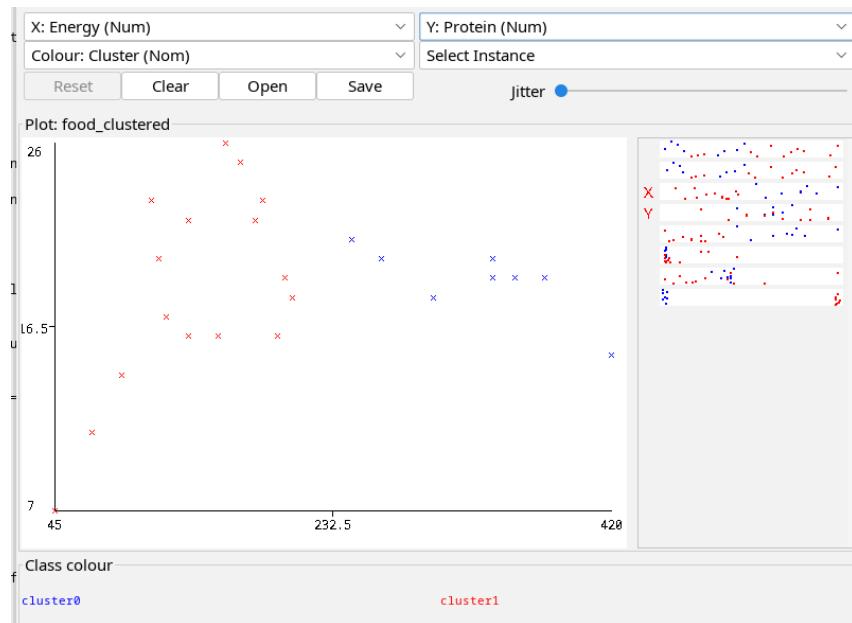


Figure 5: SimpleKMeans con  $k = 2$ .

## Comenta cada línea de código de R, aportando tu interpretación personal. Responde las preguntas.

Vamos a cargar el conjunto de datos con el que hemos estado trabajando hasta ahora. En vez de descargarnos el fichero en nuestra máquina y pasarselo por parámetro, le mandamos el enlace del directorio. Pese a que vienen por defecto, con header=TRUE, se ha declarado que la primera fila del conjunto de datos resultante no contiene datos, si no el nombre de cada columna, es decir, el nombre de cada variable. Otra opción que viene por defecto es sep=","; indica qué carácter separa dos variables entre sí, de tal forma que, por ejemplo: 500, 400 lo leerá como dos valores distintos al estar separados. Finalmente, asignamos todos estos datos a la variable food.

```
food <- read.csv(file = "http://www.sc.ehu.es/ccwbayes/docencia/md/selected-dbs/clustering/food.csv",
                 header = TRUE, sep = ",")
```

Mediante colnames se muestran por pantalla las variables del dataset, que son las siguientes:

```
colnames(food)
```

```
## [1] "Name"      "Energy"     "Protein"    "Fat"        "Calcium"   "Iron"
```

Guardamos todas las variables en foodNumeric salvo la primera columna, que es la que corresponde a las variables nominales, he ahí el nombre. Esto es el equivalente a lo que hicimos en WEKA a la hora de ignorar la misma columna.

```
foodNumeric <- food[, -1]
```

Cargamos la librería que nos permitirá particionar nuestro dataset mediante k-means.

```
library(stats)
```

Ahora vamos a llamar a la función kmeans y vamos a pasarle el conjunto de datos donde ignoramos la primera columna. A parte de esto, con centers=2 estamos estableciendo nuestro hiperparámetro k = 2, por lo que la función va a crear dos clústeres. Parece ser que la función devuelve un objeto de la clase *kmeans*, vamos a mencionar algunos atributos de interés.

1. **cluster:** Un vector de enteros, con tantas casillas como instancias hay en nuestro dataset, indicando a qué clúster pertenece.
2. **centers:** Una matriz que contiene los centroides de cada clúster.
3. **totss:** La suma de los errores al cuadrado, es decir, SSE; que, como hemos dicho previamente, es la métrica de nuestra partición.
4. **withinss:** La suma de los errores al cuadrado **de cada clúster**. Por lo que será un vector de otras dos casillas, y en la posición correspondiente a cada clúster se encontrará su error.
5. **size:** Cuántos casos se encuentran en cada clúster.
6. **iter:** Lo mismo que en WEKA, nos indica cuántas iteraciones ha hecho el algoritmo antes de encontrar la partición final.

Imprimiendo el objeto resultante, nos sale lo siguiente:

```
kmeans.res <- kmeans(foodNumeric, centers = 2) # Guardamos el resultado en kmeans.res
print(kmeans.res) #Imprimimos el objeto resultante
```

```
## K-means clustering with 2 clusters of sizes 18, 9
##
## Cluster means:
##          Energy Protein      Fat   Calcium      Iron
## 1 145.5556      19 6.444444 61.555556 2.338889
## 2 331.1111      19 27.555556 8.777778 2.466667
```

```

## 
## Clustering vector:
## [1] 2 2 2 2 1 1 1 1 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## 
## Within cluster sum of squares by cluster:
## [1] 178738.40 23751.03
## (between_SS / total_SS =  52.7 %)
## 
## Available components:
## 
## [1] "cluster"      "centers"       "totss"        "withinss"      "tot.withinss"
## [6] "betweenss"    "size"          "iter"         "ifault"

```

Ahora vamos a ver el atributo cluster del objeto. Como hemos dicho antes, va a obtener un array de veinte y siete enteros, cada uno para un alimento distinto, indicando a qué clúster pertenecen. Para asegurarnos de que no faltaba ningún alimento, he añadido una línea.

```
kmeans.res$cluster
```

## [1] 27

Vale. Con esa representación nos quedamos sin mucho que decir. Por esto mismo vamos a hacer una nueva llamada a `cbind`, una función que combina vectores con datasets. Nosotros le pasamos el dataset de `food`, —el primero de todos, no en el que removemos la columna de nombres—, y el vector de los clústeres que hemos imprimido anteriormente.

```
foodWithCluster <- cbind(food, cluster = kmeans.res$cluster)
foodWithCluster
```

##		Name	Energy	Protein	Fat	Calcium	Iron	cluster
## 1		Braised Beef	340	20	28	9	2.6	2
## 2		Hamburger	245	21	17	9	2.7	2
## 3		Roast Beef	420	15	39	7	2.0	2
## 4		Beef steak	375	19	32	9	2.6	2
## 5		Canned Beef	180	22	10	17	3.7	1
## 6		Broiled Chicken	115	20	3	8	1.4	1
## 7		Canned Chicken	170	25	7	12	1.5	1
## 8		Beef Heart	160	26	5	14	5.9	1
## 9		Roast Lamb Leg	265	20	20	9	2.6	2
## 10	Roast	Lamb Shoulder	300	18	25	9	2.3	2
## 11		Smoked Ham	340	20	28	9	2.5	2
## 12		Pork Roast	340	19	29	9	2.5	2
## 13		Pork Simmered	355	19	30	9	2.4	2
## 14		Beef Tongue	205	18	14	7	2.5	1
## 15		Veal Cutlet	185	23	9	9	2.7	1
## 16		Baked Bluefish	135	22	4	25	0.6	1
## 17		Raw Clams	70	11	1	82	6.0	1
## 18		Canned Clams	45	7	1	74	5.4	1
## 19		Canned Crab meat	90	14	2	38	0.8	1
## 20		Fried Haddock	135	16	5	15	0.5	1
## 21		Broiled Mackerel	200	19	13	5	1.0	1
## 22		Canned Mackerel	155	16	9	157	1.8	1
## 23		Fried Perch	195	16	11	14	1.3	1

```

## 24      Canned Salmon     120     17     5     159   0.7     1
## 25      Canned Sardines   180     22     9     367   2.5     1
## 26      Canned Tuna       170     25     7      7   1.2     1
## 27      Canned Shrimp     110     23     1     98   2.6     1
  
```

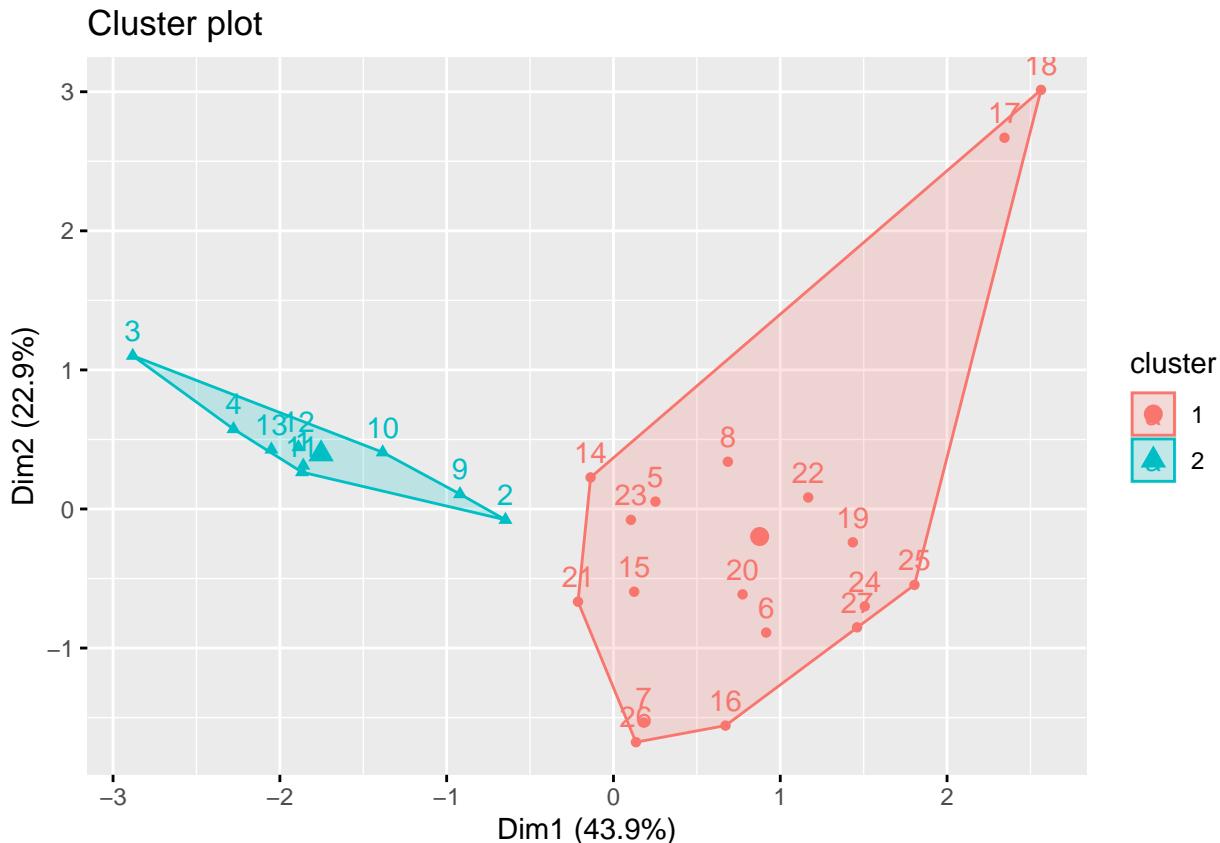
Así mucho mejor, y más ordenado que en WEKA: recordemos que en dicha aplicación en ningún momento pudimos ver a qué clúster pertenecía cada variable con tanta claridad como en esta tabla —es cierto que en el gráfico se podría haber visto, pero era algo tedioso ir aspa por aspa. Para el siguiente trozo de código, cargamos la librería “factoextra” y llamamos a la función fviz\_cluster. Ésta última función va a mostrar nuestro objeto de la clase “kmeans”, además de que le especificamos el tipo de data con el que estamos trabajando.

```

# install.packages('factoextra') #Instalación del paquete 'factoextra', un
# paquete para visualizar los datos y facilitar su análisis.
library(factoextra)
  
```

```

## Loading required package: ggplot2
## Welcome! Want to learn more? See two factoextra-related books at https://goo.gl/ve3WBa
# ¿Qué muestra la gráfica? Explica
fviz_cluster(kmeans.res, data = foodNumeric)
  
```



¿Qué representan los dos ejes de la siguiente gráfica? ¿Y sus porcentajes entre paréntesis?

Primero vamos a entender qué significan los ejes. En WEKA lo que se nos mostraba era la partición pero con respecto a distintas variables, y el gráfico que se nos muestra aquí tiene poco que ver con alguno de los que hemos visto. Es porque esta función, cuando el número de dimensiones es mayor que tres, va a calcular los componentes principales y mostrar las variables respecto a los “nuevos” ejes PCA1 y PCA2. Recordemos que el objetivo de éstos nuevos ejes era capturar la máxima varianza posible, cosa que se nos muestra entre

paréntesis para cada componente: Dim1 captura el 43,9 % y Dim2, el 22,9 %. Una gran diferencia con respecto a previos laboratorios en los que éramos capaces de obtener ejes que abarcaban un porcentaje de varianza bastante mayor —llegamos a ver hasta del 90 %!. Esto es normal: pese a que no haya una norma para ello, es de esperar que en cuantos menos ejes deseemos representar los datos y cuantas más dimensiones haya, será más difícil de obtener de, entre tan solo dos componentes principales, cubrir gran parte de la varianza. En total, estamos cubriendo un total del 66,8 %.

### Clustering Jerárquico.

Ahora vamos a intentar un nuevo modo de partición como lo son los métodos jerárquicos aglomerativos. Vamos a explicar, paso por paso, en qué se basa:

- **Paso 1:** Asignamos un clúster a cada caso, de tal forma que si nuestro conjunto de datos tiene  $n$  casos, tendremos  $n$  clústeres, ni uno más, ni uno menos.
- **Paso 2:** Dada una matriz de distancias entre todos los casos, escogemos pares más cercanos y los juntamos, de tal forma que ahora cada par más cercano el uno del otro ahora formarán un clúster.
- **Paso 3:** Ahora calculamos las distancias entre los nuevos clústeres obtenidos, que no tenemos que confundir con las distancias entre casos: ambos se calculan de formas distintas. Al igual que entre casos, la distancia entre clústeres se mide de formas distintas.
- **Paso 4:** Seguimos iterando hasta que todos los casos se encuentren en el mismo clúster.

Ahora bien, *¿cómo medimos la distancia entre clústeres?* Hay varias formas, entre ellas son tres las que vamos a mencionar:

1. **Single Linkage:** También conocido como vecino más próximo. Dados dos clústeres, la distancia entre ambos será la misma a la distancia entre los dos casos más cercanos de cada uno.
2. **Complete Linkage:** También conocido como vecino más lejano. Dados dos clústeres, la distancia entre ambos será la misma a la distancia entre los dos casos más lejanos.
3. **UPGMA:** Unweighted Pair Group Method Using Arithmetic Averages. La distancia entre dos clústeres será la media de la distancia entre todos los casos de un clúster con respecto al otro: *no* hay que calcular la distancia entre los casos dentro de un clúster, solo entre los que están en el otro.

### Por cada modo distinto de los previos obtendremos distintas particiones.

Ya que sabemos de lo que la clusterización jerárquica trata, estamos preparados para entender cada línea de código. Sin embargo, debemos hacer unas pocas preparaciones:

```
range01 <- function(x) {
  (x - min(x))/(max(x) - min(x))
}
foodNormalized <- range01(foodNumeric)
distances <- dist(foodNormalized)
```

En la primera línea, se define la siguiente función que aplica esta fórmula (usando la sintaxis de R):

$$x_{\text{normalizado}} = \frac{x - \text{valor máximo}}{\text{valor máximo} - \text{valor mínimo}}$$

Recordemos que lo que hace la normalización es acotar los valores de un dataset dado entre 0 y 1 (he ahí el nombre de la función *range01*). En la siguiente línea, por lo tanto, pasamos a dicha función nuestro dataset —evitando la variable numérica, como es obvio. Ésto es necesario porque para la clusterización jerárquica los datos deben estar normalizados.

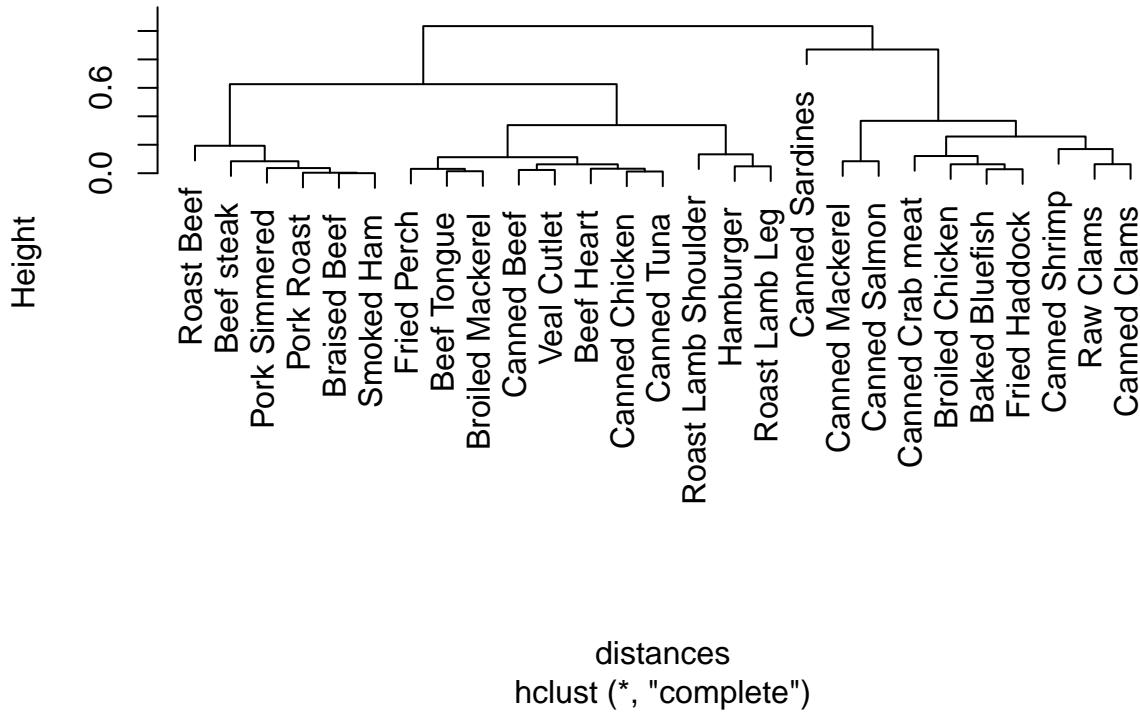
Por último, pero igual de importante, calculamos la matriz de distancias. Como hemos dicho antes, es necesario para que el algoritmo pueda juntar dos casos en función de su cercanía. No solo esto, sino que también nos ahorra mucho trabajo a la hora de medir la distancia entre clústeres, da igual el método que usemos.

Ahora que tenemos todo preparado, vamos a llamar a la función que se encarga de realizar el método jerárquico, ésta se llama `hclust`. Pasamos la matriz de distancias `y`, pese a que está definido de antemano, le indicamos que use el método del máximo, o complete linkage (explicado anteriormente), para calcular la distancia entre dos clústeres.

```

foodHClust <- hclust(distances, method = "complete") #Pasamos por parámetro la matriz de
# las distancias llamada distances.
plot(foodHClust, labels = food$name, main = "Clustering jerárquico - Complete Linkage")
  
```

## Clustering jerárquico – Complete Linkage



La figura de arriba es el *dendrograma* del proceso. Nos muestra toda la jerarquía de particiones, y cómo se juntan desde que empezamos con un clúster por caso, hasta que tenemos un único clúster. Con la altura, `height`, que se muestra a la izquierda, se denota la distancia a la que se encontraban los clústeres antes de juntarlos. Si bien en este dendrograma se muestran todas las iteraciones hasta que se unifican los clústeres, podemos decidir detener el algoritmo antes de que ello ocurra, aunque la idea del gráfico es darnos la información de dónde queremos cortar el déndograma y con qué clústeres nos queremos quedar. Como podemos ver, nos ofrece bastante flexibilidad.

¿Se observan diferencias aplicando "single linkage" en vez de "complete linkage"? Consulta la ayuda de la función "hclust".

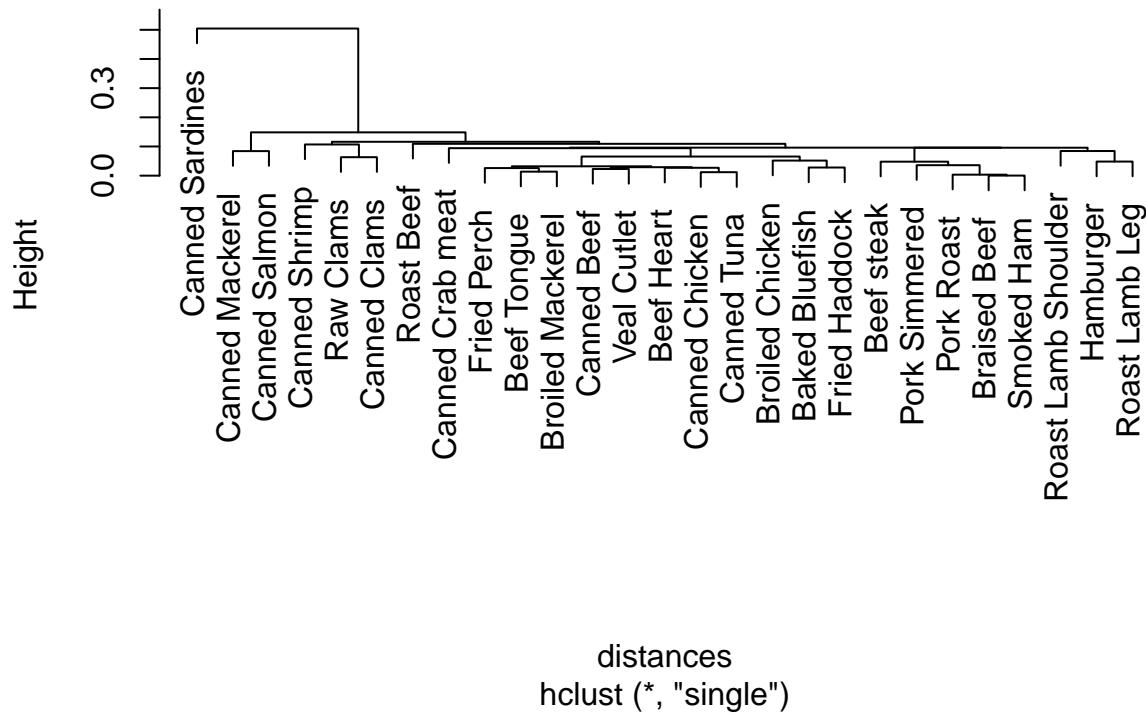
Ya hemos dicho antes que dependiendo del método de medir habrá un diferente dendrograma, dado que ésto resultará en un distinto método aglomerativo.

Sin embargo, cabe destacar que serán idénticos hasta la primera iteración porque buscan lo mismo: las parejas más cercanas entre sí. Para indicar si queremos usar una u otra en R, como hemos dicho también previamente, tenemos el parámetro `method`. Éste puede tomar distintos valores, fijándonos en la documentación de `hclust`, tenemos "single", "complete" y "average".

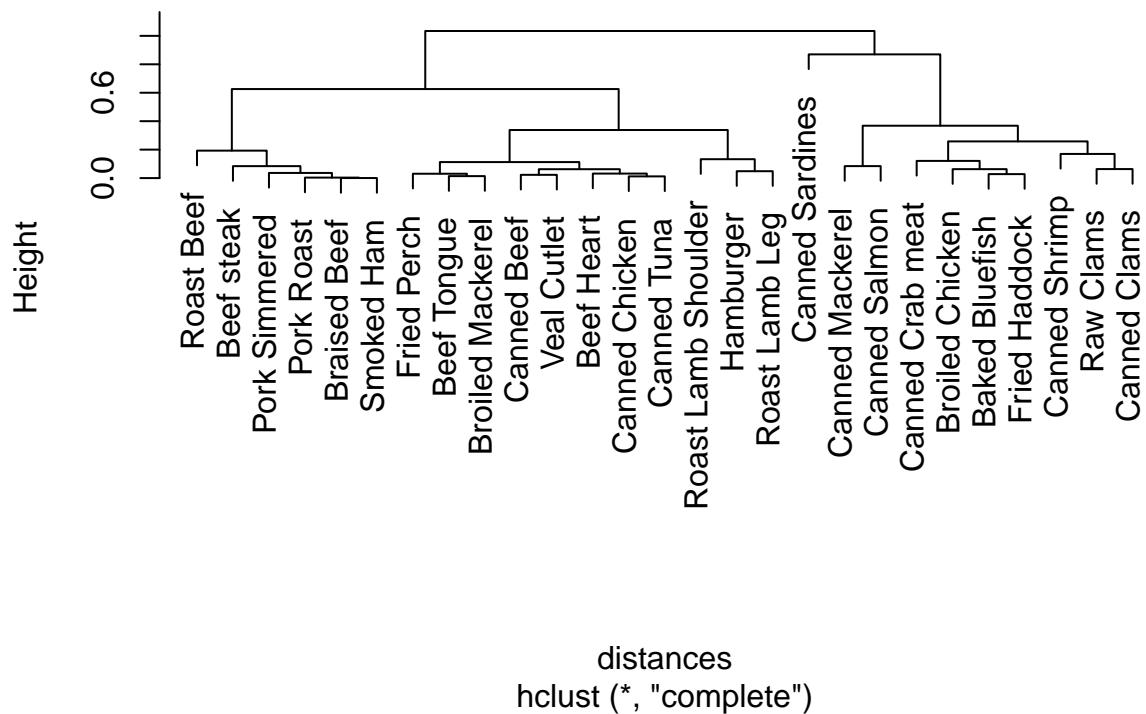
```

foodHClustSingle <- hclust(distances, method = "single")
plot(foodHClustSingle, labels = food$name, main = "Average")
  
```

## Average



## UPGMA



Ahora, vamos a fijarnos en una página web[Naftali, 2014] que simula k-means, e intentaremos explicar y entender mejor cómo funciona.

La página web referenciada es un buen recurso que profundiza en k-means y nos explica varias propiedades interesantes que no hemos aprendido en clase. A parte de esto, nos da opciones de escoger los puntos iniciales, además de que cuenta con varios datasets para practicar.

**¿A qué se refiere la opción “How to pick the initial centroids” y qué consecuencia tiene?**  
**¿A qué se refieren las tres opciones que recoge?**

Se nos brinda la opción de cómo queremos que se escojan los puntos iniciales:

1. **I'll Choose:** Nosotros somos los que escogemos los puntos iniciales
2. **Randomly:** Los puntos se escogen aleatoriamente.
3. **Farthest Point:** Primero se escoge un punto aleatorio como centroide, luego se escoge el siguiente centroide como el punto más lejano al previamente escogido. Así consecutivamente hasta haber terminado de escoger todos los centroides.

En función de la que escojamos, se formarán distintos clústeres. Importante destacar, que no es hay que seleccionar un punto, sino que podemos asignar como centroide la parte del gráfico que nosotros deseemos. Ésto nos puede interesar también a la hora de experimentar con el algoritmo, y es que da mucho juego que nosotros seamos los que escogemos los puntos: Podemos escoger puntos cercanos, formar pares, separarlos mucho... En definitiva, lo interesante sería buscar casos extremos/excepcionales para ver cómo se las arregla el algoritmo.

### What kind of data would you like?

Aquí escogemos la distribución de puntos que deseemos; tenemos un total de ocho opciones. Como esto no es más que una simulación, pensada únicamente para enseñar cómo funciona el algoritmo, no podemos pasar nuestro propio conjunto de datos lásticamente. Aun así, esto nos sirve de sobra para resolver todas nuestras dudas en el caso de que tuviéramos alguna, además de que el creador ofrece distribuciones de puntos que dan bastante juego.

### ¿Te ha gustado? Coméntame algo.

La página web es una buena herramienta para entender k-means. La parte más interesante es cuando habla sobre las propiedades del algoritmo, entre ellas me han llamado la atención varias:

#### 1. El algoritmo converge.

Previamente a leer esta propiedad pensaba que, en casos muy extremos el algoritmo divergía. Supuse que podía ocurrir que un caso estuviera “bailando” entre dos clústeres distintos indefinidamente. Ahora sé que no, puesto que eso tan solo ocurre en casos en los que el algoritmo esté mal implementado: en este tipo de casos siempre hay un criterio que decide detenerse en el caso de que haya una mínima diferencia entre centroides previos y actuales, evitando esto.

#### 2. Es posible que no se asigne ningún punto a un cluster.

Si un centroide no es el más cercano ni a un punto, se quedará sin asignación. La cosa es que hay varios enfoques para éste problema: uno de ellos es borrar el centroide, otro es asignarlo a otro punto aleatorio, o dejarlo quieto hasta que quizás en otro momento se le acabe asignando un punto. Es curioso ver cómo hasta los algoritmos más simples pueden causar dificultades.

#### 3. Estrategias de inicialización.

Este es un tema del que también se ha hablado en clase, y que me habría gustado entender mejor, pero obviamente no se puede destinar toda la asignatura a esto. Me ha parecido muy interesante el estudio que se ha hecho respecto a este tema. Entre otros, lo que he sacado en claro cómo, pese a haber habido métodos de inicialización superiores a otros, no son tan buenos como para ser la norma. Es curioso ver cómo pese a escoger el k ideal, las probabilidades de que seamos capaces de manualmente escoger la mejor inicialización sea tan baja.

**Visita distintas webs encontradas por compañeros y escoge la que más te guste.**

Todas me parecen útiles en lo que respecta a hacernos visualizar k-means, no veo ninguna que destaque sobre el resto. Quizá habría estado mejor que alguna de las páginas anteriores añadiera algo más de explicación sobre cómo se ha implementado y las decisiones que han tomado en su algoritmo, como en la página anterior. Si bien tienen que hacerlo sencillo de ver, no habría venido de más que enseñasen una simulación en la que el algoritmo “sufra” por así decirlo.

## Noveno Laboratorio (15-16 de noviembre)

Laboratorio de scripting con R. Hay que comentar cada línea de código y parámetro. Entender de lo que trata el script y añadir nuestro toque personal.

**La siguiente línea debes "adaptarla" al directorio donde estén los datasets de trabajo.**

En nuestro caso, al estar usando Rmd, usamos una herramienta llamada knit para conseguir un output en pdf. Lo que pasa es que para ésta herramienta no vale establecer el working directory con `setwd`. Es decir, la siguiente forma nos devuelve un error:

```
# setwd('/home/mikel/Documents/Latex_Proyectos/MDD Laboratorio Segunda
# Parte/datasets') <-- De ésta forma no sirve, ni si quiera nos procesaría el
# fichero. Lo dejamos comentado.
```

Es insertando ésta línea de código como conseguimos establecer el directorio exitosamente:

```
dir = "/home/mikel/Documents/Latex_Proyectos/MDD Laboratorio Segunda Parte/datasets"
knitr::opts_knit$set(root.dir = dir)
```

**Encontrarás el fichero iris.csv en egela, descárgatelo en local y cárgalo en R.**

El dataset iris es una colección que dadas las variables predictoras de una planta llamada iris, trata de predecir su especie.

Ahora que ya hemos conseguido establecer el directorio donde acostumbramos a guardar los ficheros, podemos cargarlo.

*Pero, ¿cómo lo “cargamos en R”?*

Pues depende del formato del fichero que estemos empleando, nos interesará usar una función u otra. En lo que nos concierne, hasta ahora hemos visto dos tipos de formatos: arff (Attribute Relation File Format) y csv (Comma Separated Values). Por lo tanto, sería de esperar que R nos ofrezca distintas funciones para leer cada fichero. Veamos cuál se usa para los csv:

```
iris <- read.csv("iris.csv", header = TRUE, sep = ",")
class(iris) # ¿De qué clase será la variable?
```

```
## [1] "data.frame"
```

Hemos usado `read.csv`. Pero no solo hemos pasado el fichero, sino que hemos usado otros parámetros también, veamos qué hacen:

- **header=TRUE**: Indicamos que el fichero contiene en la primera fila los nombres de cada variable. Ésto en csv es imperativo, por lo que por defecto, lo tenemos también establecido a TRUE.
- **sep=","**: Indicamos que cada valor de nuestro fichero está separado por una coma. Como bien hemos dicho antes, los ficheros csv tiene como característica que están separados por coma, por lo que de normal está establecido así.

Para terminar, guardamos el valor que la función en la variable `iris`, que no será más que un data frame.

```
summary(iris)
```

```
##   sepal.length   sepal.width   petal.length   petal.width
##   Min. :4.300   Min. :2.000   Min. :1.000   Min. :0.100
##   1st Qu.:5.100  1st Qu.:2.800  1st Qu.:1.600  1st Qu.:0.300
##   Median :5.800  Median :3.000   Median :4.350   Median :1.300
##   Mean   :5.843  Mean   :3.057   Mean   :3.758   Mean   :1.199
##   3rd Qu.:6.400  3rd Qu.:3.300  3rd Qu.:5.100   3rd Qu.:1.800
##   Max.  :7.900   Max.  :4.400   Max.  :6.900   Max.  :2.500
##   variety
```

```

##  Length:150
##  Class :character
##  Mode   :character
##
## 
## 
## 

```

Summary es una función genérica que sirve para poder hacer un resumen del objeto que nosotros le estemos pasando por parámetro. Para los objetos de la clase *data.frame* se ha implementado un formato analítico; por cada variable se muestran distintas métricas, veamos qué significa cada una.

- **Min.** El valor mínimo para la variable de la columna entre todas las instancias del dataset.
- **1st Quartile.** En castellano conocido como el primer cuartil, es el número que se encuentra en la mitad entre el último número y el mediano. Debajo de él se encuentran el % 25 de los datos.
- **Median.** Median, o el mediano, es el equivalente al segundo cuartil. Es el número que se encuentra en la mitad de de todas las instancias de la variable. Corta los datos por la mitad — el 50% de los datos se encuentra por encima, y viceversa. Hay que tener cuidado con este valor: si el número de instancias es par, no habrá número en la mitad, por lo que el valor de éste será la media aritmética entre los dos números que se encuentran en la mitad.
- **Mean.** La media aritmética de una variable dada.
- **3rd Quartile.** El tercer cuartil, el número que se encuentra entre el mediano y el valor máximo. Encima de éste se encuentran el % 75 de los casos.
- **Max.** El valor máximo para la variable de la columna entre todas las instancias del dataset.

Con la siguiente línea escribimos los headers del fichero, es decir, los nombres de cada variable:

```
names(iris)
```

```
## [1] "sepal.length" "sepal.width"  "petal.length" "petal.width"  "variety"
```

Aquí podemos ver las variables que se están teniendo en cuenta para predecir la categoría, así como la anchura de los pétalos o su largura.

Ahora enseñamos las seis primeras filas/instancias del fichero .csv para ver cómo se representa cada caso.

```
head(iris)
```

```

##   sepal.length sepal.width petal.length petal.width variety
## 1      5.1       3.5      1.4       0.2  Setosa
## 2      4.9       3.0      1.4       0.2  Setosa
## 3      4.7       3.2      1.3       0.2  Setosa
## 4      4.6       3.1      1.5       0.2  Setosa
## 5      5.0       3.6      1.4       0.2  Setosa
## 6      5.4       3.9      1.7       0.4  Setosa

```

Si queremos que enseñe más de seis instancias, siempre podemos especificarlo mediante un parámetro con el que cuenta:

```
head(iris, n = 10L) #con n=10L decimos que queremos ver las diez primeras.
```

```

##   sepal.length sepal.width petal.length petal.width variety
## 1      5.1       3.5      1.4       0.2  Setosa
## 2      4.9       3.0      1.4       0.2  Setosa
## 3      4.7       3.2      1.3       0.2  Setosa
## 4      4.6       3.1      1.5       0.2  Setosa
## 5      5.0       3.6      1.4       0.2  Setosa
## 6      5.4       3.9      1.7       0.4  Setosa

```

```

## 7      4.6      3.4      1.4      0.3  Setosa
## 8      5.0      3.4      1.5      0.2  Setosa
## 9      4.4      2.9      1.4      0.2  Setosa
## 10     4.9      3.1      1.5      0.1  Setosa
  
```

También tenemos la función contraria, que nos devuelve las últimas instancias:

```
tail(iris)
```

```

##   sepal.length sepal.width petal.length petal.width   variety
## 145      6.7      3.3      5.7      2.5 Virginica
## 146      6.7      3.0      5.2      2.3 Virginica
## 147      6.3      2.5      5.0      1.9 Virginica
## 148      6.5      3.0      5.2      2.0 Virginica
## 149      6.2      3.4      5.4      2.3 Virginica
## 150      5.9      3.0      5.1      1.8 Virginica

tail(iris, n = 10L) #existe el mismo parámetro para tail.
  
```

```

##   sepal.length sepal.width petal.length petal.width   variety
## 141      6.7      3.1      5.6      2.4 Virginica
## 142      6.9      3.1      5.1      2.3 Virginica
## 143      5.8      2.7      5.1      1.9 Virginica
## 144      6.8      3.2      5.9      2.3 Virginica
## 145      6.7      3.3      5.7      2.5 Virginica
## 146      6.7      3.0      5.2      2.3 Virginica
## 147      6.3      2.5      5.0      1.9 Virginica
## 148      6.5      3.0      5.2      2.0 Virginica
## 149      6.2      3.4      5.4      2.3 Virginica
## 150      5.9      3.0      5.1      1.8 Virginica
  
```

¿Qué aplica el siguiente comando?

```
iris$variety <- as.factor(iris$variety)
table(iris$variety) # Mostramos la frecuencia de cada categoría.
  
```

```

##
##   Setosa Versicolor  Virginica
##      50      50      50

class(iris$variety) # ¿Ahora qué clase será la columna?
  
```

```
## [1] "factor"
  
```

Antes de entender lo que el comando *as.factor* aplica, debemos entender lo que significa “factorizar” una variable, y por qué nos interesa hacerlo. Para empezar, éste proceso de factorizar se puede aplicar tanto en variables nominales como numéricas; nosotros lo empleamos porque en el paquete Caret [Kuhn and Max, 2008], no hay soporte para strings. Nos permite evaluar los distintos valores que una variable categórica puede tomar —en nuestro caso, tres— y por cada distinta categoría que haya crea un nivel o “level”. Para entenderlo mejor, veamos qué es lo que ha hecho con iris:

```
iris$variety
  
```

```

## [1] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
## [7] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
## [13] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
## [19] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
## [25] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
## [31] Setosa  Setosa  Setosa  Setosa  Setosa  Setosa
  
```

```

## [37] Setosa      Setosa      Setosa      Setosa      Setosa      Setosa
## [43] Setosa      Setosa      Setosa      Setosa      Setosa      Setosa
## [49] Setosa      Setosa      Versicolor  Versicolor  Versicolor  Versicolor
## [55] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [61] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [67] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [73] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [79] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [85] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [91] Versicolor  Versicolor  Versicolor  Versicolor  Versicolor  Versicolor
## [97] Versicolor  Versicolor  Versicolor  Virginica   Virginica   Virginica
## [103] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [109] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [115] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [121] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [127] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [133] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [139] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## [145] Virginica  Virginica   Virginica   Virginica   Virginica   Virginica
## Levels: Setosa Versicolor Virginica
  
```

Como hemos dicho previamente, por cada categoría distinta se ha creado un nivel: Setosa, Versicolor y Virginica. Además de esto, internamente, éstos valores se almacenarán como números enteros, (por ejemplo, 0=Setosa, 1=Versicolor, 2=Virginica), por lo que estaremos usando menos memoria.

Entonces, volviendo al trozo de código que hemos ejecutado, con *as.factor* estamos convirtiendo directamente el vector que pasamos por parámetro en factor, y sobreescribiendo la columna de *variety*. No es más que la forma abreviada de lo siguiente.

```

irisVarieties <- c("Setosa", "Versicolor", "Virginica") #Creamos un vector con las
# distintas categorías.
iris$variety <- factor(iris$variety, levels = irisVarieties) # Y en levels especificamos
# los niveles manualmente.
  
```

Es decir, *as.factor* encuentra las categorías por nosotros, por eso se dice que suele ser más rápido.

### Instalación del paquete caret.

```

# Primero cargamos las librerías de las que depende caret.
library(ggplot2)
library(lattice)
# Cargamos la librería caret.
library(caret)
  
```

### ¿Para qué hacemos esto?

```
set.seed("1234567890")
```

Al establecer una semilla igual, los resultados que nos ofrecerá cualquier máquina que ejecute este código en R tendrá los mismos resultados. Ésto es útil, por ejemplo cuando compartimos nuestro trabajo, para que el resto de nuestros compañeros pueda trabajar con nuestros datos.

### ¿Pero cómo funciona?

Resulta que los ordenadores no pueden generar un número aleatorio de la nada como el ser humano (que tan poco generamos números puramente aleatorios). Por eso usa un número inicial que le denominamos “semilla” o “seed”, y trás aplicarle un Random Number Generator (más conocido como RNG) obtiene el output. Es decir, cualquiera que tenga la semilla 1234567890 obtendrá los mismos resultados que nosotros. Veamos si es verdad:

```

# Usamos runif(), una función que usaremos para generar un número del 1 al 0 en
# nuestro caso (sirve para otras cosas, realmente su uso no es el que le vamos
# a dar ahora.)
primero <- runif(1)
# Aplicamos de nuevo la misma semilla. Esto es porque después de generar un
# número, para evitar que el mismo número se genere una vez más, se modifica la
# semilla.
set.seed("1234567890")
segundo <- runif(1)
identical(primero, segundo) #¿Son iguales?
  
```

## [1] TRUE

Como podemos ver, un ordenador como tal es incapaz de generar un número aleatorio: necesita algo de dónde partir, no puede crear uno de la nada, es por esto que se les suele denominar “pseudorandom numbers”. Es más, con saber el algoritmo detrás de la generación de números, también conocido como RNG, podríamos saber cuál es el output resultante con la semilla establecida.

Para terminar con esto, como dato curioso: Si bien se ha establecido que un ordenador es incapaz de generar números aleatorios, siempre puede fijarse en su entorno para obtener una semilla puramente aleatoria o Random Input Number (RIN), como lo hacen los Hardware RNG (HRNG). También existe otro método que obtiene el RIN midiendo la radiación del elemento Am-241, usado por una asociación de loterías. Más información en la bibliografía.

```

# Volvemos a establecer la misma semilla que la última vez.
set.seed("1234567890")
  
```

**¿Qué hace la siguiente función?**

```
help("createDataPartition")
```

El comando help ofrece el siguiente resumen:

## Data Splitting functions

### Description

A series of test/training partitions are created using `createDataPartition` while `createResample` creates one or more bootstrap samples. `createFolds` splits the data into k groups while `createTimeSlices` creates cross-validation split for series data. `groupKFold` splits the data based on a grouping factor.

No es más que una forma de particionar nuestra colección de datos. Podemos ver que habla de separarlas en función de cuál usaremos para entrenar y testear nuestro modelo, esto nos debería sonar a una de las técnicas de validación que hemos usado llamada Holdout. En ésta técnica usábamos un porcentaje de nuestra colección de datos total para formar nuestro modelo, mientras que el porcentaje no usado era empleado para armar la matriz de confusión.

También menciona las otras técnicas de validación que hemos dado en clase, y las funciones necesarias para realizar dichas particiones, que se encuentran en el mismo paquete Caret.

Cabe destacar que en el caso de que nuestra variable a predecir no fuera numérica, tenemos que pasarle un objeto de la clase factor, motivo por el que previamente se ha hecho dicha conversión.

La semilla se usa aquí porque al particionar, se decide aleatoriamente qué muestra va a acabar en el conjunto de test o en el conjunto de entrenamiento.

Ya sabemos lo que hace, ahora veamos cómo se le ha llamado.

**Explica cada parámetro de cada función.**

```
trainSetIndexes <- createDataPartition(y = iris$variety, p = 0.66, list = FALSE)
```

Aquí estamos ofreciendo en *y* la columna de variables categóricas. Le pasamos ésta por que es la variable que nosotros deseamos predecir, y el problema que planteaba el dataset.

Con *p* = .66 indicamos el tamaño de la partición del conjunto de entrenamiento, luego tenemos el 66 % dedicado al entrenamiento, y el 34 % dedicado al test. Si todo ha ido bien, habrá  $150 * 0.66 = 99$  casos en el conjunto de entrenamiento:

```
nrow(trainSetIndexes)
```

```
## [1] 99
```

**¿Qué recoge el objeto "trainSetIndexes"? ¿Y "-trainSetIndexes"?**

El vector resultante de la previa llamada a la función nos devuelve un vector de enteros, cada número indica la posición de una fila en la colección de datos original. Antes ya se ha dicho que consta de 99 índices a casos. Veamos un ejemplo de las diez primeras filas que ha escogido mediante el comando *head* que hemos usado antes:

```
head(trainSetIndexes, n = 10L) #Diez primeras filas de trainSet
```

```
##      Resample1
##  [1,]      2
##  [2,]      4
##  [3,]      5
##  [4,]      6
##  [5,]      7
##  [6,]     11
##  [7,]     12
##  [8,]     13
##  [9,]     14
## [10,]     15
```

Como hemos dicho antes, sabemos que la fila dos de la colección original se encuentra en el conjunto de entrenamiento, así como la fila dos, cuatro...

Ahora ya sabemos los índices de las filas, pero, *¿cómo obtenemos los casos como tal?*

R nos lo pone fácil, pues nos permite acceder a dichos índices pasando el propio vector a la colección de datos *iris*:

```
# Importante poner la coma para indicar que deseamos obtener todas las
# variables del dataframe, de otra forma nos estaría escogiendo solo los de la
# primera columna.
trainSet <- iris[trainSetIndexes, ]
head(trainSet) #Comprobamos.
```

```
##      sepal.length sepal.width petal.length petal.width variety
## 2          4.9       3.0        1.4       0.2   Setosa
## 4          4.6       3.1        1.5       0.2   Setosa
## 5          5.0       3.6        1.4       0.2   Setosa
## 6          5.4       3.9        1.7       0.4   Setosa
## 7          4.6       3.4        1.4       0.3   Setosa
## 11         5.4       3.7        1.5       0.2   Setosa
```

Se encuentran todas las filas que el vector trainSetIndexes nos indicó. Ahora que tenemos el conjunto de entrenamiento, lo suyo sería obtener el conjunto de test. Para ello, queremos obtener las filas restantes, ésto en R se puede obtener de la siguiente forma:

```
testSet <- iris[-trainSetIndexes, ]
```

Con el menos indicamos que queremos todos los índices menos los indicados.

Ahora ya tenemos tanto el conjunto de entrenamiento como el conjunto de test. Juntos deberían sumar 150:

```
nrow(trainSet)
```

```
## [1] 99
```

```
nrow(testSet)
```

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

En efecto,  $99 + 51 = 150$ , luego se ha hecho correctamente la partición.

**La siguiente función es clave, explica cada parámetro.**

```
ctrl <- trainControl("cv", number = 10)
class(ctrl) # ¿Qué clase es?
```

```
## [1] "list"
```

Después de investigar un poco, se sabe que “cv” significa la técnica de validación Cross-Validation. Por lo tanto, con *number* = 10, estamos indicándole que realice un 10-fold cross-validation.

Ahora ya sabemos cómo se obtendrán las métricas, así como la precisión, del modelo final que obtengamos de train.

**¿Cómo se codifica cuáles son las predictoras, y cuál es la clase a predecir? Describe los parámetros usados, ¿hay algún otro parámetro interesante?**

```
KNNModel1 <- train(variety ~ ., data = trainSet, method = "knn", tuneLength = 5,
  trControl = ctrl, preProcess = c("scale"))
KNNModel1
```

```
## k-Nearest Neighbors
##
## 99 samples
## 4 predictor
## 3 classes: 'Setosa', 'Versicolor', 'Virginica'
##
## Pre-processing: scaled (4)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 89, 90, 88, 89, 90, 89, ...
## Resampling results across tuning parameters:
##
##     k      Accuracy   Kappa
##     5     0.9386869  0.9073918
##     7     0.9509091  0.9255736
##     9     0.9609091  0.9407251
##    11    0.9497980  0.9240585
##    13    0.9275758  0.8907251
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 9.
```

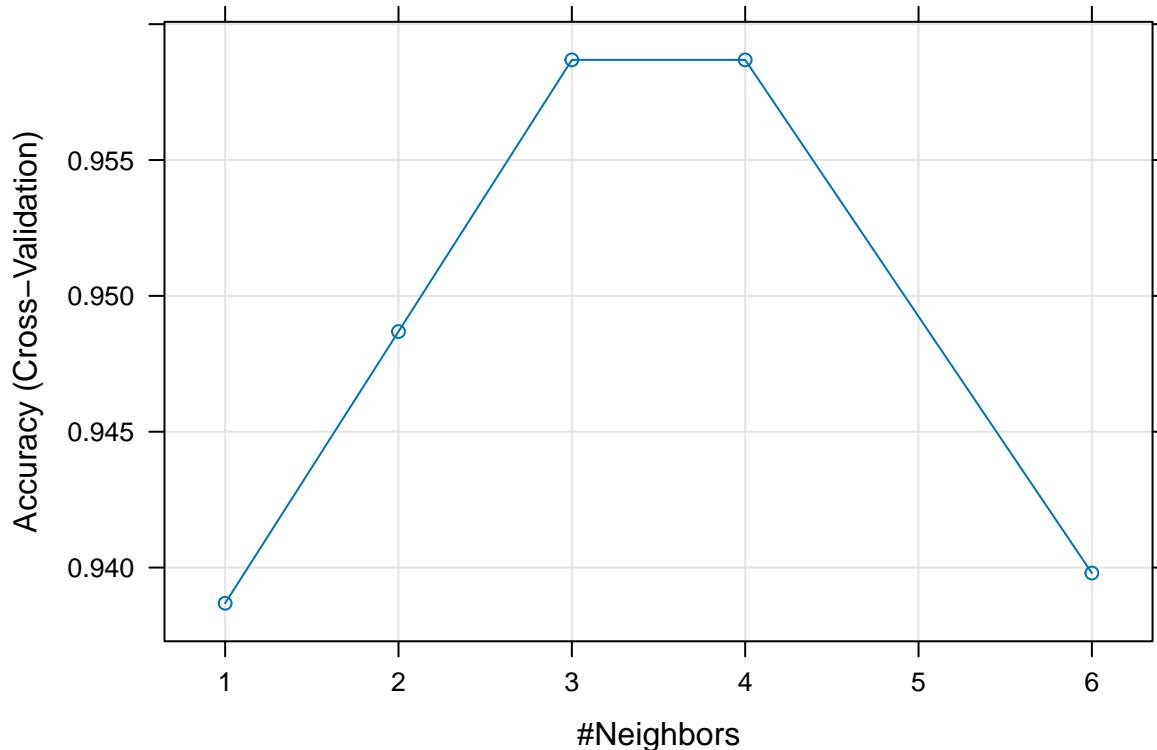
Aquí es donde entrenamos nuestro modelo, y donde vemos la utilidad de la variable trainControl. Vamos a explicar los parámetros que se encuentran:

- **data=trainSet.** En este parámetro indicamos que queremos formar el modelo con el conjunto de entrenamiento que hemos conseguido previamente.
- **method="knn".** Importante. Indicamos que vamos a usar el clasificador K-NN, K Nearest Neighbor. Éste clasificador ya lo hemos visto en laboratorios previos.
- **tuneLength=5** Otro parámetro importante. Si nos fijamos, en ningún momento estamos especificando el hiperparámetro con el que cuenta knn. Pues bien, esto es porque tuneLength se encarga de escoger la k por nosotros  $n$  veces. En nuestro caso, al hacer que  $n = 5$ , se van a crear cinco modelos distintos, uno por cada hiperparámetro. A este proceso se le llama "tuning".
- **trControl=ctrl** Específicamos que queremos usar la variable que hemos instanciado antes. Pero ahora tiene más sentido: con tuneLength=5, tendremos cinco modelos distintos, por lo que con trainControl vamos a conseguir obtener una métrica para cada uno, y saber así con cuál deberíamos quedarnos entre todas las K.
- **preProcess=c("scale")** Estamos pidiendo que escale todas las variables numéricas, lo explicaremos más adelante en la pregunta correspondiente.
- **form = variety .** Indicamos que del data frame trainSet, queremos tener en cuenta todas las columnas.

A parte de éstos parámetros, hay varios más que son interesantes. Hemos dicho que con tuneLength=5 indicamos que queremos que la función use cinco hiperparámetros tuyos, pero si queremos usar los nuestros, también podríamos mediante tuneGrid. A éste se le tendría que pasar un data frame con los números de los K que quisiéramos. Veamos cómo:

```

# Creamos un vector de hiperparámetros y los convertimos en un data frame.
k <- data.frame(k = c(1, 2, 3, 4, 6))
set.seed("1")
# Ahora especificamos en tuneGrid nuestros hiperparámetros. En nuestro caso
# hemos quitado tuneLength, pero es ignorado si le pasamos nuestros
# hiperparámetros.
My_KNNModel <- train(form = variety ~ ., data = trainSet, method = "knn", tuneGrid = k,
                      trControl = ctrl, preProcess = c("scale"))
plot(My_KNNModel)
  
```

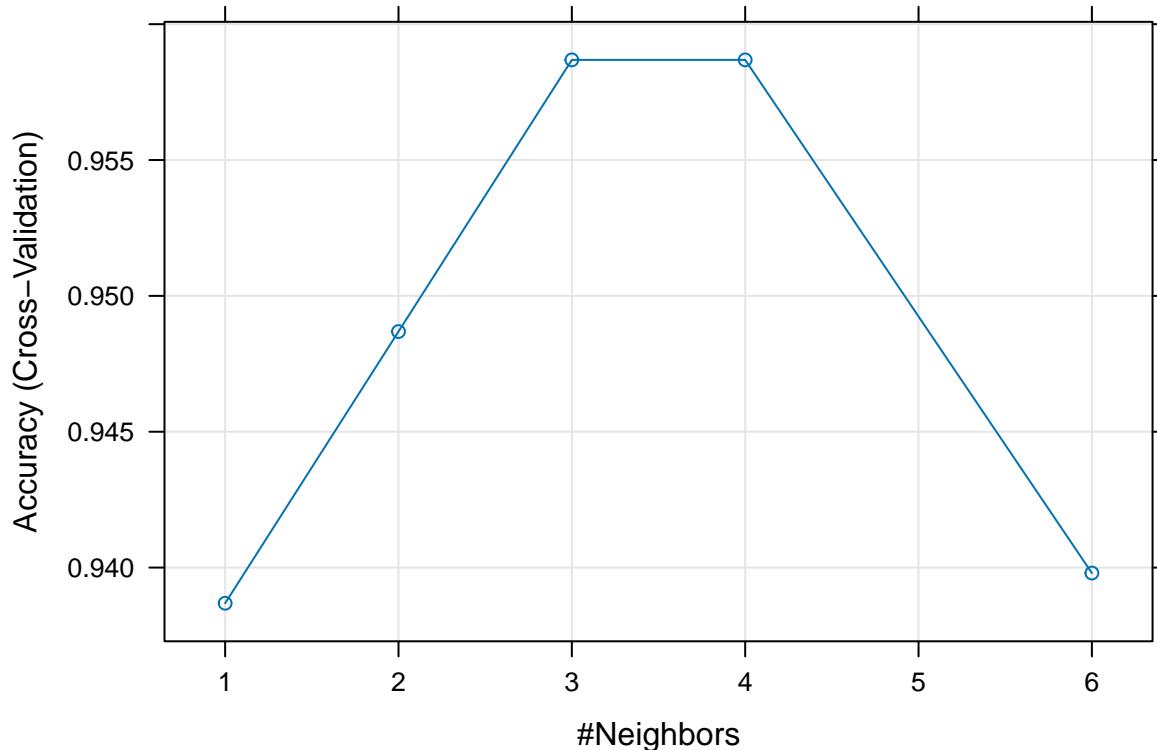


Y el gráfico nos enseña que la función ha estado realizando knn con los hiperparámetros que nosotros le hemos indicado.

Siguiendo con los parámetros, tenemos *subset*. En éste podemos indicar el subconjunto con el que queremos entrenar nuestro modelo. Es decir, que el proceso que hemos estado realizando previamente: obtener las particiones, asignarles una variable, obtener los indices de iris, asignarles otra variable... Se puede simplificar si pasamos el vector de índices a éste, y podríamos trabajar con el data frame iris original.

```

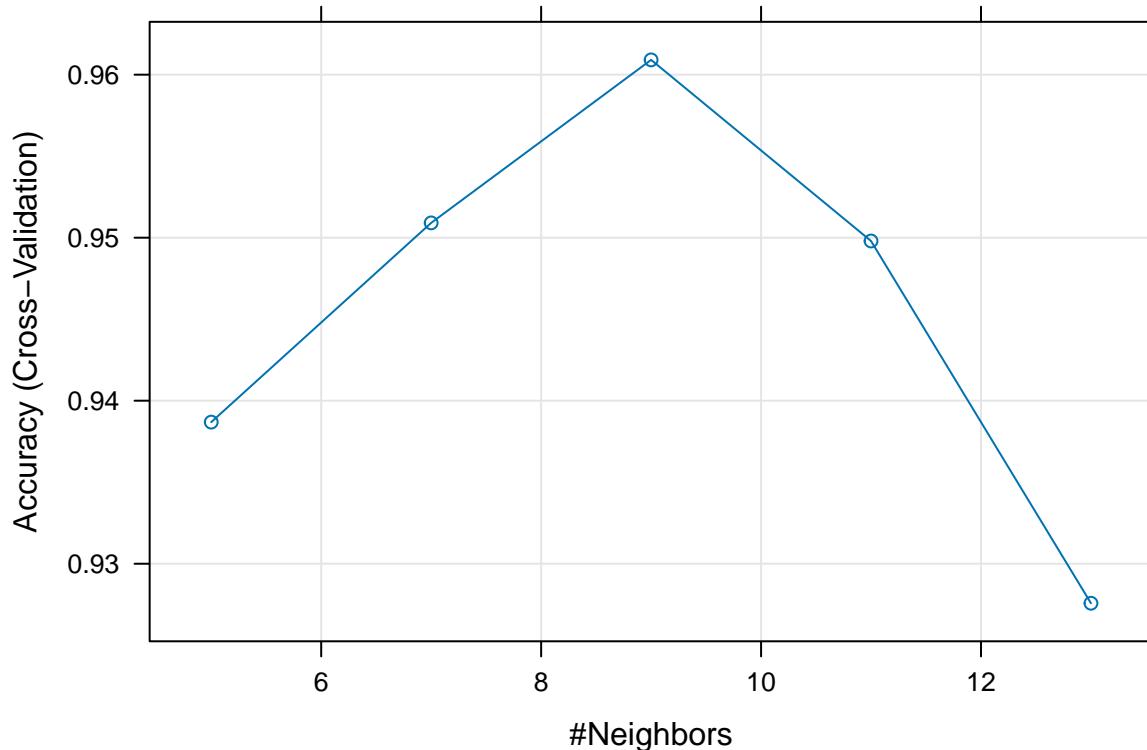
set.seed("1")
# Cambios: data=iris y subset=trainSetIndexes
plot(train(variety ~ ., data = iris, method = "knn", tuneGrid = k, trControl = ctrl,
  preProcess = c("scale"), subset = trainSetIndexes))
  
```



Se puede comprobar que son equivalentes (hemos establecido la misma semilla para que se diera este caso si son realmente iguales).

Para terminar con los parámetros: tenemos na.action, un parámetro donde podemos especificar qué hacer en el caso de que la función se encontrase con valores NA. Como vimos en WEKA, es lo mismo que uno de los filtros que nos ofrecía, donde reemplazaba los valores no disponibles numéricos con la media de todos, o en otros casos con un valor que se predecía. En éste dataset no hay ningún valor de esos, así que lo dejamos de lado.

```
plot(KNNModel1) #Visualizar el primer modelo.
```



Los gráficos son bastante intuitivos. El eje y indica el valor que el hiperparámetro  $k$  toma, y accuracy, la precisión del modelo. Curiosamente, cuando dejamos que la propia función decida los hiperparámetros, escoge números impares para evitar empates.

¿Cuál es el mejor valor para nuestro hiperparámetro?

Para ello hemos hecho un pequeño script para decidir cuál es el mejor  $K$  entre 1 y 16, explicamos cada paso en los comentarios.

```

set.seed("1234567890")  #Para obtener el mismo resultado.
hiperparametros = seq(1, 16)
k_grid <- data.frame(k = hiperparametros) #Los hiperparámetros que deseamos probar.
Num_Iterations <- 100 #Establecemos el número de iteraciones que queremos realizar.
# Creamos un vector de tantas casillas como hiperparámetros hemos creado, a
# cada índice le va a corresponder el k del mismo número.
best_times_k <- numeric(length(k_grid$k))
# Iteramos Num_iterations veces.
for (i in 1:Num_Iterations) {
  # Generamos un nuevo modelo KNN cada vez que entramos en el loop.
  KNN_Worker <- train(variety ~ ., data = trainSet, method = "knn", tuneGrid = k_grid,
    trControl = ctrl, preProcess = c("scale"))
  # En finalModel$K se guarda el K más óptimo obtenido con el método 'cv'
  # (especificado en trainControl), simplemente sumamos la casilla del k
  # correspondiente.
  best_times_k[KNN_Worker$finalModel$k] <- best_times_k[KNN_Worker$finalModel$k] +
    1
}
# Obtenemos el índice de la casilla que más veces se ha sumado por uno, ése
# será el k óptimo.
cat("El mejor K es:")
print(best_times_k)

```

```

print(which.max(best_times_k))
plot(KNN_Worker)
  
```

Cada vez que se ejecuta el previo código, nos devuelve que  $k = 9$  es el mejor valor, seguido de  $k = 4$  y  $k = 3$ . Por lo que el modelo “KNNModel1” es el mejor clasificador con el que contamos, con una precisión superior al 96 %.

**¿Qué implica "escalar" una variable numérica, tal y como hemos pedido en las opciones de preprocesso?**

Con escalar el objetivo es transformar las variables numéricas para poder representarlas de forma más sencilla. Existen varios métodos para escalar que ya se han discutido en el laboratorio de pre-procesamiento. Se observaron tres formas de escalado:

- Estandarizado:

$$X_{\text{standardize}} = \frac{X - \text{mean}(X)}{\text{sd}(X)} \quad (1)$$

- Normalizado:

$$X_{\text{normalize}} = \frac{X - \text{max}(X)}{\text{max}(X) - \text{min}(X)} \quad (2)$$

- Centralizado:

$$X_{\text{Center}} = X - \text{mean}(X) \quad (3)$$

En la página de ayuda sobre el pre-procesamiento en el paquete Caret, se destaca esta línea:

## Pre-Processing of Predictors

...

method = “center” subtracts the mean of the predictor’s data (again from the data in x) from the predictor values while method = “scale” divides by the standard deviation.

...

Es decir, que con “scale”, no solo resta cada valor de una muestra por la media, sino que también lo divide por la desviación estándar. Entre los que se han mencionado arriba, se trata de la estandarización: se acotan los valores numéricos entre 0 y 1.

**¿Se muestra la matriz de confusión en el "orden" que lo hace WEKA?**

Antes que nada, se va a averiguar cuál es el valor de K que va a emplear nuestro clasificador, para tener una mejor referencia:

```

# Se encuentra en el siguiente apartado de la instancia:
KNNModel1$finalModel$k
  
```

```

## [1] 9
  
```

```

# Con predict usamos nuestro modelo KNN con el conjunto de prueba. Lo guardamos
# en una variable.
KNNPredict1 <- predict(KNNModel1, newdata = testSet)
# Y con el resultado real de las variables, montamos la matriz de confusión.
confusionMatrix(KNNPredict1, testSet$variety)
  
```

```

## Confusion Matrix and Statistics
##
##             Reference
## Prediction   Setosa Versicolor Virginica
##   Setosa       17      0      0
##   Versicolor    0     17      1
##   Virginica     0      0     16
##
## Overall Statistics
##
##                 Accuracy : 0.9804
##                 95% CI : (0.8955, 0.9995)
##   No Information Rate : 0.3333
##   P-Value [Acc > NIR] : < 2.2e-16
##
##                 Kappa : 0.9706
##
##   Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                         Class: Setosa Class: Versicolor Class: Virginica
## Sensitivity           1.0000      1.0000      0.9412
## Specificity           1.0000      0.9706      1.0000
## Pos Pred Value        1.0000      0.9444      1.0000
## Neg Pred Value        1.0000      1.0000      0.9714
## Prevalence            0.3333      0.3333      0.3333
## Detection Rate        0.3333      0.3333      0.3137
## Detection Prevalence  0.3333      0.3529      0.3137
## Balanced Accuracy     1.0000      0.9853      0.9706
  
```

*# Fíjate que, como WEKA, muestra scores por cada clase*

La matriz de confusión, que se puede ver en la figura 1, se muestra de la misma forma que en WEKA, y dejando de lado que el modelo de WEKA ha obtenido mejores resultados, no difieren en nada. También se enseñan las mismas métricas score, salvo que éstos tienen nombres distintos. Así como Sensitivity=TPR o Specificity=FPR.

### ¿Qué efecto tiene el parámetro "tuneGrid"?

Ya se ha explicado el efecto que tiene tuneGrid a la hora de mencionar otros parámetros útiles en train. Éste parámetro permite establecer los hiperparámetros mediante un data frame, a este proceso se le conoce en Caret como “tunear”. Un ejemplo sería el siguiente:

```

# En tuneGrid especificamos el data frame. Se escogen valores impares para
# evitar empates.
data_frame_k <- expand.grid(k = c(1, 3, 5, 15, 19))
KNNModel2 <- train(variety ~ ., data = trainSet, method = "knn", tuneGrid = data_frame_k,
                    trControl = ctrl, preProc = c("scale"))
KNNModel2
  
```

```

Classifier output
Relation: iris
Instances: 150
Attributes: 5
      sepallength
      sepalwidth
      petallength
      petalwidth
      class
Test mode: split 66.0% train, remainder test

==== Classifier model (full training set) ===

IB1 instance-based classifier
using 6 nearest neighbour(s) for classification

Time taken to build model: 0 seconds

==== Evaluation on test split ===

Time taken to test model on test split: 0.01 seconds

==== Summary ===

Correctly Classified Instances      51          100      %
Incorrectly Classified Instances   0           0        %
Kappa statistic                   1
Mean absolute error               0.0335
Root mean squared error          0.0913
Relative absolute error          7.4595 %
Root relative squared error     19.1117 %
Total Number of Instances        51

==== Detailed Accuracy By Class ===

      TP Rate  FP Rate  Precision  Recall  F-Measure  MCC  ROC Area  PRC Area  Class
      1.000   0.000   1.000    1.000   1.000    1.000  1.000   1.000  Iris-setosa
      1.000   0.000   1.000    1.000   1.000    1.000  1.000   1.000  Iris-versicolor
      1.000   0.000   1.000    1.000   1.000    1.000  1.000   1.000  Iris-virginica
Weighted Avg.   1.000   0.000   1.000    1.000   1.000    1.000  1.000   1.000

==== Confusion Matrix ===

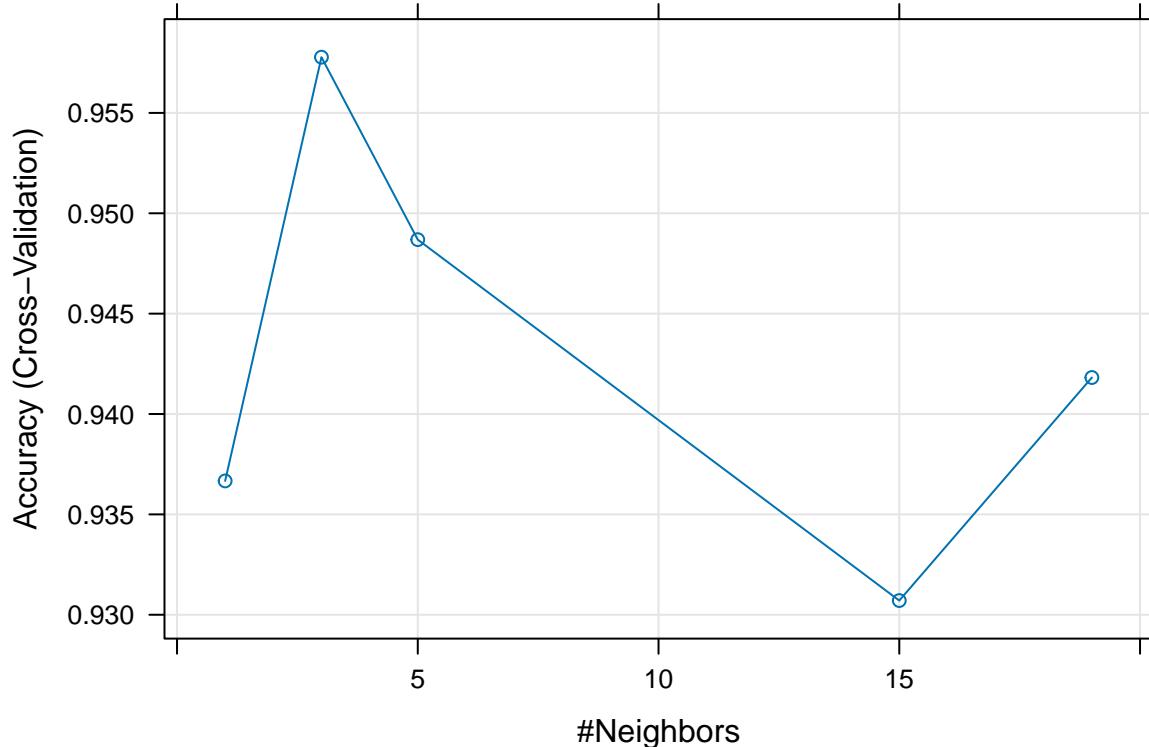
  a  b  c  <-- classified as
22  0  0 |  a = Iris-setosa
  0 14  0 |  b = Iris-versicolor
  0  0 15 |  c = Iris-virginica
  
```

Figure 1: La matriz de confusión hecha en WEKA, con  $k = 9$

```

## k-Nearest Neighbors
##
## 99 samples
## 4 predictor
## 3 classes: 'Setosa', 'Versicolor', 'Virginica'
##
## Pre-processing: scaled (4)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 89, 90, 88, 89, 90, 89, ...
## Resampling results across tuning parameters:
##
##     k   Accuracy   Kappa
##     1   0.9366667  0.9052036
##     3   0.9577778  0.9363567
##     5   0.9486869  0.9226067
##    15   0.9307071  0.8949942
##    19   0.9418182  0.9116608
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 3.

plot(KNNModel2) #Visualización.
  
```



Con los hiperparámetros escogidos, la gran mayoría que se ejecuta, que nos saldrá que el más óptimo es 3:

```

set.seed("1234567890") #Semilla para obtener los mismos datos.
for (i in 1:10) {
  KNNModel2 <- train(variety ~ ., data = trainSet, method = "knn", tuneGrid = expand.grid(k = c(1,
  3, 5, 15, 19)), trControl = ctrl, preProc = c("scale"))
  print("El más optimo es:")
  print(KNNModel2$finalModel$k)
  
```

```
}
```

```

## [1] "El más optimo es:"
## [1] 3
## [1] "El más optimo es:"
## [1] 3
## [1] "El más optimo es:"
## [1] 5
## [1] "El más optimo es:"
## [1] 3
## [1] "El más optimo es:"
## [1] 3
## [1] "El más optimo es:"
## [1] 5
## [1] "El más optimo es:"
## [1] 5
## [1] "El más optimo es:"
## [1] 1
## [1] "El más optimo es:"
## [1] 3
## [1] "El más optimo es:"
## [1] 3

```

Sin embargo, antes se ha visto cómo hay mejores opciones con  $k = 9$  y con  $k = 4$ . Así que al incluir éstos hiperparámetros, deberían eclipsar el resto de los valores, con excepción quizás de algún  $k = 3$  o  $k = 5$  incluso.

```

for (i in 1:10) {
  # Añadimos los hiperparámetros de antes, junto con el quince.
  KNNModel3 <- train(variety ~ ., data = trainSet, method = "knn", tuneGrid = expand.grid(k = c(1,
    5, 15, 19, 4, 9, 3)), trControl = ctrl, preProc = c("scale"))
  print("El más optimo es:")
  # ¿Cuál es el más óptimo de la iteración?
  print(KNNModel3$finalModel$k)
}

## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 4
## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 4
## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 9
## [1] "El más optimo es:"
## [1] 5

```

¿Qué diferencia hay entre "KNNModel1" y "KNNModel2"? ¿Qué parámetro se ha tuneado en el proceso de aprendizaje? ¿Con qué valor definitivo se ha quedado cada modelo?

Así se ha instanciado cada modelo:

```

set.seed("1234567890")
KNNModel1 <- train(variety ~ ., data = trainSet, method = "knn", tuneLength = 5,
  trControl = ctrl, preProcess = c("scale"))
KNNModel2 <- train(variety ~ ., data = trainSet, method = "knn", tuneGrid = expand.grid(k = c(1,
  3, 5, 15, 19)), trControl = ctrl, preProc = c("scale"))
  
```

La mayor diferencia reside en que para el primer modelo dejamos en manos de la función *train* la elección de hiperparámetros, mientras que en el segundo la gracia era que los hiperparámetros se establecían manualmente.

```
KNNModel1$results$k
```

```
## [1] 5 7 9 11 13
```

```
KNNModel2$results$k
```

```
## [1] 1 3 5 15 19
```

Son completamente distintos los valores K en cada modelo. La forma de encontrar el K óptimo con el que se ha quedado cada modelo ya la se ha mencionado. De todas formas, se encuentra en el apartado finalModel, que es, como dice su nombre, el modelo final a emplear.

```
KNNModel1$finalModel$k
```

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

Con el primer modelo se usa el valor  $k = 9$  como final. Se concluye que el primer modelo, intencionadamente o no, cuenta con el mejor valor de  $k$  que se ha podido encontrar en el laboratorio.

```
KNNModel2$finalModel$k
```

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

El valor más óptimo de  $k$  que la función *train* ha conseguido encontrar es con  $k = 3$ . También, se puede llegar a la conclusión de que conforme los  $k$  vayan aumentando, el modelo empeorará cada vez más. Tan solo hace falta los resultados con  $k = 19$ .

```

KNNPredict2 <- predict(KNNModel2, newdata = testSet)
confusionMatrix(KNNPredict2, testSet$variety)
  
```

```

## Confusion Matrix and Statistics
##
##           Reference
## Prediction Setosa Versicolor Virginica
##   Setosa        16         0         0
##   Versicolor     1        16         2
##   Virginica      0         1        15
##
##           Overall Statistics
##
##                 Accuracy : 0.9216
##                               95% CI : (0.8112, 0.9782)
##   No Information Rate : 0.3333
##   P-Value [Acc > NIR] : < 2.2e-16
##
##                 Kappa : 0.8824
##   
```

```

## McNemar's Test P-Value : NA
##
## Statistics by Class:
##
##           Class: Setosa Class: Versicolor Class: Virginica
## Sensitivity          0.9412          0.9412          0.8824
## Specificity          1.0000          0.9118          0.9706
## Pos Pred Value       1.0000          0.8421          0.9375
## Neg Pred Value       0.9714          0.9687          0.9429
## Prevalence           0.3333          0.3333          0.3333
## Detection Rate       0.3137          0.3137          0.2941
## Detection Prevalence 0.3137          0.3725          0.3137
## Balanced Accuracy    0.9706          0.9265          0.9265
  
```

El segundo modelo falla bastantes más veces que cuando se usaba el primer modelo, todo acorde con nuestros cálculos. Nada más que añadir al respecto.

### Ahora por nuestra cuenta, vamos a formar un modelo naïveBayes y un árbol de clasificación.

Ya se ha quemado el dataset “iris.arff” demasiado. Sería adecuado cambiar de conjunto de datos. En este caso, se va a escoger el dataset “precipitation-in-Donostia-2022” [Barrutia, 2022], una colección de datos hecha por un compañero de clase. Como es costumbre, primero se va a explicar y cargar el problema en R.

Para cargarlo, como se trata de un fichero en formato .arff, se usará la librería “foreign”, que tiene implementada una función similar a `read.csv`.

```

library(foreign) #Librería foreign para leer el .arff
# Se carga el fichero.
PrecipitationInDonostia <- read.arff("precipitation-in-Donostia-2022.arff")
# Pese a haber sido leído de un formato distinto, se trata un data frame igual
# que si se hubiera cargado un .csv.
class(PrecipitationInDonostia)
  
```

```

## [1] "data.frame"
  
```

El objetivo del problema es tratar de predecir si llueve o no llueve. Las variables predictoras son las siguientes:

```

names(PrecipitationInDonostia) #Cabecera del data frame.
  
```

```

## [1] "tempC"          "windspeedKmph"   "precip"        "humidity"
## [5] "visibility"     "pressure"       "cloudcover"    "HeatIndexC"
## [9] "DewPointC"      "WindChillC"     "WindGustKmph"  "FeelsLikeC"
## [13] "uvIndex"
  
```

- **tempC.** La temperatura ambiente dada en grados centígrados.
- **winspeedKmph.** La velocidad del viento dada en kilómetros por hora.
- **precip.** Variable a predecir, indica si está lloviendo (1) o no (0).
- **humidity.** El porcentaje de humedad en el ambiente.
- **visibility.** La visibilidad disponible, medido en kilómetros.
- **pressure.** La presión ambiente medida en mb.
- **cloudcover.** El porcentaje de nubosidad en el momento.
- **HeatIndexC.** El índice de calor, medido en grados centígrados.
- **DewPointC.** El punto de rocío en grados centígrados.
- **WindChillC.** La pérdida de calor que provoca el aire frío en movimiento en la piel.

- **WindGustKmph.** La velocidad de las ráfagas de viento en, medido en kilómetros por hora.
- **FeelsLikeC.** La sensación térmica dada en grados centígrados.
- **uvIndex.** El índice de radiación ultravioleta.

Todas estas métricas se miden para un instante dado. Aunque cabe notar que en la cabecera del problema se comunica que cada caso está separado del próximo en tres horas.

Como referencia, éstas son las diez primeras muestras:

```

head(PrecipitationInDonostia, n = 10L) #Como siempre, visualizamos las diez primeras líneas.

##      tempC windspeedKmph precip humidity visibility pressure cloudcover
## 1      10          17     0      67       10     1024          0
## 2      10          17     0      67       10     1024          3
## 3      10          16     0      67       10     1024         100
## 4      10          14     0      67       10     1025         16
## 5      17          10     0      61       10     1025         16
## 6      20          10     0      60       10     1023          8
## 7      15          14     0      69       10     1024          5
## 8      12          15     0      65       10     1024          7
## 9      11          15     0      63       10     1025         9
## 10     11          12     0      64       10     1025          8
##      HeatIndexC DewPointC WindChillC WindGustKmphFeelsLikeC uvIndex
## 1      10          4        7       36       7           1
## 2      10          4        8       36       8           1
## 3      10          4        7       33       7           2
## 4      10          4        8       30       8           3
## 5      17          10       17       19      17           5
## 6      20          12       20       15      20           6
## 7      15          9        14       29      14           1
## 8      12          5        10       32      10           1
## 9      11          5        9        32      9            1
## 10     11          5        10       24      10           1

nrow(PrecipitationInDonostia) #Número de instancias en el data frame.
  
```

```
## [1] 2760
```

Todas son variables numéricas; no hay ninguna variable categórica (no hace falta factorizar). La variable que pretendemos predecir, “precip”, es binaria: o llueve o no llueve.

Antes de crear los modelos respectivos, hay que realizar unas preparaciones al igual que con K-NN.

Por lo tanto, se particiona en conjunto de datos en dos: un 66 % pertenecerá al conjunto de entrenamiento y el resto al conjunto de prueba. Con dicho conjunto de entrenamiento se formará un modelo distinto.

Paso por paso. Primero particionamos nuestro conjunto de datos:

```

training_percentage <- 0.67
trainSetIndexDonostia <- createDataPartition(y = PrecipitationInDonostia$precip,
                                              p = training_percentage, list = FALSE)
  
```

Ahora habrá  $2760 * 0.67 = 1849.2 \approx 1850$  instancias en el conjunto de entrenamiento:

```
nrow(trainSetIndexDonostia)
```

```
## [1] 1850
```

Y por lo tanto  $2760 - 1850 = 910$  instancias en el conjunto de prueba.

Finalmente, se establecen los ajustes de control mediante *train.control*:

```

num_folds <- 20 # Vamos a crear veinte particiones de la colección de datos.  

# Y establecemos que vamos a usar 20-fold cross-validation.  

my_ctrl <- trainControl("cv", number = num_folds)
  
```

Por lo tanto, al hacer veinte folds, internamente en R se harán veinte modelos distintos, más uno, que será el modelo que crea con todo el conjunto de entrenamiento.

### Modelo Naïve Bayes

Después de ver la documentación del paquete Caret, el método a invocar en la función train será “naive\_bayes”:

```

# Entrenamos nuestro modelo Naive Bayes con el paquete Caret, en subset  

# indicamos que usaremos nuestro conjunto de entrenamiento.  

my_naiveBayes <- train(form = precip ~ ., data = PrecipitationInDonostia, method = "naive_bayes",  

  trControl = my_ctrl, preProcess = c("scale"), subset = trainSetIndexDonostia)  

my_naiveBayes
  
```

```

## Naive Bayes  

##  

## 2760 samples  

##   12 predictor  

##     2 classes: '0', '1'  

##  

## Pre-processing: scaled (12)  

## Resampling: Cross-Validated (20 fold)  

## Summary of sample sizes: 1758, 1757, 1757, 1758, 1758, 1758, ...  

## Resampling results across tuning parameters:  

##  

##   usekernel  Accuracy  Kappa  

##   FALSE      0.8393993 0.5989863  

##   TRUE       0.8599462 0.6421356  

##  

## Tuning parameter 'laplace' was held constant at a value of 0  

## Tuning  

##   parameter 'adjust' was held constant at a value of 1  

## Accuracy was used to select the optimal model using the largest value.  

## The final values used for the model were laplace = 0, usekernel = TRUE  

## and adjust = 1.
  
```

Aquí podemos observar dos modelos distintos que se han creado, y con cuál se ha quedado en función de unas métricas. Para cada modelo distinto, se han usado distintos hiperparámetros. Ha habido una “optimización” de los siguientes valores: usekernel, laplace y adjust, se mencionarán más tarde, en el apartado de tuneado.

```

# Creamos el conjunto de prueba.  

testSetDonostia = PrecipitationInDonostia[-trainSetIndexDonostia, ]  

# Ponemos a prueba nuestro modelo.  

my_NaiveBayesPredict <- predict(my_naiveBayes, newdata = testSetDonostia)  

confusionMatrix(my_NaiveBayesPredict, testSetDonostia$precip) #Matriz de confusión.
  
```

```

## Confusion Matrix and Statistics  

##  

##             Reference  

## Prediction   0   1  

##               0 612  40  

##               1  69 189  

##
  
```

```

##           Accuracy : 0.8802
##      95% CI : (0.8573, 0.9006)
## No Information Rate : 0.7484
## P-Value [Acc > NIR] : < 2e-16
##
##           Kappa : 0.6948
##
## McNemar's Test P-Value : 0.00732
##
##           Sensitivity : 0.8987
##      Specificity : 0.8253
## Pos Pred Value : 0.9387
## Neg Pred Value : 0.7326
##      Prevalence : 0.7484
## Detection Rate : 0.6725
## Detection Prevalence : 0.7165
## Balanced Accuracy : 0.8620
##
## 'Positive' Class : 0
##
  
```

El modelo resultante cuenta con una precisión del 87 % para el conjunto de pruebas.

*Pero, ¿se podrán mejorar éstas métricas?*

Se podrá mejorar, si quiera un poco, con el tuneado de los hiperparámetros que hemos mencionado antes. Éstos se puede ver en la siguiente página: <http://topepo.github.io/caret/available-models.html>. Se facilita la información en la siguiente tabla:

Model	Method Value	Type	Libraries	Tuning Parameters
Naive Bayes	naive_bayes	Classification	naivebayes	laplace, usekernel, adjust

Table 1: Los hiperparámetros a tunear en Naïve Bayes.

El que suena es el de Laplace, que se usa para evitar divisiones por cero en casos en los que el modelo no hubiera aprendido una variable. En clase lo hemos conocido como “Rule of Succession” o la aproximación de Laplace.

A parte de ésto, también se da la opción de si se quiere usar KDE (Kernel Distribution Estimation) junto con un bandwith (que se establece con adjust), pero ésto ya se escapa de nuestro conocimiento...

Sin embargo, sirve como curiosidad para saber que de hecho sí se puede mejorar la precisión del modelo:

```

# Establecemos varios valores para los hiperparámetros previamente mencionados.
# Escogemos unos valores arbitrarios para kernel y adjust.
tuneado <- expand.grid(laplace = c(0, 0.5, 1), usekernel = c(FALSE, TRUE), adjust = c(0.5,
  1, 1.5))
my_naiveBayes2 <- train(form = precip ~ ., data = PrecipitationInDonostia, method = "naive_bayes",
  trControl = my_ctrl, preProcess = c("scale"), subset = trainSetIndexDonostia,
  tuneGrid = tuneado)
  
```

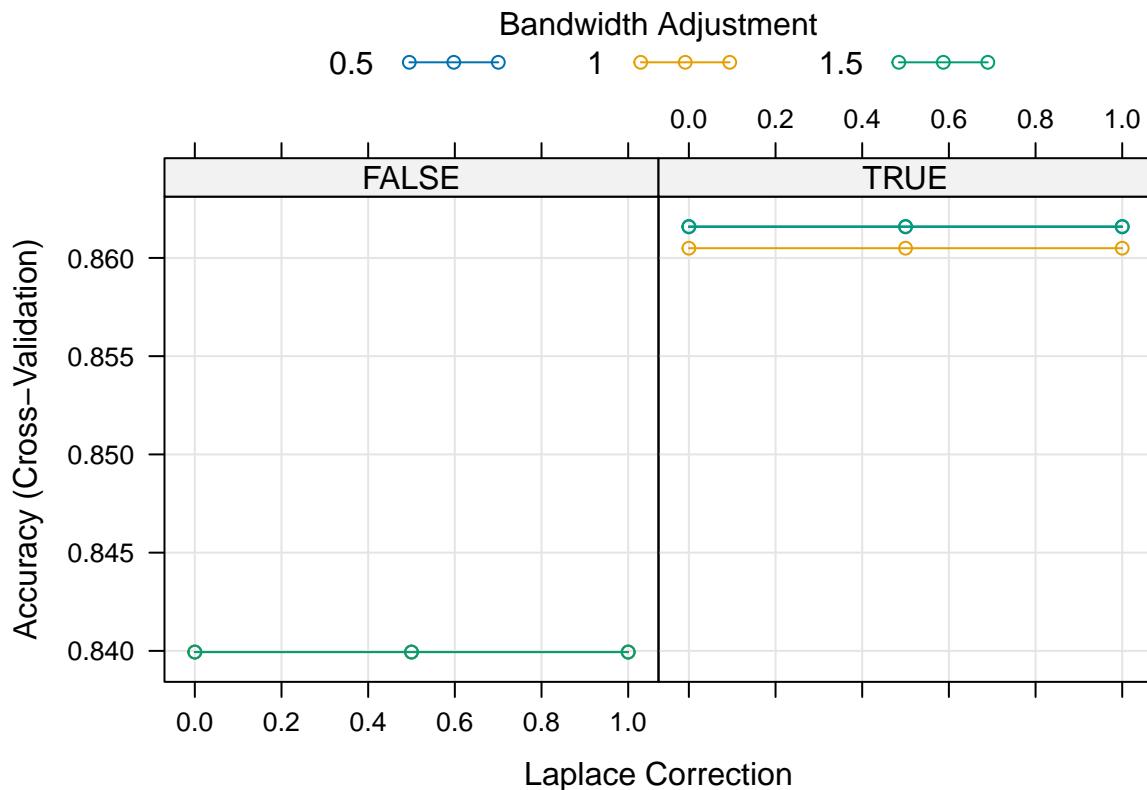
Veamos con qué valores óptimos se ha quedado entre los que le hemos ofrecido:

```
my_naiveBayes2$finalModel$tuneValue
```

```

##   laplace usekernel adjust
## 6       0      TRUE     1.5
  
```

```
plot(my_naiveBayes2)
```



En el gráfico podemos ver que da igual el valor de laplace, que no mejora ni empeora la precisión (y se mantiene en 0), esto será porque en nuestro conjunto de prueba no hay ninguna variable que el modelo de Naive Bayes no haya tenido en cuenta con anterioridad, evitando así la necesidad de Laplace Smoothing.

Comparación de ambos modelos:

```
# Ponemos a prueba el modelo tuneado.  

my_NaiveBayesPredict2 <- predict(my_naiveBayes2, newdata = testSetDonostia)
```

Matriz de confusión para el primer modelo Bayesiano.

```
print(confusionMatrix(my_NaiveBayesPredict, testSetDonostia$precip))
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   0   1
##             0 612  40
##             1  69 189
##
##             Accuracy : 0.8802
##                 95% CI : (0.8573, 0.9006)
##     No Information Rate : 0.7484
##     P-Value [Acc > NIR] : < 2e-16
##
##             Kappa : 0.6948
##
##     Mcnemar's Test P-Value : 0.00732
##
```

```

##           Sensitivity : 0.8987
##           Specificity : 0.8253
##      Pos Pred Value : 0.9387
##      Neg Pred Value : 0.7326
##          Prevalence : 0.7484
##      Detection Rate : 0.6725
## Detection Prevalence : 0.7165
##     Balanced Accuracy : 0.8620
##
##      'Positive' Class : 0
##
  
```

Matriz de confusión para el segundo modelo Bayesiano.

```
print(confusionMatrix(my_NaiveBayesPredict2, testSetDonostia$precip))
```

```

## Confusion Matrix and Statistics
##
##      Reference
## Prediction 0 1
## 0 612 41
## 1 69 188
##
##           Accuracy : 0.8791
##           95% CI : (0.8562, 0.8996)
## No Information Rate : 0.7484
## P-Value [Acc > NIR] : < 2e-16
##
##           Kappa : 0.6916
##
## McNemar's Test P-Value : 0.01004
##
##           Sensitivity : 0.8987
##           Specificity : 0.8210
##      Pos Pred Value : 0.9372
##      Neg Pred Value : 0.7315
##          Prevalence : 0.7484
##      Detection Rate : 0.6725
## Detection Prevalence : 0.7176
##     Balanced Accuracy : 0.8598
##
##      'Positive' Class : 0
##
  
```

En efecto, el segundo modelo es mejor en cuanto a precisión, pero la mejora es pequeña. El tuneado en Naïve Bayes no es tan significativo como lo era con el Nearest Neighbor.

Para terminar, se va a realizar un árbol de clasificación. Esta última parte será más corta ya que el procedimiento es el mismo, y no se ha encontrado mucho que comentar.

## Árboles de clasificación.

En la documentación de Caret, hay varios algoritmos con los que se puede formar un árbol decisión. Entre ellos, se encuentra el algoritmo J48, o el C4.5, que es el que más se ha usado en el contexto de la asignatura.

Como viene siendo costumbre, se invoca la función *train* con el método “C4.5”.

```

library("RWeka")

##
## Attaching package: 'RWeka'

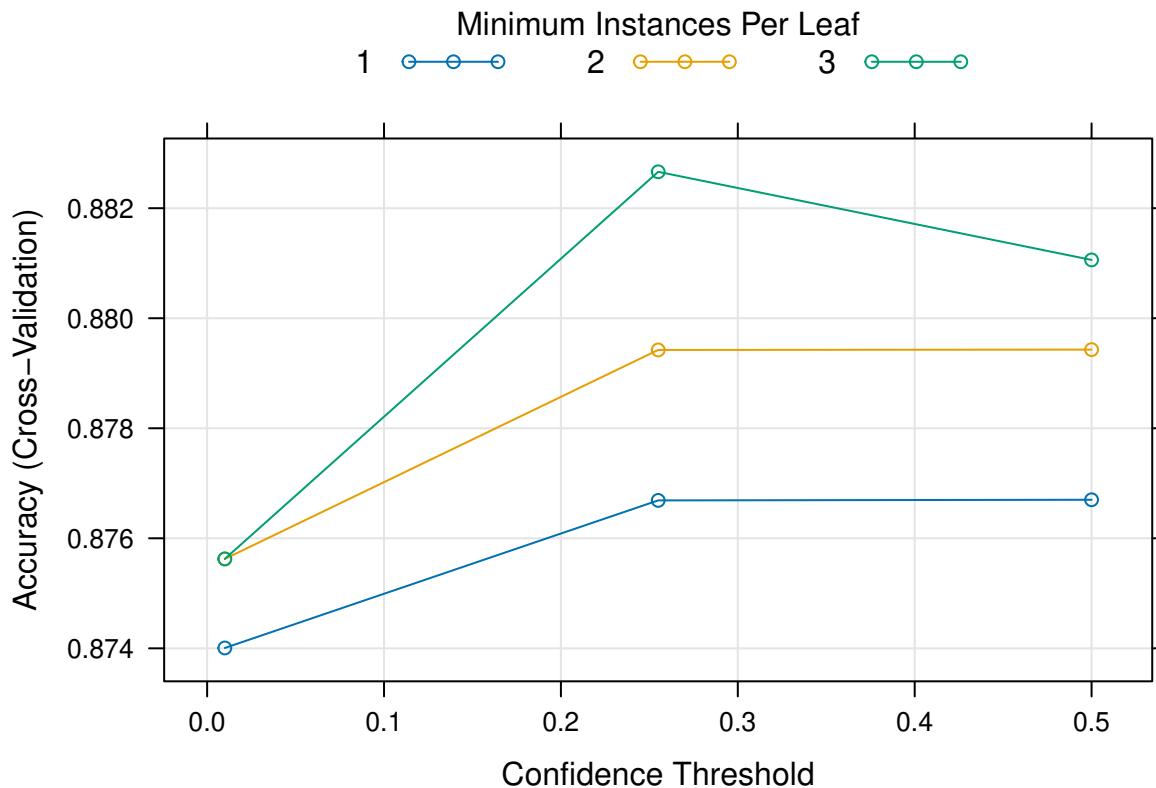
## The following objects are masked from 'package:foreign':
##
##     read.arff, write.arff

my_tree <- train(form = precip ~ ., data = PrecipitationInDonostia, method = "J48",
  trControl = my_ctrl, preProcess = c("scale"), subset = trainSetIndexDonostia)
my_tree

## C4.5-like Trees
##
## 2760 samples
##   12 predictor
##   2 classes: '0', '1'
##
## Pre-processing: scaled (12)
## Resampling: Cross-Validated (20 fold)
## Summary of sample sizes: 1758, 1758, 1758, 1757, 1757, 1758, ...
## Resampling results across tuning parameters:
##
##     C      M Accuracy Kappa
## 0.010  1  0.8740065 0.6686403
## 0.010  2  0.8756253 0.6724251
## 0.010  3  0.8756253 0.6724251
## 0.255  1  0.8766889 0.6776942
## 0.255  2  0.8794238 0.6840552
## 0.255  3  0.8826613 0.6917414
## 0.500  1  0.8767006 0.6732899
## 0.500  2  0.8794296 0.6810466
## 0.500  3  0.8810601 0.6825612
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were C = 0.255 and M = 3.
  
```

Se puede observar que ha ocurrido también un tuneado de variables, ésta vez no es familiar ninguno de los parámetros. Se deja tal y como está.

```
plot(my_tree)
```



Finalmente, se predice para el conjunto de datos prueba y se observa la matriz de confusión.

```
# Ponemos a prueba el modelo.
my_tree_predict <- predict(my_tree, newdata = testSetDonostia)
# Matriz de confusión
confusionMatrix(my_tree_predict, testSetDonostia$precip)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   0     1
##           0 631  59
##           1  50 170
##
##             Accuracy : 0.8802
##             95% CI : (0.8573, 0.9006)
##             No Information Rate : 0.7484
##             P-Value [Acc > NIR] : <2e-16
##
##             Kappa : 0.6778
##
##             Mcnemar's Test P-Value : 0.4435
##
##             Sensitivity : 0.9266
##             Specificity  : 0.7424
##             Pos Pred Value : 0.9145
##             Neg Pred Value : 0.7727
##             Prevalence   : 0.7484
##             Detection Rate : 0.6934
```

```
##      Detection Prevalence : 0.7582
##      Balanced Accuracy : 0.8345
##
##      'Positive' Class : 0
##
```

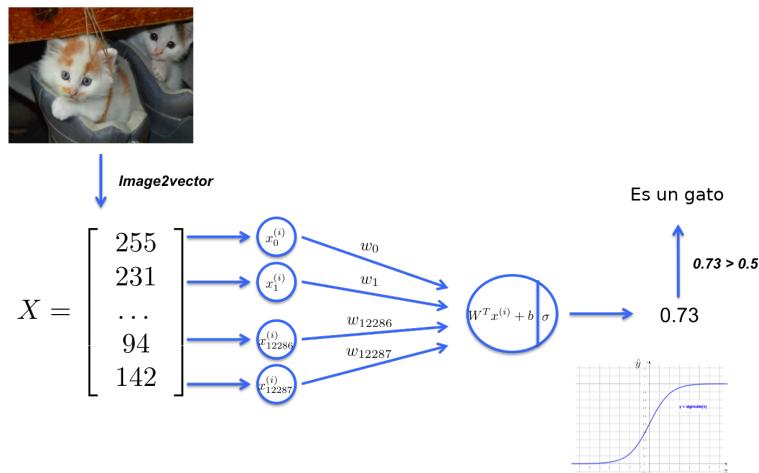
En comparación con el modelo naïve Bayes, ha brindado un mejor resultado, con un 89 % de precisión.

## Décimo laboratorio (22-23 de noviembre)

En éste laboratorio se implementa una clasificación de gatos con regresión logística. Se pide completar los trozos de código, se indicarán **de la siguiente forma**.

### Clasificando gatos con Regresión Logística

En el ejercicio anterior cargamos los datos de unos ficheros con imágenes de gatos. En este segundo ejercicio, aprenderemos un modelo muy simple de regresión logística. Recuerda que ese modelo se puede considerar una versión muy simple de una red neuronal. A continuación, el diagrama que describe el modelo que vamos a implementar:



Para una muestra de datos  $x^{(i)}$ :

$$\begin{aligned} z^{(i)} &= W^T x^{(i)} + b \\ \hat{y} &= a^{(i)} = \sigma(z^{(i)}) \\ (a^{(i)}, y^{(i)}) &= -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \end{aligned}$$

La función de coste se consigue calculando sobre todas las muestras de train:

$$J = \frac{1}{m} \sum_{i=1}^m (a^{(i)}, y^{(i)})$$

Estas son las tareas principales a realizar en este ejercicio:

- Inicializar los parámetros del modelo.
- Aprender los parámetros óptimos del modelo minimizando la función de coste.
- Clasificar las muestras de test con los parámetros aprendidos.
- Analizar los resultados y extraer conclusiones.

### Preparación del entorno.

```

import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage

```

```

def load_dataset():
    train_dataset = h5py.File("train_catvnoncat.h5", "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set
    # features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set
    #labels

    test_dataset = h5py.File("test_catvnoncat.h5", "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

    classes = np.array(test_dataset["list_classes"][:]) # the list of classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig,
    classes

# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()

# Example of a picture
index = 11
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" +
classes[np.squeeze(train_set_y[:, index])].decode("utf-8") + "' picture.")

```

```

m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px_height = train_set_x_orig.shape[1]
num_px_width = train_set_x_orig.shape[2]
num_px = num_px_width

print ("Cantidad de muestras de train: m_train = " + str(m_train))
print ("Cantidad de muestras de test: m_test = " + str(m_test))
print ("Altura de las imágenes: num_px_height = " + str(num_px_height))
print ("Anchura de las imágenes: num_px_width = " + str(num_px_width))
print ("Tamaño de cada imagen: (" + str(num_px_height) + ", " + str(num_px_width)
+ ", 3)")

print ("Estructura de las muestras de train X: " + str(train_set_x_orig.shape))
print ("Estructura de las etiquetas de train Y: " + str(train_set_y.shape))
print ("Estructura de las muestras de test X: " + str(test_set_x_orig.shape))
print ("Estructura de las etiquetas de test Y: " + str(test_set_y.shape))

# Vamos a reestructurar los datos, convirtiendo el cubo que forma una imagen
# [height, width, channels] a un vector de height*width*channel elementos
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

```

```

print ("X_train.shape: " + str(train_set_x_flatten.shape))
print ("Y_train.shape: " + str(train_set_y.shape))
print ("X_test.shape: " + str(test_set_x_flatten.shape))
print ("Y_test.shape: " + str(test_set_y.shape))

print ("Verifica que los datos se han reestructurado correctamente: " +
str(train_set_x_flatten[0:5,0]))

# Para estandarizar los datos, dividiremos el valor de cada pixel por el valor
# maximo
# (255 en este caso).
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
  
```

*Resultado:*

```

Cantidad de muestras de train: m_train = 209
Cantidad de muestras de test: m_test = 50
Altura de las imágenes: num_px_height = 64
Anchura de las imágenes: num_px_width = 64
Tamaño de cada imagen: (64, 64, 3)
Estructura de las muestras de train X: (209, 64, 64, 3)
Estructura de las etiquetas de train Y: (1, 209)
Estructura de las muestras de test X: (50, 64, 64, 3)
Estructura de las etiquetas de test Y: (1, 50)
X_train.shape: (12288, 209)
Y_train.shape: (1, 209)
X_test.shape: (12288, 50)
Y_test.shape: (1, 50)
Verifica que los datos se han reestructurado correctamente: [17 31 56 22 33]
  
```

### Construyendo el algoritmo.

Para entrenar una red neuronal, tenemos que implementar las siguientes líneas:

1. Definir la estructura del modelo (*¿cuántas variables de entrada?*).
2. Inicializar los parámetros del modelo.
3. Loop:
  - Calcular el valor de la función de coste (forward pass).
  - Calcular el valor de los gradientes (backward pass).
  - Actualizar los parámetros (gradient descent).

A continuación, vamos a implementar cada parte de forma separada para juntar todas al final, implementando una red neuronal.

### 1. Funciones ayudantes

Implementa la función sigmoid() ( $\sigma$ ). Más adelante la necesitaremos, para calcular las predicciones  $\sigma(W^T x + b)$ . Recuerda que  $\sigma(z) = \frac{1}{1+e^{-z}}$

Como sabemos, la función sigmoid se usa en problemas de clasificación binaria. El valor que se le pasa lo acotará entre los valores 0 y 1, luego se puede establecer un umbral, el cual si superado, la clase predicha será uno, o cero de lo contrario.

```

# sigmoid funtzioa

def sigmoid(z):
  """
  Calcula el sigmoide de z

  Input:
  z -- Un numero real o un numpy-array de numeros reales.

  Output:
  s -- sigmoid(z)
  """

## PON TU CODIGO AQUI ## (≈ 2 lineas)

nominador = 1 + np.exp(-z)
s = np.divide(1, nominador)

#####
return s

print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))

```

*Resultado:*

```
sigmoid([0, 2]) = [0.5          0.88079708]
```

2. Inicializar los parámetros

**Ejercicio:** Tienes que inicializar los parámetros en el siguiente código. Para eso, crea el array w con ceros. **Pista:** mira la función np.zeros().

```

def initialize_with_zeros(dim):
  """
  Esta funcion crea un vector de ceros de dimensiones (dim, 1), para inicializar w y b.

  Argument:
  dim -- tamaño del vector w.

  Returns:
  w -- vector w de ceros con dimension (dim, 1).
  b -- parametro b inicializado a 0.
  """

## PON TU CODIGO AQUI ## (≈ 2 lineas)

w = np.zeros((dim, 1))
b = 0

```

```
#####
assert(w.shape == (dim, 1))
assert(isinstance(b, float) or isinstance(b, int))

return w, b

dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

### 3. Forward y backward pass

Una vez inicializados los parámetros del modelo, dado un dataset de train, tenemos que implementar las funciones forward y backward que nos permitan aprender los parámetros. **Ejercicio:** Implementa la función de coste y su gradiente en la función propagate().

**Pistas:**

Forward pass:

- Consigue  $X$ .
- Calcula  $A = \sigma(W^T X + b) = (a^{(1)}, a^{(2)}, \dots, a^{(m)})$
- Calcula la función de coste  $J = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Backward pass:

- $\frac{\delta J}{\delta w} = \frac{1}{m} X(A - Y)^T$
- $\frac{\delta J}{\delta b} = \frac{1}{m} \sum_{i=1}^m (a^{(i)} - y^{(i)})$

```
def propagate(w, b, X, Y):
    """
    Calcula la funcion de coste y su gradiente ejecutando el forward y
    el backward pass.

    Input:
    w -- pesos, numpy-array de tamaño (num_px * num_px * 3, 1).
    b -- parametro b (bias).
    X -- Conjunto de muestras X de train. Dimension: (num_px * num_px * 3,
    number of examples)
    Y -- Conjunto de etiquetas Y de train (0 -> no gato, 1 -> gato). Dimension:
    (1, number of examples)

    Output:
    cost -- Coste de la regresion logistica.
    dw -- gradiente de la funcion de coste respecto a w.
    db -- gradiente de la funcion de coste respecto a b.
```

*Pista:*

*- Escribe tu código paso a paso para la propagación. np.log(), np.dot()*  
*"""*

```
m = X.shape[1]

# FORWARD PASS (empezando desde X hasta calcular el coste)
## PON TU CODIGO AQUI ## (≈ 2 líneas de código)

A = sigmoid(np.dot(w.T, X) + b)
cost = -1/m * np.dot(np.log(A), Y.T) + np.dot(np.log(1-A), (1-Y).T)

#####
# BACKWARD PASS (calcula el gradiente)
## PON TU CODIGO AQUI ## (≈ 3 líneas de código)

dz = A - Y
dw = 1/m * np.dot(X, (A-Y).T)
db = 1/m * np.sum(A-Y)

#####
assert(dw.shape == w.shape)
assert(db.dtype == float)
cost = np.squeeze(cost)
assert(cost.shape == ())

grads = {"dw": dw,
         "db": db}

return grads, cost

w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]])
,np.array([[1,0,1]])
grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

*Resultado:*

```
dw = [[0.99845601] [2.39507239]]
db = 0.001455578136784208
cost = -10.19846287286503
```

#### 4. Gradient descent

En este punto ya:

- has inicializado los parámetros,

- eres capaz de calcular la función de coste y el gradiente,
- vas a actualizar los parámetros usando el algoritmo "gradient descent" (descenso de gradiente)

**Ejercicio:** implementa el algoritmo de descenso de gradiente en la función optimize(). Recuerda que el objetivo es aprender los parámetros  $w$  y  $b$  que minimicen la función  $J$ . Para cualquier parámetro  $\theta$ , éste se actualiza segun la regla  $\theta = \theta - \alpha\delta\theta$ , donde  $\alpha$  es la tasa de aprendizaje (learning rate).

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    Esta función optimiza los parámetros w y b para minimizar la función de
    coste siguiendo el algoritmo de gradient descent.
```

*Input:*

$w$  -- pesos, numpy-array de tamaño ( $num\_px * num\_px * 3, 1$ ).  
 $b$  -- parámetro  $b$  (bias).  
 $X$  -- Conjunto de muestras  $X$  de train. Dimension: ( $num\_px * num\_px * 3$ ,  
number of examples)  
 $Y$  -- Conjunto de etiquetas  $Y$  de train (0 -> no gato, 1 -> gato). Dimension:  
(1, number of examples)  
 $num\_iterations$  -- numero de iteraciones del algoritmo.  
 $learning\_rate$  -- tasa de aprendizaje de la regla de actualización.  
 $print\_cost$  -- cuando es True, imprime el valor del gradiente cada 100  
iteraciones.

*Output:*

$params$  -- diccionario que guarda los parametros  $w$  y  $b$ .  
 $grads$  -- diccionario que guarda los gradientes de los parametros  $w$  y  $b$   
respecto a la funcion de coste.  
 $costs$  -- lista donde se guardan los valores de coste. Los usaremos para  
generar una grafica.

*Pistas:*

Tienes que implementar dos pasos e iterar sobre ellos:  
1) Calcular el coste y el gradiente para los parametros actuales.  
Usa propagate()  
2) Actualizar los parametros  $w$  y  $b$  utilizando la regla de actualizacion  
del descenso de gradiente .

"""

```
costs = []
params = {'w':w,
          'b':b}
m = X.shape[1]

for i in range(num_iterations):

    # Calculo del coste y del gradiente (≈ 1 linea de codigo)
    ## PON TU CODIGO AQUI ##

    grads, cost = propagate(w, b, X, Y)

#####
#####
```

```

# Retrieve derivatives from grads
dw = grads["dw"]
db = grads["db"]

## PON TU CODIGO AQUI ## #regla de actualizacion (≈ 3 lineas de codigo)

w == learning_rate*dw
b == learning_rate*db
#####
# Guarda los costes.
if i % 100 == 0:
    costs.append(cost)

# Imprime el coste cada 100 iteraciones.
if print_cost and i % 100 == 0:
    print ("Cost after iteration %i: %f" %(i, cost))

params = {"w": w,
          "b": b}

grads = {"dw": dw,
          "db": db}

return params, grads, costs

params, grads, costs = optimize(w, b, X, Y, num_iterations= 100,
learning_rate = 0.009, print_cost = False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))

```

*Resultados:*

```

w = [[0.19033591] [0.12259159]]
b = 1.9253598300845747
dw = [[0.67752042] [1.41625495]]
db = 0.21919450454067657

```

**Ejercicio:** La función anterior calcula los parámetros  $w$  y  $b$ . La última función que necesitamos es la función predict(). Esta función será capaz de predecir la clase de unas muestras de test  $X$ , dados los parámetros  $w$  y  $b$ . Para ello, tenemos que dar dos pasos:

1. Calcula  $\hat{y} = A = \sigma(W^T X + b)$ .
2. Convierte la predicción a 0 (si  $\hat{y} < 0.5$ ) o a 1 (si  $\hat{y} \geq 0.5$ ). Guarda todas las predicciones en el array Y\_prediction.

```

def predict(w, b, X):
    """

```

*Dados los parametros de la regresion logistica ( $w$ ,  $b$ ), predice las clases (0 o 1) de las muestras.*

*Input:*

$w$  -- pesos, numpy-array de tamaño ( $num\_px * num\_px * 3, 1$ ).

$b$  -- parametro  $b$  (bias)).

$X$  -- Conjunto de muestras  $X$  de test. Dimension: ( $num\_px * num\_px * 3, number\ of\ examples$ )

*Output:*

$Y\_prediction$  -- numpy-array con todas las predicciones obtenidas para las muestras en  $X$  (0/1).

'''

```
m = X.shape[1]
```

```
Y_prediction = np.zeros((1,m))
```

```
w = w.reshape(X.shape[0], 1)
```

```
# Calcula el vector "A", donde tendremos las probabilidades de que cada foto
# contenga un gato.
```

```
## PON TU CODIGO AQUI ## (≈ 1 linea)
```

```
Z = np.dot(w.T, X) + b
= sigmoid(Z)
```

```
#####
for i in range(A.shape[1]):
```

```
# Convierte las probabilidades A[0, i] a predicciones (0/1)
## PON TU CODIGO AQUI ## (≈ 2 lineas)
```

```
if A[0][i] < 0:
    Y_prediction[0][i] = 0
else:
    Y_prediction[0][i] = 1
```

```
#####
assert(Y_prediction.shape == (1, m))
```

```
return Y_prediction
```

```
w = np.array([[0.1124579],[0.23106775]])
```

```
b = -0.3
```

```
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
```

```
print ("predicciones = " + str(predict(w, b, X)))
```

## Construye el modelo y el clasificador

Ya tenemos todos los ingredientes. Ahora usaremos todas las funciones anteriores para aplicar un clasificador de regresión logística a nuestro dataset.

**Ejercicio:** Implementa el modelo usando las funciones anteriores. Utiliza los siguientes nombres para declarar las variables:

- Y\_prediction\_test: predicciones hechas sobre el dataset de test.
- Y\_prediction\_train: predicciones hechas sobre el dataset de train.

```

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,
          learning_rate = 0.5, print_cost = False):
    """
    Implementa el modelo de regresion logisitca usando las funciones implementadas
    anteriormente.

    Input:
    X_train -- numpy-array con el conjunto de muestras X de train.
    Dimension: (num_px * num_px * 3, m_train)
    Y_train -- numpy-array con el conjunto de etiquetas Y de train (0 -> no gato,
    1 -> gato). Dimension: (1, m_train)
    X_test -- numpy-array con el conjunto de muestras X de test.
    Dimension: (num_px * num_px * 3, m_test)
    Y_test -- numpy-array con el conjunto de etiquetas Y de test (0 -> no gato,
    1 -> gato).
    Dimension: (1, m_test)
    num_iterations -- Numero de iteraciones del algoritmo de gradient descent.
    learning_rate -- hyperparametro que representa la tasa de aprendizaje
    (learning rate) de la regla de actualizacion en la función optimize().
    print_cost -- True para imprimir el coste cada 100 iteraciones.

    Output:
    d -- diccionario con la informacion sobre el modelo (coste, predicciones de test,
    predicciones de train, w, b, tasa de aprendizaje y numero de iteraciones).
    """
    ## PON TU CODIGO AQUI ##

    # Inicializa los parametros a 0 (≈ 1 linea)
    w, b = initialize_with_zeros(64 * 64 * 3)

    # Gradient descent (≈ 1 linea)

    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,
                                     learning_rate, print_cost)

    # Extrae los parametros w y b

    w = params["w"]
    b = params["b"]

    # Predice las clases para las muestras de train y de test (≈ 2 lineas)

    Y_prediction_train = predict(w, b, X_train)
  
```

```

Y_prediction_test = predict(w, b, X_test)

#####
# Imprime la metrica de accuracy tanto para train como para test.
print("train accuracy: {}".format(100 - np.mean(np.abs(Y_prediction_train
- Y_train)) * 100))
print("test accuracy: {}".format(100 - np.mean(np.abs(Y_prediction_test
- Y_test)) * 100))

d = {"cost": costs,
      "Y_prediction_test": Y_prediction_test,
      "Y_prediction_train" : Y_prediction_train,
      "w" : w,
      "b" : b,
      "lr" : learning_rate,
      "iterations": num_iterations}

return d

d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000,
learning_rate = 0.005, print_cost = True)

# Vamos a imprimir un ejemplo.
index = 45
plt.figure(1)
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))

print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a \""
+ classes[test_set_y[0,index]].decode("utf-8") + "\" picture.")

# Imprime la grafica del coste (curva de aprendizaje)
costs = np.squeeze(d['cost'])
plt.figure(2)
plt.plot(costs)
plt.ylabel('Coste')
plt.xlabel('Iteraciones (en porcentajes)')
plt.title("Tasa de aprendizaje =" + str(d["lr"]))
plt.show()

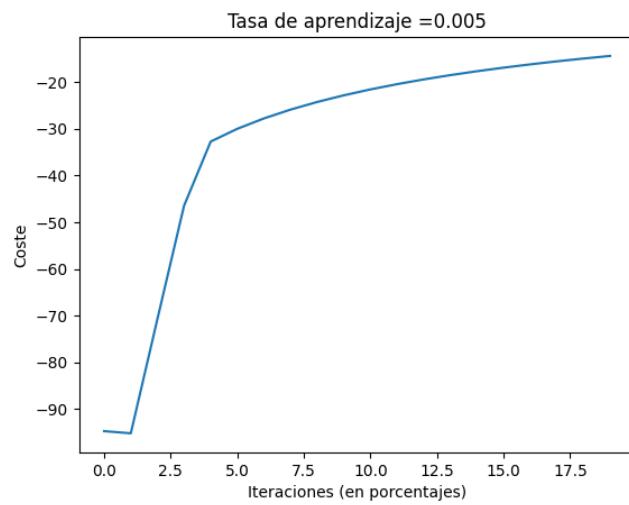
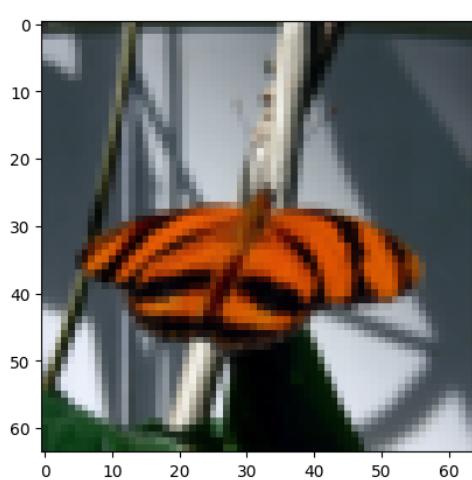
```

*Resultado:*

```

Cost after iteration 0: -94.722376
Cost after iteration 100: -95.183291
Cost after iteration 200: -70.704499
Cost after iteration 300: -46.471559
Cost after iteration 400: -32.737661
Cost after iteration 500: -30.004790
Cost after iteration 600: -27.783839

```



```

Cost after iteration 700: -25.895950
Cost after iteration 800: -24.265025
Cost after iteration 900: -22.837660
Cost after iteration 1000: -21.575000
Cost after iteration 1100: -20.447969
Cost after iteration 1200: -19.434299
Cost after iteration 1300: -18.516604
Cost after iteration 1400: -17.681075
Cost after iteration 1500: -16.916586
Cost after iteration 1600: -16.214046
Cost after iteration 1700: -15.565944
Cost after iteration 1800: -14.966003
Cost after iteration 1900: -14.408927
train accuracy: 34.44976076555024 %
test accuracy: 66.0 %
y = 0, you predicted that it is a "non-cat" picture.
    
```

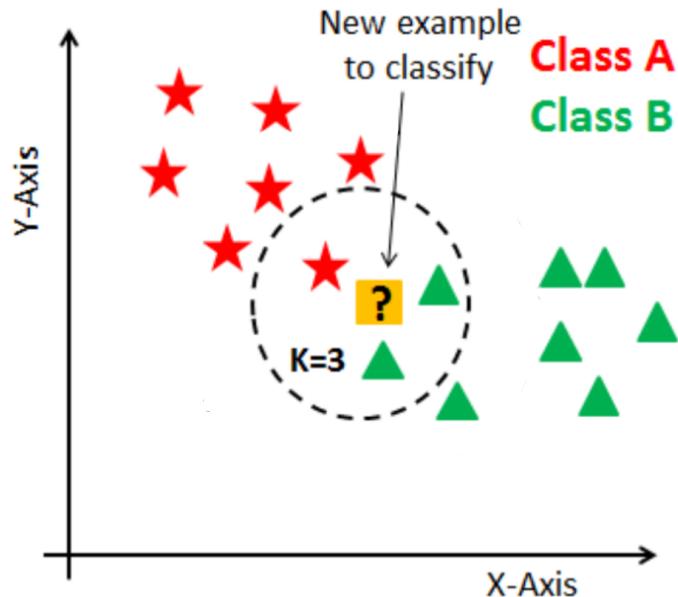
## Undécimo laboratorio (4-5 de diciembre)

Este laboratorio se trata de otro cuaderno en el que se deben implementar lo que requerido. Como la última vez, el código añadido se indicará **de la siguiente forma**.

### K-Nearest Neighbors

Si bien existen muchos modelos de clasificación supervisada, uno de los más conocidos (e intuitivos) es el **K-Nearest Neighbors**, también conocido como **K-NN**. Este modelo de clasificación se basa en las distancias entre individuos del conjunto de datos. Dada cualquier muestra a clasificar  $x$ , el funcionamiento del **K-NN** es el siguiente:

1. Se calcula la distancia entre la muestra  $x$  y todas las muestras del conjunto de entrenamiento. La métrica de distancia utilizada dependerá de la elección del usuario.
2. Se calcula la distancia entre la muestra  $x$  y todas las muestras del conjunto de entrenamiento. La métrica de distancia utilizada dependerá de la elección del usuario.
3. Se predice la clase de  $x$  en base a la clase de los  $K$  vecinos más cercanos. La técnica utilizada dependerá de la elección del usuario.



En la imagen anterior se puede observar un ejemplo de un 3-NN (es decir, un K-NN que solo se fija en los 3 vecinos más cercanos) para un conjunto de datos con solo 2 variables numéricas. En este caso, podemos considerar cada muestra como un punto en un plano de dos dimensiones, y por lo tanto, podemos utilizar la **distancia euclídea** como métrica de distancia. Si nos fijamos en las 3 muestras más cercanas a la muestra a clasificar, podemos observar que encontramos una muestra de la clase A y dos muestras de la clase B. Por lo tanto, utilizando la técnica del **voto mayoritario**, clasificaremos la nueva muestra como perteneciente a la clase más común entre sus vecinos más cercanos, en este caso, la clase B.

Teniendo todo esto en cuenta, en este *notebook* trataremos de programar desde cero un **K-NN** básico, y lo modificaremos iterativamente para mejorar su rendimiento. Finalmente, crearemos un pequeño esquema de **parameter tuning** para seleccionar la mejor configuración de hiper-parámetros para el modelo.

## Preparación del entorno

Lo primero que vamos a hacer será importar los paquetes que necesitaremos para la implementación del *K-NN* (*numpy*, *sklearn*). Además, también importaremos varios paquetes de visualización de datos para poder visualizar mejor los resultados que obtengamos a lo largo de este *notebook* (*seaborn*, *matplotlib*).

```
import numpy as np # Implementación
from sklearn import datasets, model_selection # Implementación
import seaborn as sns # Para visualizar
import matplotlib as mpl # Para visualizar
import matplotlib.pyplot as plt # Para visualizar
```

Una vez importados todos los paquetes, el siguiente paso es cargar y preparar la base de datos sobre la que trabajaremos. En concreto, en este *notebook* consideraremos el **Olivetti Faces data-set** disponible en la librería *sklearn*. Esta base de datos consta de 400 imágenes de caras en blanco y negro distribuidas en 40 clases distintas (numeradas del 0 al 39). Cada imagen tiene un tamaño de 64x64 pixeles, donde el color de cada pixel se representa con un número real entre el 0 (*negro*) y el 1 (*blanco*). Cada una de las clases se asocia con la cara de una persona concreta.

```
X, y = datasets.fetch_olivetti_faces(return_X_y=True) # En X guardamos las
#variables predictoras, en y guardamos las clases.

fig = plt.figure(figsize=(10, 10))
for i in range(8):
    fig.add_subplot(4, 2, i+1)
    plt.imshow(X[5*i].reshape(64, 64), cmap = mpl.cm.gray, interpolation="nearest")
    plt.title("Class "+str(y[5*i]))
    plt.axis("off")
```



A continuación dividiremos el conjunto de datos en un conjunto de entrenamiento (80%) y un conjunto de test (20%). Para ello utilizaremos la función `train_test_split` del paquete `sklearn`. Configuraremos la función para que nos devuelva dos conjuntos con una proporción de clases similar (ver parámetro `stratify`).

Se empieza con la primera parte a implementar. La documentación para el parámetro `stratify` de scikit learn es la siguiente:

---

**`sklearn.model_selection.train_test_split`**

**`stratify : array-like, default=None`**

*If not None, data is split in a stratified fashion, using this as the class labels.*

---

Luego se tiene que indicar en el parámetro `stratify` en función de qué se quiere hacer la partición. En este caso se escoge la y.

```

DIVIDE LA BASE DE DATOS EN CONJUNTO DE ENTRENAMIENTO Y TEST
#LA PROPORCIÓN DE CLASES SERÁ SIMILAR EN AMBOS CONJUNTOS

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
stratify=y, test_size=0.2, random_state=8) # Mantenemos la misma proporción de
# clases en cada subconjunto.

n_classes = 40

print ("Cantidad de muestras de train:", len(X_train))
print ("Cantidad de muestras de cada clase en train:",
np.unique(y_train,return_counts=True)[1])
print ("Cantidad de muestras de test:", len(X_test))
print ("Cantidad de muestras de cada clase en test:",
np.unique(y_test,return_counts=True)[1])

```

*Output:*

```

Cantidad de muestras de train: 320
Cantidad de muestras de cada clase en train: [8 8 8 8 8 8 8 8
8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8]
[8]

Cantidad de muestras de test: 80
Cantidad de muestras de cada clase en test: [2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2]
```

## 1. K-NN básico

Como hemos mencionado anteriormente, los píxeles de cada imagen en nuestra base de datos se representan con números reales entre el 0 y el 1. Por lo tanto, nos encontramos ante una base de datos numérica. En estos casos, una elección natural para nuestro **K-NN** es utilizar la conocida **distancia euclídea**. Dadas dos muestras de n variables numéricas  $x = \{(1), x(2), \dots, x(n)\}$  y  $z = \{z(1), z(2), \dots, z(n)\}$ , la distancia euclídea entre ellas se calcula como:

$$d_e(x, z) = \sqrt{\sum_{i=1}^n (x(i) - z(i))^2} \quad (2)$$

Dicho esto, la primera tarea a realizar será crear un primer **K-NN básico** que utilice la distancia euclídea y el voto mayoritario para clasificar las instancias del conjunto de test.

```

###CALCULA LA DISTANCIA EUCLIDEA ENTRE LOS VECTORES NUMÉRICOS A Y B
def euclidean(A,B,*args):

#####AÑADE AQUÍ TU CÓDIGO#####

subtraction = np.subtract(A, B) # Restamos los dos vectores.
power = np.square(subtraction) # Elevamos el vector resultante al cuadrado.
sum = np.sum(power) # Sumamos todos los elementos del vector.
dist = np.sqrt(sum) # Ya hemos obtenido la distancia
return dist

#CALCULA LA CLASE EN FUNCIÓN DE LOS VECINOS (VOTO MAYORITARIO)
def majority(neighbors_classes,*args):

#####AÑADE AQUÍ TU CÓDIGO#####

# PISTA: Cuenta la cantidad de veces que aparece cada clase en los vecinos.
# más cercanos, y quedate con aquella que sea más frecuente.
frequencies = np.unique(neighbors_classes, return_counts=True) # Obtenemos
# las frecuencias de cada clase.
selected_class = np.argmax(frequencies[1]) # Obtenemos
# el índice de la clase más frecuente.
return frequencies[0][selected_class] # Devolvemos la clase más frecuente.

###DEVUELVE LA CLASE DE LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
def get_neighbors(test_row,X_train,y_train,k):
  dists = np.array([euclidean(test_row,train_row) for train_row in X_train])

#####AÑADE AQUÍ TU CÓDIGO#####

#PISTA: La posición i del array dists guarda la distancia entre la muestra
#a clasificar y la muestra i-ésima del conjunto de entrenamiento.
sorted_dists_indexes = np.argsort(dists) # Ordenamos las distancias.
neighbors_classes = y_train[sorted_dists_indexes[:k]] # Devolvemos
# los índices hasta k de y_train.

return neighbors_classes

###CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k):
  predictions = np.empty(len(X_test),dtype=y_train.dtype)
  for test_ind in range(len(X_test)):
    ###ENCUENTRA LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
    test_row = X_test[test_ind]
    neighbors_classes = get_neighbors(test_row,X_train,y_train,k)
    ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS (VOTO MAYORITARIO)
    predictions[test_ind] = majority(neighbors_classes)
  return predictions

```

Ahora que hemos implementado una primera versión de nuestro **K-NN**, vamos comprobar cómo funciona a la hora de predecir la identidad de las imágenes de test. Para ello, implementaremos dos funciones adicionales:

- Una función simple que nos calcule el **accuracy** del modelo en base a las clases predichas y reales.
- Una función que nos visualice en forma de mapa de calor **la matriz de confusión** de las predicciones realizadas. Para ello, utilizaremos la función heatmap del paquete seaborn.

Utilizaremos las funciones implementadas para comprobar el rendimiento del modelo con  $K = 5$ .

```

#DEVUELVE EL ACCURACY EN BASE A LAS PREDICCIONES Y LAS CLASES REALES
def accuracy(pred,real):

  #####AÑADE AQUÍ TU CÓDIGO#####

  aciertos = np.equal(pred, real) # Obtenemos un vector de tamaño n, en cada cas
  # lla guardaremos True si coinciden, False de otra manera.
  num_aciertos = np.count_nonzero(aciertos) # Contamos las veces que coinciden.
  accuracy = num_aciertos / len(real)
  return accuracy

#MUESTRA LA MATRIZ DE CONFUSIÓN EN FORMA DE HEATMAP
def confusion_matrix(pred,real,n_classes):
  conf_mat = np.zeros((n_classes,n_classes))
  #CONSTRUYE LA MATRIZ DE CONFUSIÓN

  #####AÑADE AQUÍ TU CÓDIGO#####

  #PISTA: La posición (i,j) de la matriz de confusión deberá contener la
  #cantidad de veces que una muestra con clase real j ha sido clasificada
  #como clase i.

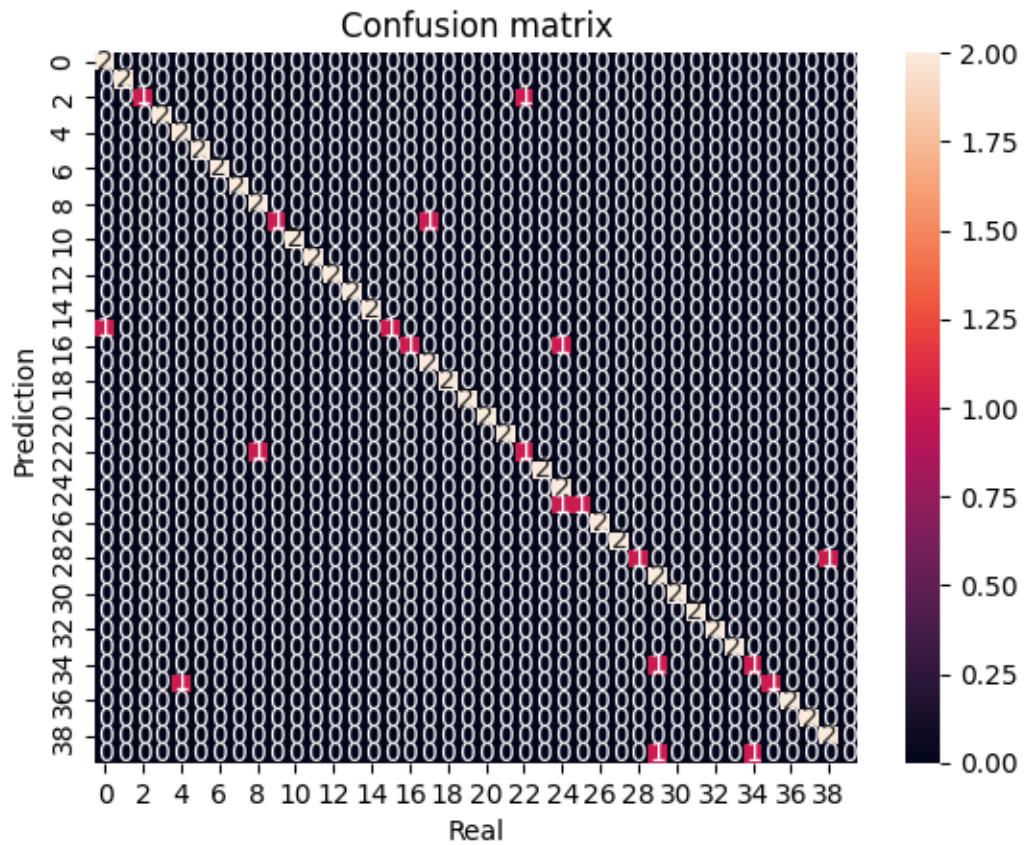
  np.add.at(conf_mat, (real, pred), 1)

#CREA Y MUESTRA EL HEATMAP BASADO EN LA MATRIZ DE CONFUSIÓN
sns.heatmap(conf_mat,annot=True,fmt="g")
plt.title("Confusion matrix")
plt.ylabel("Prediction")
plt.xlabel("Real")
plt.show()

pred = k_nearest_neighbors(X_train,X_test,y_train,k=5)
confusion_matrix(pred,y_test,n_classes)
print("Test accuracy:",str(accuracy(pred,y_test)))

```

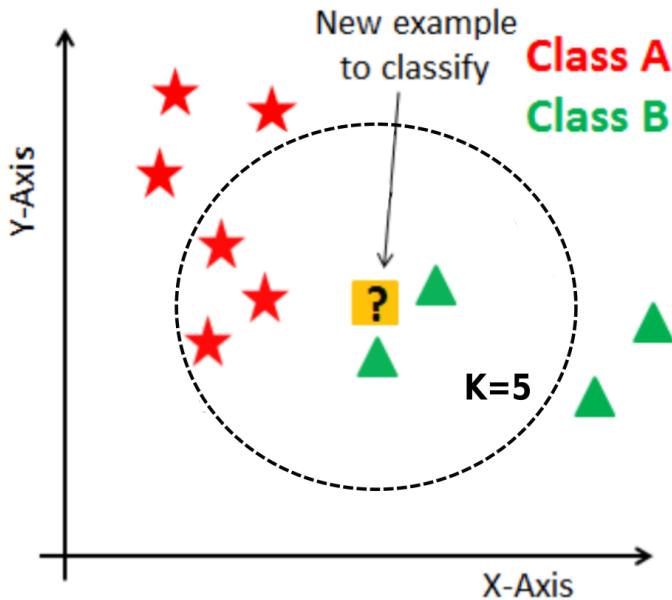
*Output:*



Test accuracy: 0.8625

Como se puede observar, los resultados obtenidos son bastante buenos para un modelo tan simple. Realizando todo de forma correcta, se debería obtener un accuracy cercano al 0.86. Sin embargo, como veremos a continuación, aún hay espacio para la mejora.

## 2. K-NN ponderado según distancia



En la imagen previa se puede observar una situación en la que se utiliza un 5-NN para predecir la clase de una muestra dada. En este ejemplo, solo dos de los cinco vecinos más cercanos pertenecen a la clase *B*, mientras que los otros tres pertenecen a la clase *A*. Siguiendo la estrategia del **voto mayoritario**, el modelo predeciría la clase *A* para la nueva muestra, ya que es la clase más común entre sus vecinos.

Sin embargo, en la imagen se observa que los dos vecinos más cercanos son aquellos que pertenecen a la clase *B*. En este caso, ¿No deberíamos dar más importancia a aquellas muestras que estén más cerca? ¿Es justo darles la misma importancia a los *K* vecinos más próximos, sin considerar la distancia a la que se encuentran?

Si bien no hay una respuesta universal para estas preguntas, considerar la distancia a los vecinos más cercanos a la hora de hacer la predicción puede ser una buena opción en muchas situaciones, sobre todo cuando las muestras se encuentran cerca de la frontera de decisión entre clases. Una forma de hacer esto es utilizar un **voto mayoritario ponderado** a la hora de seleccionar la clase a predecir. En esta estrategia, la importancia de los vecinos va decreciendo según aumenta la distancia con respecto a la muestra a clasificar. Es decir, dada un muestra  $x$ , un conjunto de vecinos más cercanos  $\{z_1, z_2, \dots, z_K\}$  y sus respectivas clases  $\{y_1, y_2, \dots, y_K\}$ , la clase de  $x$  se predice como:

$$y' = \underset{c_i \in C}{\operatorname{argmax}} \sum_{j=1}^K \left( \frac{\delta_{c_i y_j}}{d(x, z_j)} \right) \quad (3)$$

donde  $\{c_1, c_2, \dots, c_m\}$  es el conjunto de todas las posibles clases y  $d(x, z)$  es la distancia entre las muestras  $x$  y  $z$ . La expresión  $\delta_{cy}$  es 1 si la condición  $c = y$  es cierta, mientras que es 0 en caso contrario. Si nos fijamos en la anterior ecuación, observaremos que la contribución de cada vecino a la votación es inversamente proporcional a su distancia con respecto a la muestra a clasificar. Por lo tanto, en este tipo de **K-NN ponderado** no todos los vecinos tienen la misma relevancia.

Teniendo esto en cuenta, vamos a implementar un **K-NN** que acepte distintos tipos de estrategias de selección de clase en base a los vecinos más cercanos. Para ello, deberemos modificar el **K-NN** del ejercicio anterior de la siguiente forma:

- Modificar la función `get_neighbors` de forma que nos devuelva no solo los vecinos más cercanos, sino la distancia a la que se encuentran.
- Modificar la función `k_nearest_neighbors` de forma que acepte un parámetro extra llamado `selection`, el cuál nos indicará qué estrategia utilizar para elegir la clase en base a los vecinos más cercanos.
- Añadir una función `weighted_majority` que implemente el voto mayoritario ponderado.

```
#CALCULA LA CLASE EN FUNCIÓN DE LOS VECINOS (VOTO MAYORITARIO PONDERADO)
def weighted_majority(neighbors_classes,neighbors_dists):
    #####AÑADE AQUÍ TU CÓDIGO#####
    #PISTA: Fíjate en la clase de cada uno de los vecinos más cercanos,
    #y pondera cada caso según la distancia a la muestra a clasificar.
    #...

    pesos_ponderados = np.reciprocal(neighbors_dists)
    clases_ponderadas = np.zeros(np.max(neighbors_classes) + 1)
    np.add.at(clases_ponderadas, neighbors_classes, pesos_ponderados)
    selected_class = np.argmax(clases_ponderadas)

    return selected_class

###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
def get_neighbors(test_row,X_train,y_train,k):
    #####AÑADE AQUÍ TU CÓDIGO#####
    #PISTA: Copia y modifica el código implementado en el ejercicio anterior de
    #forma que la función devuelva la distancia a la que se encuentran los vecinos
    #más cercanos.
    #...

    all_dists = np.array([euclidean(test_row,train_row) for train_row in X_train])
    sorted_dists_indexes = np.argsort(all_dists) # Ordenamos las distancias.
    neighbors_dists = all_dists[sorted_dists_indexes[:k]]
    neighbors_classes = y_train[sorted_dists_indexes[:k]] # Devolvemos los índices
    # hasta k de y_train.

    return neighbors_classes, neighbors_dists

###CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k,selection):
    predictions = np.empty(len(X_test),dtype=y_train.dtype)
    for test_ind in range(len(X_test)):
        ###ENCUENTRA LOS K VECINOS MÁS CERCANOS (DISTANCIA EUCLIDEA)
        test_row = X_test[test_ind]
        neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,y_train,k)
        ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
        predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
    return predictions
```

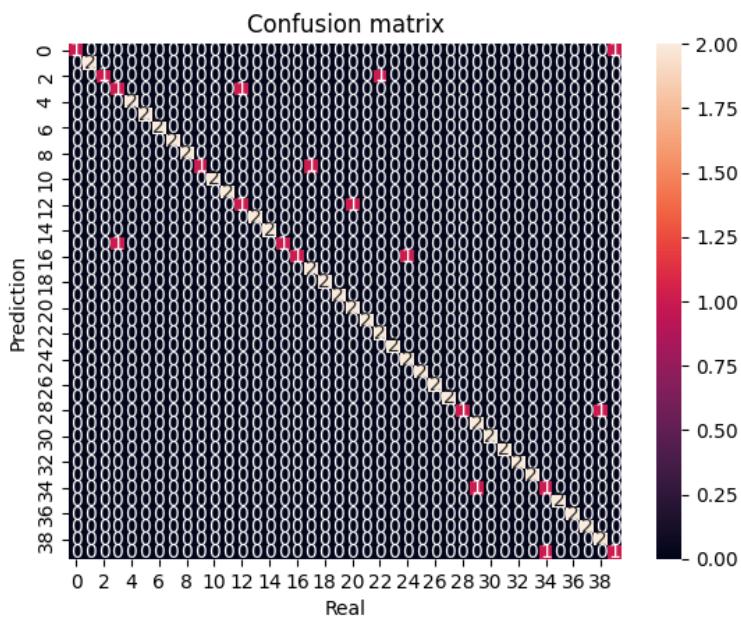
A continuación, comprobaremos los resultados que se obtienen en el conjunto de test si utilizamos la estrategia del voto mayoritario ponderado. Al igual que en el anterior ejercicio, nos fijaremos solo en los 5 vecinos más próximos.

*#INDICA QUE LA SELECCIÓN DE CLASE EN BASE A LOS VECINOS MÁS CERCANOS SE REALIZARÁ A TRAVÉS DE UN VOTO MAYORITARIO PONDERADO*

*#PISTA: En Python, se pueden pasar funciones como argumentos de otra función.*

```
pred = k_nearest_neighbors(X_train,X_test,y_train,k=5,selection='weighted_majority')
#Ponemos en selection "weighted_majority", de esta se reemplazará selection por ése.

confusion_matrix(pred,y_test,n_classes)
print("Test accuracy:",str(accuracy(pred,y_test)))
```



Test accuracy: 0.875

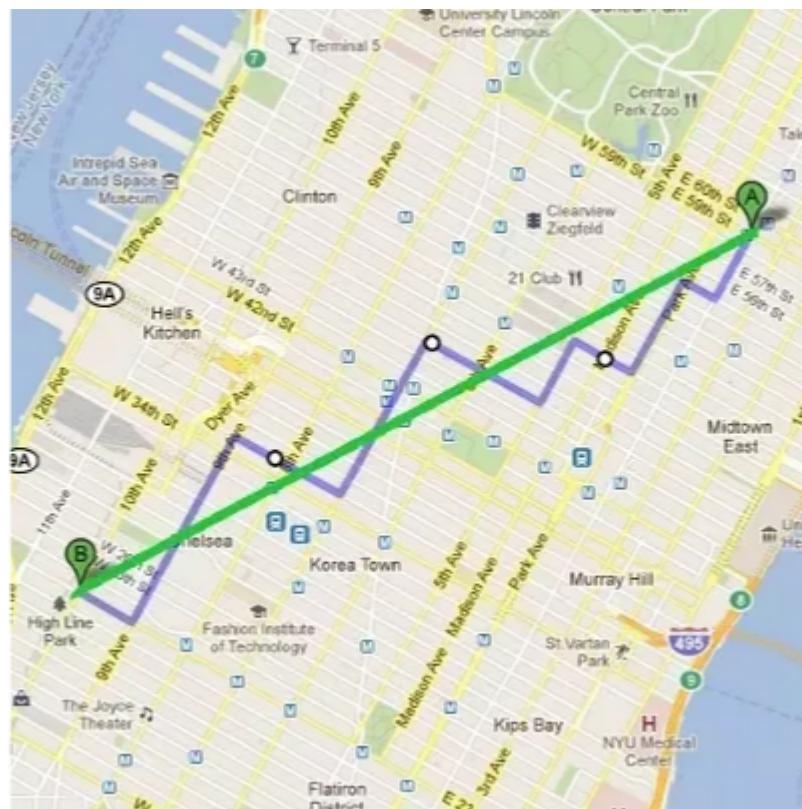
Como se puede observar, los resultados obtenidos son algo mejores que en el caso del voto mayoritario. En concreto, se obtienen valores de accuracy de alrededor de 0.87, lo que implica una subida de 0.01 con respecto al anterior ejercicio.

### 3. Distancias alternativas

Hasta ahora, los **K-NN** que hemos implementado para nuestra base de datos se han basado siempre en la **distancia euclídea**. Sin embargo, esta no es la única opción. Por ejemplo, si los datos fueran categóricos en lugar de numéricos, nos veríamos obligados a utilizar otras métricas de distancia, tales como la **distancia de Jaccard** o la **distancia de Hamming**.

Incluso con datos numéricos, existen multitud de alternativas a la distancia euclídea. De hecho, existen otras métricas que pueden ser incluso más adecuadas cuando la dimensionalidad del problema es alta (como en nuestro caso). Una de las más conocidas y utilizadas es la **distancia de Manhattan**. Dadas dos muestras de  $n$  variables numéricas  $x = \{x(1), x(2), \dots, x(n)\}$  y  $z = \{z(1), z(2), \dots, z(n)\}$ , la distancia de Manhattan entre ellas se calcula como:

$$d_m(x, z) = \sum_{i=1}^n |x(i) - z(i)| \quad (4)$$



La imagen anterior muestra visualmente la diferencia entre la distancia euclídea (*verde*) y la de Manhattan (*azul*) para dos puntos en un plano de dos dimensiones (en este ejemplo, un mapa). La distancia euclídea calcula la distancia más corta entre ambos puntos, mientras que la distancia de Manhattan calcula la suma de las diferencias en todas las dimensiones (o variables).

En este ejercicio, vamos a mejorar nuestro **K-NN** para que acepte distintas métricas de distancia. Para ello, haremos lo siguiente:

- Modificar las funciones *get\_neighbors* y *k\_nearest\_neighbors* para que acepten un nuevo argumento *distance* que nos indicará la métrica de distancia a utilizar.
- Añadir una nueva función *manhattan* que implemente la distancia de Manhattan. Recordemos que la distancia euclídea ya la implementamos en el primer ejercicio (función *euclidean*)

```
#CALCULA LA DISTANCIA DE MANHATTAN ENTRE LOS VECTORES NUMÉRICOS A Y B
```

```
def manhattan(A,B,*args):
```

```
#####AÑADE AQUÍ TU CÓDIGO#####
```

```
#...
```

```
subtractions = np.abs(np.subtract(A, B))
```

```
dist = np.sum(subtractions)
```

```
return dist
```

```
###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS
```

```
def get_neighbors(test_row,X_train,y_train,k,distance):
```

```
#####AÑADE AQUÍ TU CÓDIGO#####
```

*#PISTA: Copia y modifica el código implementado en el ejercicio anterior de forma que la función admita distintas métricas de distancia.*

#...

```

all_dists = np.array([distance(test_row,train_row) for train_row in X_train])
# En vez de llamada a euclidean, llamada a distance.
sorted_dists_indexes = np.argsort(all_dists) # Ordenamos las distancias.
neighbors_dists = all_dists[sorted_dists_indexes[:k]] # Obtenemos las
# distancias de los k más cercanos.
neighbors_classes = y_train[sorted_dists_indexes[:k]] # Devolvemos los índices
# hasta k de y_train.

return neighbors_classes, neighbors_dists

```

###CLASIFICADOR K-NN

```

def k_nearest_neighbors(X_train,X_test,y_train,k,selection,distance):
  predictions = np.empty(len(X_test),dtype=y_train.dtype)
  for test_ind in range(len(X_test)):
    #####ENCUENTRA LOS K VECINOS MÁS CERCANOS
    test_row = X_test[test_ind]
    neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,
    y_train,k,distance)
    #####CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
    predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
  return predictions

```

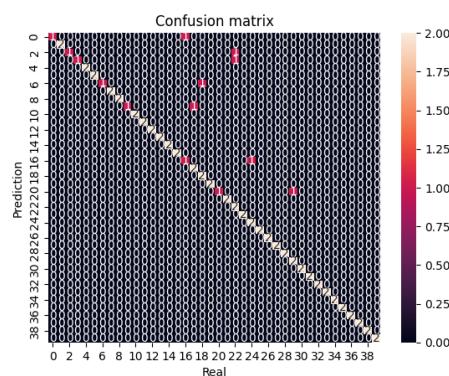
Finalmente, probemos cómo funciona nuestro **K-NN** con la distancia de Manhattan y el voto mayoritario ponderado implementado en el ejercicio anterior. Como hasta ahora, consideraremos  $K = 5$ .

```

pred = k_nearest_neighbors(X_train,X_test,y_train,k=5,selection=weighted_majority,
distance=manhattan)

confusion_matrix(pred,y_test,n_classes)
print("Test accuracy:",str(accuracy(pred,y_test)))

```



Test accuracy: 0.9125

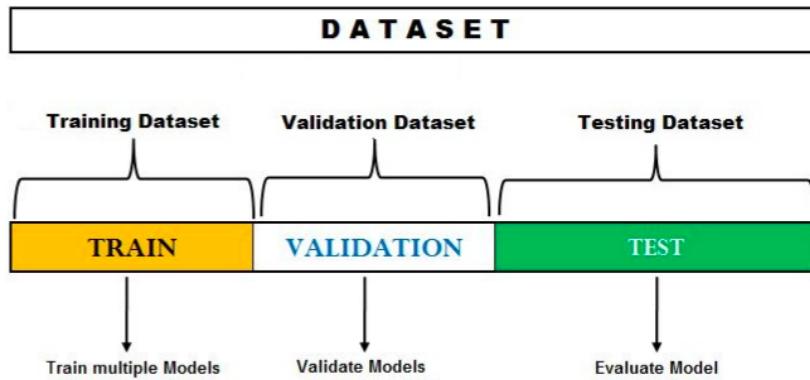
Con solo cambiar la métrica de distancia, hemos conseguido aumentar el accuracy del modelo hasta alrededor de 0.91. Por lo tanto, elegir la distancia correcta para cada problema es crucial a la hora de crear un buen modelo **K-NN**.

#### 4. Parameter tuning

Como hemos visto hasta el momento, parece que elegir una buena configuración de hiper-parámetros (distancia, método de selección de clase...) resulta vital para obtener un modelo con un buen rendimiento. Esto no ocurre solo en el **K-NN**, sino que cualquier modelo de clasificación con hiper-parámetros necesita ser ajustado de antemano. Este proceso de elegir la mejor configuración posible para un modelo es lo que se conoce como **parameter tuning**.

Si bien hasta ahora hemos estado probando las distintas versiones del **K-NN** sobre el conjunto de test, esto no debe hacerse a la hora de realizar el parameter tuning. El conjunto de test solo debe de utilizarse para comprobar el rendimiento del modelo final, lo que nos dará una idea de cómo *generaliza* nuestro clasificador al presentarle datos nunca antes vistos. Si utilizásemos el conjunto de test para hacer el parameter tuning, los resultados que obtendríamos sobre dicho conjunto estarían sesgados debido a nuestras decisiones, por lo que las métricas de generalización no serían fiables. Este fenómeno es lo que se conoce como *data leaking*. Por lo tanto, la elección de los hiper-parámetros debe de ser realizada tomando en cuenta solo el conjunto de entrenamiento.

Para ello, en este ejercicio dividiremos el conjunto de entrenamiento en dos porciones diferenciadas, y utilizaremos una de ellas como **conjunto de validación**. Es decir, crearemos los modelos considerando solo una porción de las muestras de entrenamiento, y utilizaremos el resto para probar el rendimiento de las distintas configuraciones de hiper-parámetros. Por lo tanto, nuestra base de datos quedará particionada de la siguiente forma.



El parameter tuning que implementaremos se centrará en optimizar la cantidad de vecinos a considerar ( $K$ ), y supondremos que tanto la distancia como el método de selección de clase ya han sido definidos de antemano por un experto. En concreto, utilizaremos la **distancia de Manhattan** y el **voto mayoritario ponderado** implementados en ejercicios anteriores.

El parámetro  $K$  será elegido mediante una estrategia **grid search**. Este método se basa en definir un conjunto de configuraciones de hiper-parámetros que se evalúan una por una sobre el conjunto de validación. Después, se comparan los resultados obtenidos por cada configuración, y se seleccionan aquellos hiper-parámetros que obtengan los mejores resultados. Es importante destacar que esta estrategia no prueba todas las posibles combinaciones de los hiper-parámetros, sino que tendremos que decidir de antemano qué valores queremos probar.

Teniendo todo esto en cuenta, vamos a implementar un parameter tuning basado en grid search para optimizar el parámetro  $K$  de nuestro **K-NN**. También incluiremos una visualización de los valores de accuracy obtenidos en el conjunto de validación por las distintas configuraciones del hiper-parámetro  $K$  (función plot del paquete matplotlib).

```

#CREA Y MUESTRA LA GRÁFICA QUE REPRESENTA EL ACCURACY EN VALIDACIÓN OBTENIDO
#POR LOS DISTINTOS VALORES DE K
def plot_tuning(k_values,k_acc):
    plt.plot(k_values,k_acc,"o-")
    plt.ylabel("Validation Accuracy")
    plt.xlabel("k")
    plt.xticks(k_values)
    plt.show()

#REALIZA EL PARAMETER TUNING DEL PARÁMETRO K DEL K-NN
def parameter_tuning(X_train,y_train,validation_split,k_values):
    #DIVIDE EL CONJUNTO DE ENTRENAMIENTO PARA OBTENER EL CONJUNTO DE VALIDACIÓN
    X_train_split, X_validation, y_train_split, y_validation =
        model_selection.train_test_split(X_train, y_train, stratify=y_train,
                                         test_size=validation_split, random_state=8)
    #EVALUA LOS DISTINTOS MODELOS EN EL CONJUNTO DE VALIDACIÓN
    k_acc = np.zeros(len(k_values))

    #####AÑADE AQUÍ TU CÓDIGO#####
    #PISTA: La posición i del array k_acc deberá contener el accuracy de validación
    #del modelo K-NN con el valor de K indicado en la posición i del array k_values.
    #Recuerda que los modelos deberán utilizar la distancia de manhattan y el voto
    #mayoritario ponderado.
    #...

    i = 0
    for k in k_values:
        pred = k_nearest_neighbors(X_train_split,X_validation,y_train_split,k=k,
                                   selection=weighted_majority,distance=manhattan)
        k_acc[i] = accuracy(pred, y_validation)
        i += 1

plot_tuning(k_values,k_acc)
#ELIGE EL VALOR DE K CON EL QUE SE HAN OBTENIDO MEJORES RESULTADOS

#####AÑADE AQUÍ TU CÓDIGO#####
#PISTA: Deberás devolver tanto el mejor accuracy de validación como el valor
#de K para el que se haya obtenido dicho resultado.
#...

best_k_index = np.argmax(k_acc)
max_acc = k_acc[best_k_index]
best_k = k_values[best_k_index]

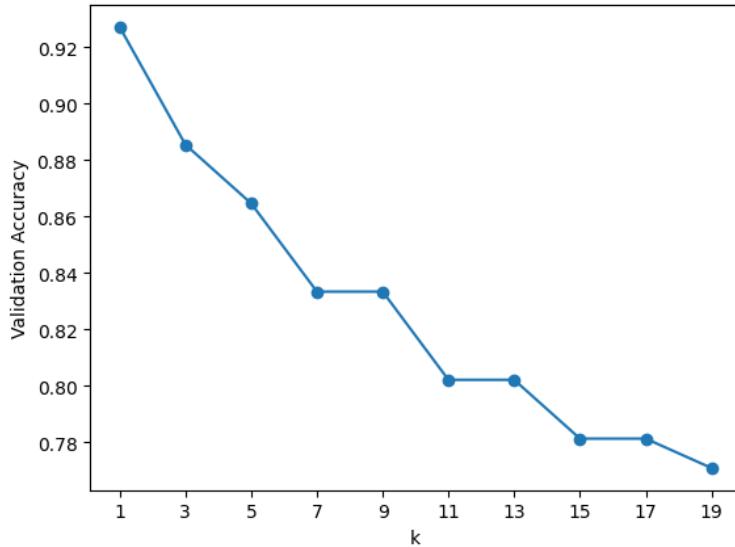
return max_acc,best_k

```

A continuación, probemos nuestro parameter tuning. Para ello, utilizaremos un conjunto de validación compuesto por el 30% de las muestras de entrenamiento. En cuanto a los valores de  $K$  a considerar, probaremos con: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19].

```

k_values = np.array([1,3,5,7,9,11,13,15,17,19])
max_acc, best_k = parameter_tuning(X_train,y_train,0.3,k_values)
print("Best validation accuracy:",max_acc, " , Best k:",best_k)
  
```



Best validation accuracy: 0.9270833333333334 , Best k: 1

Al parecer, el modelo obtiene los mejores resultados con  $K = 1$ . De hecho, la gráfica debería mostrar que el accuracy en el conjunto de validación decrece según aumenta el valor de  $K$ . Esto tiene sentido ya que la cantidad de muestras por cada clase en nuestra base de datos es muy pequeña, por lo que considerar muchos vecinos podría llevar a error.

Ahora sí, una vez que hemos decidido la configuración de los hiper-parámetros para el modelo, podemos comprobar su rendimiento sobre el conjunto de test. En este punto consideraremos el conjunto de entrenamiento entero para construir el modelo, incluyendo las instancias que utilizamos para la validación durante el parameter tuning.

```

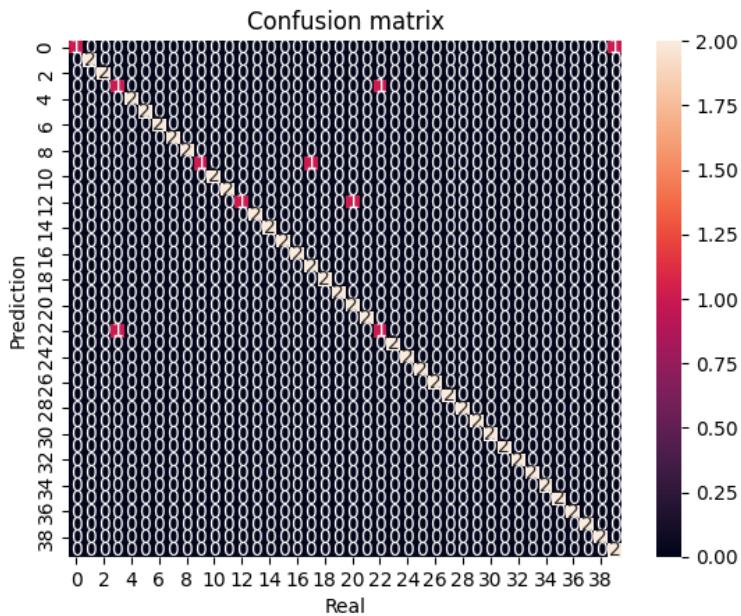
pred = k_nearest_neighbors(X_train,X_test,y_train,k=best_k,
                           selection=weighted_majority,distance=manhattan)

confusion_matrix(pred,y_test,n_classes)
print("Test accuracy:",str(accuracy(pred,y_test)))
  
```

Con el parámetro  $K$  optimizado el accuracy en el conjunto de test aumenta hasta alrededor de 0.94. Este accuracy resulta muy similar al obtenido durante la validación, lo que demuestra que nuestro **K-NN** es capaz de generalizar de forma correcta a muestras nunca antes vistas.

## 5. K-NN con funciones de librería

A lo largo de este *notebook* hemos implementado un **K-NN** que hemos mejorado iterativamente, añadiendo distintas métricas de distancia, métodos de selección de clase, e incluso un parameter tuning para decidir el valor de  $K$ . Todo esto ha sido hecho a mano, utilizando solo funciones de librería básicas como apoyo.



Sin embargo, a la hora de trabajar con este tipo de modelos en el mundo real no hace falta que implementemos todo desde cero. Existen decenas de paquetes que incluyen métodos de clasificación como el **K-NN**, siendo un ejemplo claro el conocido *sklearn*.

Este ejercicio consiste en ejecutar un **K-NN** con las mismas características que el modelo final del anterior apartado utilizando solo funciones del paquete *sklearn*. Para ello, será necesario investigar la documentación de dicho paquete para entender las distintas funciones y sus argumentos. Si todo se realiza correctamente, los resultados en el conjunto de test deberían ser idénticos a los obtenidos por nuestro modelo.

En la documentación de scikilearn se ve cómo se devuelve un objeto de la clase KNeighborsClassifier.

```

from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# DEFINE, CREA Y EVALUA EL MODELO
neigh = KNeighborsClassifier(n_neighbors=1, weights='distance',
                            metric='manhattan', algorithm="brute")

##### AÑADE AQUÍ TU CÓDIGO #####
# PISTA: Crea el modelo utilizando las muestras de train y evalúalo sobre el conjunto
# de test. Utiliza solo funciones de la librería sklearn.
# ...

neigh.fit(X_train, y_train) # Se ajusta el modelo al conjunto de entrenamiento.
pred = neigh.predict(X_test)
print("Test accuracy:", str(accuracy_score(pred, y_test)))

```

Test accuracy: 0.9375

Se puede ver cómo se ha obtenido la misma precisión para ambos modelos.

## EXTRA-1. Variables categóricas

Si bien hasta el momento hemos considerado una base de datos con variables numéricas, el **K-NN** puede ser utilizado para bases de datos categóricas o incluso mixtas. La principal diferencia en estos casos es la métrica de distancia a utilizar, la cuál deberá ser una métrica válida para el tipo de datos sobre el que trabajemos.

En este ejercicio, implementaremos desde cero dos métricas distintas para conjuntos de datos con variables binarias: la **distancia de Hamming** y la **distancia de Jaccard-Needham**. El muy conocido paquete *scipy* incluye este tipo de métricas, por lo que sería una buena idea investigar en su documentación para entender cómo se definen dichas distancias.

La documentación dice lo siguiente:

Dados dos vectores u y v (de una dimensión), la **distancia de Hamming** es la proporción de los componentes no coincidentes entre éstos. Luego,

$$\frac{c_{01} + c_{10}}{n} \quad (5)$$

donde  $c_{ij}$  es el número de veces que  $u[k] = i$  y  $v[k] = j$  se cumple para  $k < n$ .

Es decir, por cada vez que no coincidan las casillas correspondientes entre dos vectores, se suma. Finalmente, se divide por n, el tamaño de los vectores.

Véase el ejemplo que se ofrece en *scipy*, para entender cómo se lleva a cabo y luego poder implementarlo:

```
from scipy.spatial import distance # Se importa la función que implementa hamming

distance.hamming([1, 0, 0], [0, 1, 0])
```

0.6666666666666666

Devuelve 0.6 puesto que el número de números no coincidentes es de 2, en las primeras y segundas casillas. Luego  $\frac{2}{3} = 0.666\ldots$  Se ha hecho una función bastante similar previamente, que servía para contar las veces que una predicción hecha por el modelo es válida o no.

```
#CALCULA LA DISTANCIA DE HAMMING ENTRE LOS VECTORES BINARIOS A Y B
def hamming(A,B,*args):
    count_falses = np.count_nonzero(A != B) # Contar el número de veces
    # que sale False
    dist = count_falses / len(A)
    return dist
```

Una vez más, la documentación de *scipy* es más que suficiente para entender cómo funciona la distancia de **Jaccard-Needham**. Dice lo siguiente:

La **distancia de Jaccard-Needham**, calcula la disimilitud entre dos vectores dados u y v (de una dimensión) de la siguiente manera:

$$\frac{C_{10} + C_{01}}{C_{11} + C_{01} + C_{10}} \quad (6)$$

donde  $c_{ij}$  es el número de veces que  $u[k] = i$  y  $v[k] = j$  se cumple para  $k < n$ .

El numerador es el mismo que en la forma de Hamming; lo que cambia es el denominador: ésta vez

no es el tamaño del vector, sino las instancias en las que los dos elementos correspondientes de cada vector coinciden con el valor 1, más en las que no coinciden.

Se ilustra mejor en el siguiente ejemplo:

```

from scipy.spatial import distance
distance.jaccard([1, 0, 0], [0, 1, 0])

```

1.0

El output es uno porque pese que de la misma forma que en el previo ejemplo los no coincidentes son 2, se divide por: elementos que coinciden con el valor 1 (0) más el denominador, luego  $\frac{2}{2} = 1$ .

El denominador se calcula de la misma forma. Para contar las veces con el valor uno, se añade otra condición.

```

#CALCULA LA DISTANCIA DE JACCARD-NEEDHAM ENTRE LOS VECTORES BINARIOS A Y B
#CONSIDERA QUE 1=TRUE y 0=False
def jaccard(A,B,*args):
    count_falses = np.count_nonzero((A != B))
    equal_one_count = np.count_nonzero((A == 1) & (B == 1)) #Simplemente se
    # indica la coincidición que se quiere contar.
    dist = count_falses / (equal_one_count + count_falses)
    return dist

```

Para probar las métricas que acabamos de implementar, vamos a **discretizar** nuestra base de datos numérica (**Olivetti Faces data-set**) de forma que cada muestra sea un vector binario. Para ello, lo que haremos será convertir todos los valores menores o iguales que 0.5 a 0, y todos los valores mayores que 0.5 a 1. La variable clase no deberá ser modificada.

```

#DISCRETIZA LA BASE DE DATOS SIN MODIFICAR LA VARIABLE CLASE (TANTO EN TRAIN COMO EN
# TEST)

#####AÑADE AQUÍ TU CÓDIGO#####

#Con compresión de listas no hay problema alguno:
X_train = np.array([[0 if val < 0.5 else 1 for val in sublist] for sublist in X_train])
X_test = np.array([[0 if val < 0.5 else 1 for val in sublist] for sublist in X_test])

fig = plt.figure(figsize=(10, 10))
for i in range(8):
    fig.add_subplot(4, 2, i+1)
    plt.imshow(X_train[5*i].reshape(64, 64), cmap = mpl.cm.gray, interpolation="nearest")
    plt.title("Class "+str(y_train[5*i]))
    plt.axis("off")

```



Figure 6: Output

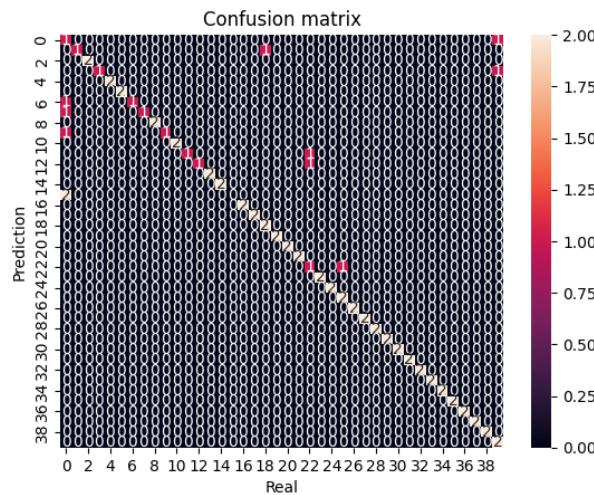
A continuación utilizaremos nuestro **K-NN** con las **distancias de Hamming** y **Jaccard-Needham** para clasificar las instancias de test usando la base de datos discretizada. Nos fijaremos únicamente en el vecino más cercano.

```

pred = k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,
                           distance=hamming)
confusion_matrix(pred,y_test,n_classes)
print("Hamming test accuracy:",str(accuracy(pred,y_test)))

```

*Output:*

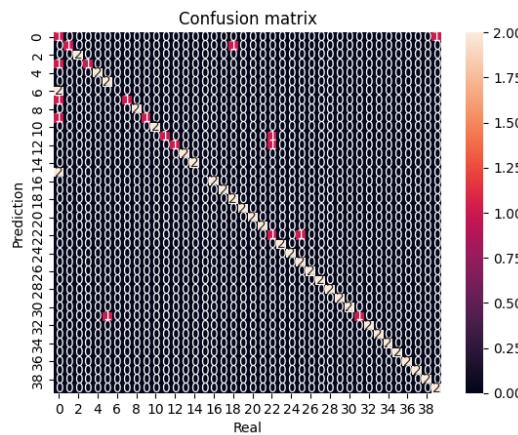


Hamming test accuracy: 0.8625

Con la distancia de Jaccard:

```
pred = k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,
                           distance=jaccard)
confusion_matrix(pred,y_test,n_classes)
print("Jaccard test accuracy:",str(accuracy(pred,y_test)))
```

*Output:*



Jaccard test accuracy: 0.8375

Como se puede observar, obtenemos resultados bastante buenos incluso con la pérdida de información derivada de la discretización. En caso de haber realizado todo correctamente, se deberían obtener valores de accuracy de alrededor de 0.86 en el caso de la **distancia de Hamming** y de alrededor de 0.84 en el caso de la **distancia de Jaccard-Needham**.

## EXTRA-2. Información Mutua

Una de las métricas más utilizadas para medir la correlación entre variables categóricas es la **información mutua**. La información mutua nos indica la **cantidad de información** sobre una variable que nos proporciona el conocer el valor de otra variable. Este tipo de medidas son muy útiles para conocer qué variables predictoras nos dan más información sobre la variable clase a predecir.

Lo primero que vamos a hacer en este ejercicio es crear una función que nos permita calcular la información mutua  $I(X; Y)$  entre dos variables categóricas  $X$  e  $Y$ . Para ello, será necesario buscar y entender la fórmula de esta popular métrica de correlación. Dadas dos variables, calcular la información mutua de  $X$  respecto a  $Y$  se calcula de la siguiente forma:

$$I(X; Y) = H(Y) + H(X|Y) \quad (7)$$

Donde,

$$H(Y) = - \sum_{y \in \mathcal{Y}} P(Y = y) \log_2 P(Y = y) \quad (8)$$

$$H(X, Y) = - \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P(X = x, Y = y) \log_2 P(X = x, Y = y) \quad (9)$$

```

#CALCULA LA ENTROPIA DE SHANNON EN FUNCIÓN DE LAS PROBABILIDADES DE ENTRADA
def shannon_entropy(P_x):

#####AÑADE AQUÍ TU CÓDIGO#####
#...

entropy = -np.sum(P_x * np.log2(P_x))

return entropy

#CALCULA LA INFORMACIÓN MUTUA ENTRE LAS VARIABLES CATEGÓRICAS X E Y
def mutual_information(X,Y):
  length = len(X)

#####AÑADE AQUÍ TU CÓDIGO#####

#PISTA: Dederás calcular las probabilidades necesarias para el cálculo de las
#entropias H(X), H(Y) y H(X,Y).
#...

# Calculamos la información mutua de X e Y

frequencies_X = np.unique(X, return_counts=True)[1]
frequencies_Y = np.unique(Y, return_counts=True)[1]
P_x = frequencies_X / len(X)
P_y = frequencies_Y / len(Y)
H_x = shannon_entropy(P_x)
H_y = shannon_entropy(P_y)
P_XY = np.histogram2d(X,Y,bins=[np.unique(X), np.unique(Y)])[0]
H_xy = shannon_entropy(P_XY)

return H_x + H_y - H_xy

```

Como hemos mencionado anteriormente, la información mutua puede ser muy útil para descubrir cuales son las variables predictoras más relevantes en una base de datos. Esta información puede ser crucial a la hora de calcular las distancias entre muestras en el **K-NN**, ya que nos permite dar más relevancia a aquellas variables con una gran relación con el valor de clase, e ignorar aquellas que solo añadan ruido al proceso de predicción.

Por este motivo, el último paso a realizar será añadir una nueva métrica de distancia para variables categóricas llamada **weighted\_hamming**, la cuál será una versión modificada de la **distorancia de Hamming**. La principal particularidad de esta métrica es que incluirá un *\*pesado de variables predictoras\**. En nuestro caso, dicho pesado será proporcional a la **información mutua** con respecto a la variable clase. Es decir, el peso de cada variable  $X_i$  se calculará de la siguiente forma:

$$w_i = \frac{I(X_i; Y)}{\sum_{j=1}^n I(X_j; Y)} \quad (10)$$

donde  $\{X_1, X_2, \dots, X_n\}$  es el conjunto de todas las variables predictoras e  $Y$  es la variable clase. Los pesos a utilizar en la métrica de distancia se especificarán mediante un nuevo argumento *\*weights\** en la función **k\_nearest\_neighbors**.

```

#CALCULA LA DISTANCIA DE HAMMING ENTRE LOS VECTORES BINARIOS A Y B PONDERADA EN BASE
#a los pesos en W
def weighted_hamming(A,B,W):

#####AÑADE AQUÍ TU CÓDIGO#####

count_falses = np.sum((A != B) * W) # Contar el número de veces
# que sale False
weighted_dist = count_falses / len(A)
return weighted_dist

###DEVUELVE LA CLASE Y DISTANCIA DE LOS K VECINOS MÁS CERCANOS
def get_neighbors(test_row,X_train,y_train,k,distance,weights):

#####AÑADE AQUÍ TU CÓDIGO#####

#PISTA: Copia y modifica la función del ejercicio 3 de forma que acepte
#métricas de distancia con peso de variables.
#...

all_dists = np.array([distance(test_row,train_row, weights) for train_row in X_train])
sorted_dists_indexes = np.argsort(all_dists) # Ordenamos las distancias.
neighbors_dists = all_dists[sorted_dists_indexes[:k]] # Obtenemos las distancias
# de los k más cercanos.
neighbors_classes = y_train[sorted_dists_indexes[:k]] # Devolvemos los índices
# hasta k de y_train.
return neighbors_classes, neighbors_dists

#CLASIFICADOR K-NN
def k_nearest_neighbors(X_train,X_test,y_train,k,selection,distance,weights=None):
  predictions = np.empty(len(X_test),dtype=y_train.dtype)
  for test_ind in range(len(X_test)):
    ###ENCUENTRA LOS K VECINOS MÁS CERCANOS
    test_row = X_test[test_ind]
    neighbors_classes, neighbors_dists = get_neighbors(test_row,X_train,y_train,k,
    distance,weights)
    ###CALCULA LA CLASE SEGÚN LOS K VECINOS MÁS CERCANOS
    predictions[test_ind] = selection(neighbors_classes,neighbors_dists)
  return predictions

```

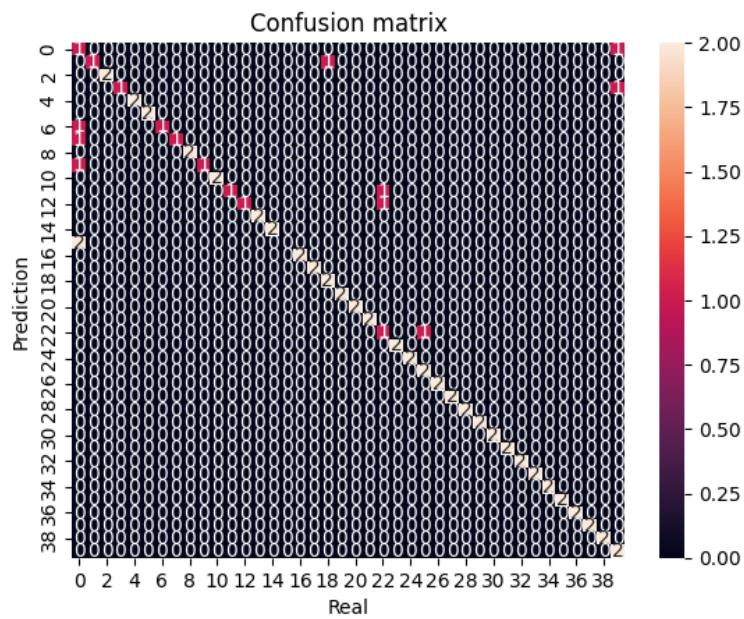
Finalmente, vamos a probar la nueva métrica de distancia con pesado de variables utilizando nuestra base de datos discretizada. Es importante tener en cuenta que el peso de las variables se debe calcular tomando en cuenta SOLO las instancias de entrenamiento, ya que de lo contrario sufriríamos de data leaking. Al igual que en el anterior ejercicio, nos fijaremos solo en el vecino más cercano para clasificar las instancias de test.

```
#CALCULA EL PESADO DE VARIABLES EN BASE A LA INFORMACIÓN MUTUA
weights = np.zeros(len(X_train[0]))
#####AÑADE AQUÍ TU CÓDIGO#####
MI_array = np.array([mutual_information(X_train[:,i], y_train)
for i in range(len(X_train[0]))])
sum_MI = np.sum(MI_array)
weights = MI_array / sum_MI
#PISTA: La posición i del array weights debe contener el peso de la i-ésima
#variable. Los pesos se calculan en base a la información mutua con respecto
#al valor de clase.
#...

pred = k_nearest_neighbors(X_train,X_test,y_train,k=1,selection=majority,
distance=weighted_hamming,weights=weights)

confusion_matrix(pred,y_test,n_classes)
print("Test accuracy:",str(accuracy(pred,y_test)))
```

Output:



Test accuracy: 0.8625

Como se puede observar, obtenemos resultados de accuracy ligeramente superiores a los obtenidos por la **distancia de Hamming** sin pesado de variables.

Por curiosidad, antes de finalizar vamos a representar en forma de imagen en blanco y negro los pesos utilizados para las variables predictoras en el modelo anterior. Dado que cada variable en nuestra base de datos representa un píxel de una imagen, esta visualización nos dará una idea sobre qué partes de las imágenes son más relevantes a la hora de clasificar los rostros del **Olivetti Faces data-set**. Teniendo esto en cuenta, el color blanco representa píxeles con una información mutua más alta con respecto a la variable clase, mientras que lo contrario ocurre con el color negro.

```
plt.imshow(weights.reshape(64, 64), cmap = mpl.cm.gray, interpolation="nearest")
plt.title("Mutual information with respect to Class label")
plt.axis("off")
plt.show()
```

*Output:*

Mutual information with respect to Class label



Curiosamente, las partes más importantes para reconocer a una persona (al menos con nuestro modelo) reside en los alrededores del ojo (frente, nariz, algo de las mejillas) y casi nada en la boca. También se recalca la importancia de los ojos, que, pese a que en esta base de datos las fotos sean en blanco y negro, no deja de ser decisivo para descartar algunas clases. Como apunte final, resulta risible que con un simple antifaz estaríamos dificultándole el trabajo al modelo significativamente.

## Duodécimo laboratorio (13-14 de diciembre)

Último laboratorio. Trata sobre algoritmos genéticos, que se usan para encontrar si bien no el máximo global, una solución muy buena a un problema NP-hard, problemas que tienen la misma dificultad que los problemas NP más difíciles dentro de dicho subconjunto. La base de datos en la que se va a implementar el algoritmo genético se encuentra en el directorio habitual. Éste se llama *ACB-BasketballGamePrediction-2012-2013.arff* [Elgezabal, 2012], pese a que no se diga en la documentación de la base de datos, el objetivo se trata en predecir si dicho equipo ha ganado o no. Las variables predictoras que se han empleado son las siguientes:

- **Código de temporada.** Numérico. Indica el número de la temporada, en este caso 57.
- **Código partido.** Numérico.
- **Fecha.** Date, siguiendo el formato año-mes-día.
- **Numero espectadores.** Numérico.
- **Puntos.** Numérico.
- **t2\_intentos.** Numérico. Números de intentos de **un punto** a canasta.
- **t2\_intentos.** Numérico. Número de intentos de **dos puntos** a canasta.
- **t3\_intentos.** Numérico. Número de intentos de **tres puntos** a canasta.
- **tx\_convertidos.** Numérico. Hay tantas varias variables predictoras como puntos se pueden ganar con un tiro. El número de tiros a canasta que se han encestado, ganando así x puntos.
- **tx\_porcentaje acierto.** Numérico.
- **Rebotes defensivos.** Numérico.
- **Rebotes ofensivos.** Numérico.
- **Asistencias.** Numérico
- **Balones perdidos.** Numérico.
- **Balones robados.** Numérico.
- ...

Para obtener una base de datos como las que se han estado trabajando, se hacen uso de principalmente dos técnicas:

- **Web Scraping.** Con esta manera nosotros mismos nos descargamos una página web y procesamos los datos sobre la misma con varias herramientas a nuestra disposición, ya sean bots, web crawlers y demás. Los datos que se obtienen, más tarde, se añaden en una estructura de nuestra conveniencia, ya sea una lista, una matriz, o en éste caso, un .arff, indicando las relaciones y los atributos. Sin embargo, dado que esto se hace sin permiso del dueño del dominio, se pueden llegar a problemas legales, siendo la más común una denuncia por infracción de derechos de autor. También cabe destacar que los propios dueños de una página web suelen implementar técnicas para evitar web scraping por distintos motivos, uno ellos es, por ejemplo, CAPTCHA.
- **APIs.** Para empezar, una API es un protocolo de comunicación que permite que dos aplicaciones establezcan una canal entre ellas. Éstas son formas legítimas de obtener los datos de la aplicación, página web, etc. que queramos, dado que las ofrece el propio dueño. Por supuesto, la mayoría de las veces el dueño de los datos va a querer una recompensación por esos datos, por lo que la API suele ser de pago, cobra por transacción, o funciona por medio de suscripciones, pero nunca gratis.

Ahora bien, vamos a definir las opciones que vamos a tomar para el algoritmo genético uno por uno, y explicar el razonamiento detrás de él.

- Codificación de los individuos.** De cara a realizar el algoritmo, la codificación de cada individuo es, posiblemente, uno de los pasos más importantes en el algoritmo. Es un gran determinante de las siguientes decisiones: dependiendo en cómo lo codifiquemos, el cruce, mutación, y la función objetiva tendrán un aspecto u otro. En este problema concreto nos importa qué variables usaremos para predecir el resultado de un partido, luego tiene sentido que representemos cada caso como un vector de n casillas, donde n es el número de variables predictoras, el valor que ésta pueda tener es 0 (si lo incluimos) o 1 (si no lo incluimos). Con esta ilustración se entiende mejor:

Población inicial					
1	01101000	01101001	00100001	0110	
2	1001	11000011	10110001	01100001	
3	01101011	01101001	00100000	0110	
4	1001	01101110	01111010	01100001	

Table 1: Una generación “arbitraria” de la población inicial, habrá más que solo cuatro.

- Evaluación de los individuos.** Para evaluarlos, usaremos la precisión que el modelo obtiene. Éste valor es importante, pues se debe tener en cuenta a la hora de cruzar. La tabla se queda así:

	Población inicial	Fitness
1	01101000 01101001 00100001 0110	0.8132
2	1001 11000011 10110001 01100001	0.1534
3	01101011 01101001 00100000 0110	0.1312
4	1001 01101110 01111010 01100001	0.3573

Table 2: Con la columna de fitness añadida.

- Operador de cruce.** Una vez se escogen dos individuos, se decidirá como operador de cruce un punto pivote, a partir del cual la mitad de un individuo y la otra de otro crearemos un hijo y viceversa. Antes de ilustrarlo con un ejemplo primero vamos a establecer el operador de mutación.
- Operador de mutación.** A los individuos que salen del cruce entre otros dos se le asignará una pequeña probabilidad de mutación. Es decir, que estará sujeto a que tenga un cambio aleatoriamente que no tenga por qué mejorar su precisión. En nuestro caso, decidiremos aleatoriamente si vamos a mutar un individuo, y en tal caso negaríamos una posición aleatoria del vector.

	Emparejamientos	Punto de cruce	Nueva población mutada	Fitness
1	011010001101001001000010110 1001011011100111101001100001	16	011010001101001101001100001 1001011011100111001000010110	0.9424 0.8533
	1001110000111011000101100001 011010110110100100100000110	14	1001110000111011000101100001 0110101101101011001101100001	0.4323 0.8953

Table 3: Con el siguiente color se indica que ha habido una mutación.

- Criterio de para del algoritmo genético.** Existen muchos, pero normalmente se suelen establecer dos criterios simultáneos. El primero se puede llamar el “criterio de seguridad”, se establece para que el algoritmo pare en algún momento. El otro tiene más que ver con cómo evoluciona la población, cosa que sería lo ideal. Por ejemplo, cuando el fitness de los individuos no cambia, puesto que ya se habrá alcanzado la mejor solución de todas.

A parte de éstas, existen otros parámetros que se pueden modificar. Es por esto que se le suelen llamar a los algoritmos genéticos una familia, puesto que de esta idea salen distintos heurísticos de búsqueda.

**Borra las siguientes variables: Código temporada, código partido, fecha, hora. El genético tiene problemas con ellas, ¿con cuántas variables predictoras nos quedamos?**

pues si hay 28 variables predictoras, y quitamos cuatro...  $28 - 4 = 24$ .

**¿Cuántos subconjuntos distintos de variables predictoras existen?**

$$2^{24} = 16777216 \text{ diferentes subconjuntos.}$$

Es decir, resolver ésto con fuerza bruta no es una opción. Es por eso que recurrimos a los algoritmos genéticos en primer lugar.

**Explica los parámetros de *WrapperSubsetEval***

El parámetro que indica la técnica que se va a evaluar cada subconjunto es **evaluationMeasure**. Por defecto evaluará la precisión del modelo, tal y como se ha establecido previamente. Ahora bien, a parte de usarlo como “peso” para cruzarlo con otros individuos, también se usará como criterio de parada: si nos fijamos en **threshold**, veremos que el algoritmo no se detendrá salvo cuando la desviación estándar de la media de la precisión de todos los individuos de la población es mayor a dicho umbral = 0.01. Como detalle también se puede observar que se usa una semilla, es decir, que la primera población se escoge aleatoriamente, lo dejamos tal y como está.

**¿Crees que podríamos fijar un clasificador con más coste de CPU, como por ejemplo redes neuronales?**

Sí, pese a que el coste de CPU será mayor, se podría acelerar el proceso de aprendizaje bastante. Este proceso recuerda al “tuneado” de hiperparámetros que hemos realizado en el anterior laboratorio. A parte de lo que vamos a hacer con el clasificador de Naïve-Bayes, es decir, ir comprobando la precisión de cada modelo por individuo, otra forma en la que nos podríamos valer de usar el algoritmo genético en las redes neuronales puede ser para establecer el pesaje de cada neurona.

Al fin y al cabo, lo que una red neuronal busca es minimizar una función objetivo, y hace lo mismo que el genético en este aspecto para ello: actualizar valores, explotar, y explorar (buscar en otra vecindad), solo que en este caso se usa forward propagation, backward propagation... Cosa que podríamos sustituir asignando valores aleatorios a los pesos, tomar el valor de la función como fitness, y darle más importancia a los pesos más eficientes para luego cruzarlos.

**Entiende los primeros cinco parámetros de *GeneticSearch***

1. **Crossover Prob.** La probabilidad que dos individuos de la población tendrán para cruzarse. Ojo. Con esto no se está diciendo que cualquier individuo podrá cruzarse con el otro pese a que tenga un fitness pésimo, y es que antes ya se han escogido los mejores individuos para que se crucen, esta probabilidad simplemente sirve para saber quién con quién se cruza.
2. **Max Generations.** El segundo criterio de paro que es común en todos los algoritmos genéticos. Para que el algoritmo acabe en algún momento, se ha establecido un número máximo de etapas o generaciones, es decir, en el caso de que no se haya encontrado un buen resultado como para terminar el algoritmo, se detendrá después de veinte iteraciones.
3. **Mutation Prob.** La probabilidad de mutación de un descendiente entre dos individuos, poco más que añadir.
4. **Population Size.** El tamaño de la población inicial. En este caso están establecidos veinte.
5. **Report Frequency.** Establecido a veinte, el mismo número de veces que se permite iterar al algoritmo. Su función es dar un reporte de una generación cada n veces. Como se va a establecer ahora a uno, implicará que se verá un informe de cada generación.

```

Selected attributes: 1,6,7,12,13,16,17,18,21,23 : 10
    numero_espectadores
    t3_intentos
    t3_convertidos
    rebotes_defensivos
    rebotes_ofensivos
    balones_robados
    contraataques
    tapones_a_favor
    faltas_recibidas
    valoracion
  
```

Figure 7: Primera selección de variables con los valore por defecto.

```

Learning scheme: weka.classifiers.bayes.NaiveBayes
Scheme options:
Subset evaluation: classification accuracy
Number of folds for accuracy estimation: 5

Selected attributes: 1,12,13,16,17,23 : 6
    numero_espectadores
    rebotes_defensivos
    rebotes_ofensivos
    balones_robados
    contraataques
    valoracion
  
```

Figure 8: Con prob = 0.7

La primera selección de variables se encuentra en la figura 7, con un mérito de 0.8333. El algoritmo se ha detenido con el segundo criterio de parada, ha llegado al máximo número de iteraciones. Pensando que quizás se pueda tratar a que se da poca probabilidad a los individuos con mayor fitness, vamos a subir el valor a “jugar” con el valor.

Primero, parece que cambiando tan solo el valor de probabilidad de cruce no conseguimos una gran mejora, pues obtenemos valores parecidos a 0.83. Un ejemplo de ello se encuentra en la figura 8, en cuyo caso se obtiene un mérito de 0.83987.

También, si se aumenta el número de iteraciones, se obtienen mejores méritos, como lo es el caso con los escogidos en la figura 9, que su valor es de 0.84.722, pero una vez más, no es nada demasiado relevante.

Finalmente, voy a freír mi ordenador y aprovechar que es invierno para calentar mi habitación y ejecutar el programa con 1000 iteraciones como máximo y una población de 100. Se puede ver en la figura 10. El mérito es de 0.84722 una vez más, ¿podría ser que es un máximo global?

**¿Tenemos garantías de que la mejor solución que hemos encontrado en todas nuestras búsquedas-ejecuciones sea el óptimo global? ¿Por qué?**

```

Selected attributes: 1,12,13,16,20,23 : 6
    numero_espectadores
    rebotes_defensivos
    rebotes_ofensivos
    balones_robados
    mates
    valoracion
  
```

Figure 9: Con max iteraciones = 200

```

Selected attributes: 1,12,13,16,20,23 : 6
    numero_espectadores
    rebotes_defensivos
    rebotes_ofensivos
    balones_robados
    mates
    valoracion
  
```

Figure 10: Max Iteraciones = 1000 y Population Size = 100.

```

Ranked attributes:
0.3952 23 valoracion
0.2339 2 puntos
0.1211 12 rebotes_defensivos
0.0959 24 equipo
0.0937 14 asistencias
0.0779 5 t2_porcentaje_acierto
0.0713 8 t3_porcentaje_acierto
0.0555 4 t2_convertidos
0.0548 7 t3_convertidos
0.0369 17 contraataques
0.0264 18 t1_convertidos
0.0218 16 balones_robados
0.0184 15 balones_perdidos
0.0166 6 t3_intentos
0.0153 3 t2_intentos
0.0122 22 faltas_realizadas
0.0111 28 mates
0.0109 9 t1_intentos
0.0103 13 rebotes_ofensivos
0.0098 18 tapones_a_favor
0.0096 21 faltas_recibidas
0.0092 19 tapones_en_conta
0.0089 11 t1_porcentaje_acierto
0.0085 1 numero_espectadores

Selected attributes: 23,2,12,24,14,5,8,4,7,17,10,16,15,6,3,22,28,9,13,18,21,19,11,1 : 24
  
```

Figure 11: Ranking de los atributos.

No tenemos garantías ni podemos comprobarlo, dado que para saberlo tendríamos que compararlo con el resto de comparaciones. Lo que sí sabemos es que es una buena solución, dado que se trata del óptimo local de su vecindad, un resultado más que decente. Además, hemos visto que en nuestro caso hemos obtenido los mismos atributos con una búsqueda más agresiva.

#### ¿Qué otros metaheurísticos de optimización, aparte de los genéticos, nos ofrece WEKA?

Se encuentran los siguientes:

- **Best First.** Escoge la mejor opción y la añade al conjunto. En el caso de selección de atributos, se supone que en vez de escoger una variable aleatoria, escogería la que más aumenta la precisión, así continuamente hasta que no se pueda más.
- **Exhaustive Search.** Fuerza bruta, comprueba todos los subconjuntos posibles y se queda con el máximo. Sobra decir que esto es muy costoso para nuestro caso, lo descartamos.
- **Greedy Stepwise.** Es una forma menos voraz que el propio greedy. Es un algoritmo bastante extenso que hace uso de un valor p para decidir si una variable entra (o sale) en el siguiente enlace se explica con más detenimiento. Se encuentra dentro de la documentación del paquete Caret sobre el que ya se ha trabajado previamente.
- **Random Search.** Si bien hay varias formas de realizar una búsqueda aleatoria, la forma que WEKA ha escogido es la siguiente: Se empieza por un punto aleatorio o dado por el usuario. A partir de dicho subconjunto, Random Search se encarga de generar otros subconjuntos de variables y ofrecer uno que ofrezca un resultado mejor o igual.
- **Ranker.** ofrece un ranking de los atributos. Ésta clasificación la podemos escoger nosotros mismos, usando la información que aporta cada variable hemos obtenido el ranking que sale en la figura 11. Como se puede ver, este no es un algoritmo de búsqueda, para eso está Rank Search.
- **Rank Search.** Con el ranking obtenido en la figura 11, se empieza con un subconjunto con tan solo la primera variable, y se van añadiendo las siguientes variables hasta que no se reporte una mejora.

Ofrece unas líneas de código, en R o en Python, para nuestro propósito: "feature selection in wrapper form using a genetic algorithm".

```

# Cargamos la librería Caret y sus dependencias.
library(ggplot2)
library(lattice)
# Librería para que el proceso del genético no se eternice.
library(doParallel)

## Loading required package: foreach
## Loading required package: iterators
## Loading required package: parallel
cl <- makePSOCKcluster(detectCores() - 1)
registerDoParallel(cl)
# Cargamos la librería Caret.
library(caret)
library(randomForest)

## randomForest 4.7-1.1
## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin
set.seed("1241273") # Establecemos una semilla para las pruebas.

```

Cargamos el conjunto de datos.

```

# Directo a la variable basket.
dir <- "/home/mikel/Documents/Latex_Proyectos/MDD Laboratorio Segunda Parte/datasets/"
dir <- paste(dir, "ACB-BasketballGamePrediction-2012-2013.csv", sep = "")
basket <- read.csv(dir)

```

En la documentación de Caret se explica muy bien en lo que un algoritmo genético se basa: imitan la evolución Darwiniana en pos de encontrar los mejores individuos, simulamos un mundo en el que nosotros decidimos las reglas. Primero de todo, vamos a quitar las mismas variables que en WEKA.

```

# Nos quedamos con las columnas a partir de la cuarta.
basket_selection <- basket[, 5:length(basket)]

```

Según la documentación de Caret, la función que se encarga del genético es la siguiente:

```
help(gafs)
```

Según la página de ayuda, tiene varios parámetros que deberemos entender antes de implementar nada. En el parámetro x se coloca la colección de datos sin la clase, y en el parámetro y un vector con la clase correspondiente, vamos a prepararlo.

```

# Esta será nuestra x.
basket_samples <- basket_selection[, -ncol(basket_selection)]
# Esta será nuestra y, vector de outcomes.
basket_outcomes <- basket_selection$clase_resultado
basket_outcomes <- as.factor(basket_outcomes)

```

Vamos a enlistar y explicar los parámetros que tiene en común con WEKA.

- **iters.** El número de iteraciones máximo, como hemos dicho antes en WEKA, éste es un criterio de parada secundario, no es lo ideal acabar el algoritmo de ésta forma en la selección de variables. Por defecto son diez.
- **popSize.** El número de la población inicial. Por defecto se empiezan con cincuenta.
- **pcrossover.** La probabilidad de cruce entre dos individuos. Por defecto es del 80 %.
- **pmutation.** La probabilidad de mutación de un nuevo individuo. por defecto es del 10 %.

Es decir, los parámetros que son claves para empezar un algoritmo genético siempre se mantienen. Ahora vamos a echar un vistazo a gafsControl.

```
help(gafsControl)
```

Nada nuevo con respecto a los demás Control que hemos visto con otros modelos, así como el K-NN. Sirven para especificar mejor las decisiones que va a tomar el algoritmo.

```
# Paralelizamos para que el proceso no se eternice.  
ctrl <- gafsControl(functions = caretGA, method = "cv", number = 5, allowParallel = TRUE,  
genParallel = TRUE)
```

El apartado functions se puede personalizar al máximo: se puede establecer la función de cruce, la codificación de cada individuo... Todo. Para hacer esta parte breve, vamos a usar rfGA, un objeto que ya está a nuestra predisposición y que usa Random Forest. Con el método “repeatedcv” indicamos que vamos a usar cross-validation para evaluar cada modelo y con repeats = 5 indicamos que k = 5. Ahora que ya sabemos cómo funciona y lo que debemos pasar, vamos a realizar una llamada y examinar el resultado, ¿será el mismo que el obtenido con WEKA?

```
rf_ga <- gafs(x = basket_samples, y = basket_outcomes, iters = 2, popSize = 20, method = "naive_bayes",  
metric = "Accuracy", gafsControl = ctrl)  
rf_ga
```

Genetic Algorithm Feature Selection

```
612 samples  
24 predictors  
2 classes: 'derrota', 'victoria'
```

```
Maximum generations: 2  
Population per generation: 20  
Crossover probability: 0.8  
Mutation probability: 0.1  
Elitism: 0
```

```
Internal performance values: Accuracy, Kappa  
Subset selection driven to maximize internal Accuracy
```

```
External performance values: Accuracy, Kappa  
Best iteration chose by maximizing external Accuracy  
External resampling method: Cross-Validated (5 fold)
```

During resampling:

- \* the top 5 selected variables (out of a possible 24):
  - rebotes\_defensivos (100%), valoracion (100%), balones\_perdidos (60%), numero\_espectadores (60%), balones\_robados (40%)
- \* on average, 8.4 variables were selected (min = 2, max = 15)

In the final search using the entire training set:

- \* 10 features selected at iteration 1 including:  
t2\_convertidos, t3\_convertidos, t1\_intentos, rebotes\_defensivos, rebotes\_ofensivos ...
- \* external performance at this iteration is

Accuracy	Kappa
0.8121	0.6241

Antes de analizar los resultados, cabe destacar que se han establecido dos iteraciones como máximo puesto que el proceso es bastante costoso incluso si se está paralelizando.

Como se puede ver, las variables que se han escogido son rebotes defensivos, valoración, balones perdidos, numero espectadores y balones robados. No coincide del todo con lo que WEKA nos ofrece.

### Conclusión

En los enlaces ofrecidos en el laboratorio, lo que más me ha llamado la atención son la de maneras que hay para codificar un individuo, y con qué premiarlo. En bipedal walkers me parece un acierto dar un bonus a los individuos que den un paso, y es que si éste no existiera las generaciones resultantes solo intentaría mantenerse de pie el máximo tiempo posible. Aun así, tal y como hemos visto en estas últimas pruebas, con tantas generaciones como 100 que he dejado la página, los sujetos siguen desplomándose tras unos pasos. Con esto llego a la conclusión de que para que un algoritmo de este estilo lleva más y más tiempo cuanto más complejo sea el problema y por supuesto, más variables tenga (pero esto ocurre con todo).

## References

- [Alkassar et al., 2005] Alkassar, A., Nicolay, T., and Rohe, M. (2005). Obtaining true-random binary numbers from a weak radioactive source. *Lecture Notes in Computer Science*, 3481:634–646.
- [Barrutia, 2022] Barrutia, J. (2022). Precipitation in Donostia 2022. <http://www.sc.ehu.es/ccwbayes/docencia/md/selected-dbs/supervised-classification/precipitacion-in-Donostia-2022.arff>.
- [Elgezabal, 2012] Elgezabal, J. (2012). ACB-BasketballGamePrediction-2012-2013. <http://www.sc.ehu.es/ccwbayes/docencia/md/selected-dbs/supervised-classification/ACB-BasketballGamePrediction-2012-2013.arff>.
- [Hartigan, 1975] Hartigan, J. (1975). *Clustering Algorithms*. John Wiley and Sons, New York.
- [Kuhn and Max, 2008] Kuhn and Max (2008). Building Predictive Models in R Using the Caret Package. *Journal of Statistical Software*, 28(5):1–26.
- [Naftali, 2014] Naftali, H. (2014). Visualizing k-means clustering. <https://www.naftaliharris.com/blog/visualizing-k-means-clustering/>.