Introducción al aprendizaje no supervisado

Juan C. Laria

2020-03-04

Contents

Chapter 1

Clústers partitivos

Utilizaremos los datos de Weber and Weber (1974) sobre consumo de proteínas en los países europeos en la década de los '70 del pasado siglo.

Comenzamos por cargar los datos en **R**.

```
data.protein = read.csv("data/protein.csv")
```

En este caso, el archivo que queremos importar en **R** se encuentra en un recurso web, en lugar de la ruta local usual. Una vez importados los datos, podemos ver de qué se tratan.

```
library(dplyr)
library(magrittr)

data.protein %>% head
```

```
dim(data.protein)
```

Tenemos 25 países y 10 variables, aunque una de estas variables es `Country`, que contiene el identificador de cada fila. Debemos transformar este dataset para deshacernos de la columna `Country` sin perder la información de las filas. Esto lo hacemos con el comando `rownames` (que es de lectura y asignación).

```
rownames(data.protein)
```

```
rownames(data.protein) = data.protein$Country
```

Si ahora vemos nuestro dataset, observamos que la información de la variable `Country` está repetida en los nombres de las filas.

```
data.protein %>% head
```

Por tanto, podemos eliminar esa columna

```
data.protein$Country = NULL
```

Como podemos observar, aunque posiblemente las variables estén medidas en las mismas unidades, no se encuentran en el mismo rango de valores. Por ejemplo, `Cereals` toma valores altos comparada con el resto. Para evitar que la escala individual de las variables afecten el resultado final, escalamos previamente la matriz de datos.

```
data.protein %<>% scale %>% as.data.frame
data.protein %>% head
```

1.1 K medias

K-means es, probablemente, uno de los algoritmos de clúster más eficientes computacionalmente. Sin embargo, tiene dos desventajas fundamentales.

1. Se considera únicamente la distancia euclídea.
2. Hay que especificar desde el comienzo el número de clusters que queremos.

El funcionamiento de este algoritmo se ilustra aquí.

Para hacer k-medias en R utilizamos la función `kmeans` del paquete base `stats`. En este caso, especificaremos 5 clusters.

```
km = kmeans(data.protein, centers = 5)
km$cluster
```

Como vemos, la función `kmeans` devuelve una agrupación, de hecho, `km$cluster` tiene una estructura que comparten todos los algoritmos de agrupación (observación, clúster).

1.2 Librería `factoextra`

`factoextra` es un paquete para extraer y visualizar las salidas de distintos análisis exploratorios en R. Para visualizar los grupos resultantes, podemos utilizar la función `fviz_cluster` de la librería `factoextra`. Esta función visualizará los datos utilizando las dos primeras componentes principales.

```
library(factoextra)
fviz_cluster(km, data=data.protein, labelsize = 8)

 # Interactivo
library(plotly)
fviz_cluster(km, data=data.protein, labelsize = 8)%>%
ggplotly()
```

1.3 Variación de información

¿Cómo podríamos medir cuán diferentes son dos agrupaciones dadas sobre los mismos datos? Por ejemplo, la salida de `km$cluster` escalando y sin escalar los datos.

La *variación de información* proporciona una medida para decir cuánta información comparten (o en este caso, no comparten) dos particiones de los datos. De hecho, es una distancia en el sentido matemático estricto de distancia.

```
cluster_scale <- km$cluster

# Repetimos el mismo análisis pero sin escalar
data.protein = read.csv("data/protein.csv")
rownames(data.protein) = data.protein$Country
data.protein$Country = NULL
km = kmeans(data.protein, centers = 5)

cluster_no_scale <- km$cluster
```

El paquete `mcclust` incluye una función para calcular la variación de información.

```
library(mcclust)
vi.dist(cluster_scale, cluster_no_scale, parts = TRUE)
```

¿ Con una agrupación aleatoria qué obtendríamos ?

```
cluster_rand <- sample(5, length(cluster_scale), replace = TRUE)

vi.dist(cluster_rand, cluster_scale)
```

1.4 Minibatch kmeans

MiniBatch k-means ha sido propuesto como una alternativa al algoritmo *k-means* para agrupar datos masivos. La ventaja de *MiniBatch k-means* es que reduce el coste computacional al no utilizar todos los datos en cada iteración, sino una muestra aleatoria de tamaño fijo.

En esta sección ilustraremos el uso de *MiniBatch k-means* en segmentación de imágenes (*Color Quantization*). Utilizaremos las librerías de R `ClusterR` para el algoritmo, y `OpenImageR` para representar las imágenes.

```
library(ClusterR)
library(OpenImageR)
```

En este ejemplo utilizaremos una imagen de arte rupestre. Primero descargamos y leemos la imagen.

```
img = readImage("data/beach.png")
dim(img)
```

```
## [1] 299 299   3
```

Podemos observar que la imagen tiene una resolución de 299, 299, 3 píxeles, separados en tres canales de colores (RGB), por lo que en realidad tenemos 8.9401×10^4 datos y 3 variables. Para mostrar la imagen en pantalla utilizamos la función `imageShow`.

```
imageShow(img)
# img <- RGB_to_HSV(img)
```

A continuación, convertimos nuestra matriz `381x514x3` en otra matriz de dimensión `195834x3`.

```
img.vector = apply(img, 3, as.vector)
dim(img.vector)
```

Ya estamos en condiciones de hacer MiniBatch K-means a nuestra imagen.

```
mbkm = MiniBatchKmeans(img.vector, clusters = 3)
```

A continuación, vamos a sustituir la información de cada píxel por el centro del clúster al que pertenece. De esta forma obtendremos una imagen que tiene solamente 10 colores.

```
mb.pred = predict_MBatchKMeans(img.vector, mbkm$centroids)
new.img = mbkm$centroids[mb.pred, ]
```

Devolvemos la imagen a su estructura original para poderla representar.

```
dim(new.img) = c(nrow(img), ncol(img), 3)
imageShow(new.img)
```

1.5 K mediano

Partition around medoids puede verse como una versión robusta de *kmeans*, ya que los centroides son observaciones en lugar de promedios de observaciones. Esto da lugar a clusters más interpretables. Adicionalmente, su implementación tiene ventajas con respecto a *kmeans*, por ejemplo, no se limita a la distancia euclídea.

Para este algoritmo, utilizaremos la función `pam` de la librería `cluster`.

```
library(cluster)
pm = pam(data.protein, k=5)
knitr::kable(pm$medoids)
```

El valor que devuelve la función `pam` es similar al que devuelve `kmeans`, excepto que en lugar de tener la propiedad `centers`, ahora tenemos `medoids`.

Podemos visualizar la agrupación resultante.

```
fviz_cluster(pm, data=data.protein, labelsize = 8)
```

1.5.1 `CusterR::clara`

Como es lógico, *pam* es computacionalmente más costoso que *k-means*, pues calcular el *medoid* es mucho más difícil que hacer un promedio. Existe una alternativa eficiente a *k-medoids*, *clara* para agrupar grandes volúmenes de datos. Se basa en agrupar primero una muestra de los datos originales y luego asignar los datos restantes a estos grupos.

Utilizaremos la función `Clara_Medoids` del paquete `ClusterR`, porque vamos a trabajar con la misma foto que usamos para ilustrar `MiniBatchKmeans`.

```
clara.m = Clara_Medoids(img.vector, clusters = 10, samples = 10,
                        sample_size = 0.001)
```

Hay que ser muy cuidadosos aquí con los parámetros, `sample` y `sample_size`, porque tenemos muchos datos, y esta función es muy costosa computacionalmente si aumentamos estos parámetros.

A continuación asignamos cada pixel a su color correspondiente, y mostramos la nueva imagen.

```
clara.pred = predict_Medoids(img.vector, clara.m$medoids)
new.img = clara.m$medoids[clara.pred$clusters, ]
dim(new.img) = c(nrow(img), ncol(img), 3)
imageShow(new.img)
```

1.6 Selección de K

En esta sección estudiaremos tres técnicas para calcular el número óptimo de clusters. Primero, vamos a sacar una muestra de nuestros datos de menos tamaño, ya que los cálculos que haremos serán más costosos.

```
set.seed(1)
img.vector.small = img.vector[sample(nrow(img.vector),1000),]
```

1.6.1 Método del codo

La idea básica de los algoritmos partitivos es obtener cluster con la mínima WSS (within-cluster sum of squares), que mide cuán compactos son los clusters. Pudiéramos intentar quedarnos con el número de clusters `nclust` que minimiza

este valor. Sin embargo, WSS siempre decrece a medida que consideramos un mayor número de grupos.

El método del codo mira el valor de WSS con respecto al número de grupos considerados, y busca el primer punto en que hay un cambio brusco en la curva, es decir, que adicionar un grupo nuevo no mejora demasiado con respecto a lo que ya había.

```
library(factoextra)

fviz_nbclust(img.vector.small, kmeans, method = "wss") +
    geom_vline(xintercept = 3, linetype = 2)+
  labs(subtitle = "Elbow method")
```

1.6.2 Método average silhouette

La idea básica de este método es medir la calidad de la agrupación en función de cuán bien encierra los datos en los diferentes grupos. ¿Debería cambiar mucho la silueta del cluster si quitamos alguna de sus observaciones?

El método de la silueta promedio calcula la silueta de los grupos para distintos números de grupos `nclust`. El mejor número es aquel que maximiza la silueta.

```
fviz_nbclust(img.vector.small, kmeans, method = "silhouette")+
  labs(subtitle = "Silhouette method")
```

1.6.3 Método Gap statistic

Este se puede considerar el más formal de los métodos, y puede ser aplicado a cualquier método de clustering, incluyendo clúster jerárquico.

```
set.seed(3)
gap_stat = cluster::clusGap(img.vector.small, FUN = kmeans,
                            K.max = 20)
fviz_gap_stat(gap_stat,
              maxSE = list(method="Tibs2001SEmax"))
+labs(subtitle = "Gap statistic method")
```

Chapter 2

Clústers jerárquicos

2.1 hclust

Para crear un *cluster jerarquico* aglomerativo utilizaremos la función `hclust` del paquete básico `stats`. La sintaxis de esta función es `hclust(d, method)`, donde `d` es una *matriz de distancias* entre las observaciones y `method` (función *linkage*) describe el criterio que usaremos para unir distintos clusters.

Para calcular la matriz de distancias `d` podemos usar la función `dist`, en este caso, `dist(data.protein, method = "euclidean")`, donde el parámetro `method` describe qué distancia estamos calculando. Esta no es la única función que podemos utilizar para calcular distancias. Por ejemplo, si nuestros datos son geográficos `(longitud, latitud)`, usaríamos la función `distm` del paquete `geosphere`.

```
hc = hclust(dist(data.protein, method = "euclidean"),
            method = "complete");
hc
```

Podemos utilizar el método `plot` de nuestro cluster `hc`, con el parámetro `hang=-1`, que dibuja todas las etiquetas al mismo nivel.

```
plot(hc, hang = -1, cex=0.8)
```

Lo que **R** ha dibujado ha sido el *dendograma* de `hc`. Este gráfico se utiliza para describir la asignación de los clusters para cada valor de `Height`. Cuanto más cerca del cero se juntan las observaciones, en este caso los países, más similares son en cuanto a consumo de proteínas. Para cada valor específico de `Height` tenemos una asignación diferente de los clusters.

2.1.1 Distancias. ¿Qué hace la función `dist`?

Vamos a explorar el resultado de la función `dist` aplicada a nuestros datos.

```
d = dist(data.protein, method = "euclidean"); head(d)
```

Como vemos, `d` es una matriz simétrica, con ceros en la diagonal, que **R** almacena en forma de *vector*, y para ahorrar espacio solamente almacena el triángulo inferior de la matriz. Si hacemos `d[1]` obtenemos la distancia (euclídea) entre Austria y Albania.

```
d[1]
sum((data.protein["Austria",] - data.protein["Albania",])^2)^(1/2)
```

Como vemos, `d[1]` coincide con la distancia euclídea entre las observaciones (vectores) `data.protein["Austria",]` y `data.protein["Albania",]`, calculada usando la fórmula. Existen además otras distancias que podemos calcular usando la función `dist`.

```
d = dist(data.protein, method = "maximum"); d[1]
max(abs(data.protein["Austria",] - data.protein["Albania",]))
```

```
dist(data.protein[c("Albania", "Austria"),], method = "manhattan")
sum(abs(data.protein["Austria",] - data.protein["Albania",]))
```

2.1.2 Linkages. ¿Qué especifica el parámetro `method` en la función `hclust`?

Comparemos los dos *dendogramas* que se obtienen al variar el parámetro `method` de la función `hclust`.

```
    par=par(mfrow=c(1,2), cex=0.5)
    hc1 = hclust(dist(data.protein, method = "euclidean"),
                 method = "complete")
    hc2 = hclust(dist(data.protein, method = "euclidean"),
                 method = "single")
    plot(hc1, hang=-1)
    plot(hc2, hang=-1)
    par(par)
```

Como observamos, hemos obtenido *dendogramas* muy diferentes al cambiar `method="complete"` por `method = "single"`.

En general, una función *linkage* especifica una **similitud** (no necesariamente una distancia en el sentido matemático) entre dos conjuntos (clusters) de datos.

2.1.3 Los métodos `cutree` y `rect.hclust`

Independientemente de la distancia (o *dis-similitud*) que consideremos entre las observaciones, y el método *linkage* para agrupar, usualmente el objetivo que perseguimos al hacer clúster jerárquico es reportar posibles grupos latentes en las observaciones. Sin embargo, hasta ahora hemos visto cómo obtener un *dendogram*, pero no cómo decidir qué grupos considerar.

La función `cutree` del paquete básico `stats` realiza un corte horizontal del *dendograma*. Podemos proporcionar uno de los dos, un número fijo de grupos mediante el parámetro `k`, o una altura en la cual cortar con el parámetro `h`.

```
cutree(hc1, k=3)
```

La función `cutree` retorna un vector de tipo `Named int` de longitud es número de observaciones, con los índices de pertenencia a los grupos. Podemos utilizar esta información para trabajar con los diferentes grupos.

```
cut = cutree(hc1, k=5)
for (i in 1:5) {
    write(paste0("Cluster ",i,":\n",
                 toString(names(which(cut==i))),
                "\n-----------------\n"), "")}
```

Otra función que puede ser muy útil para representar la agrupación obtenida es `rect.hclust`, también del paquete base `stats`.

```
plot(hc1, hang = -1, cex=0.7)
rect.hclust(hc1, k=5)
```

2.2 Librería `protoclust`

Recientemente, Bien and Tibshirani (2011) han introducido un nuevo tipo de *linkage*, el **minimax linkage**. Este tiene la propiedad de que para un corte a altura `h`, cualquier punto está a distancia menor que `h` del centro de su cluster.

Para utilizar el *linkage minimax*, debemos instalar el paquete `protoclust`

```
library(protoclust)
```

La función `protoclust` retorna un objeto similar a la salida de `hclust`.

```
pc = protoclust(dist(data.protein, method = "euclidean"))
plot(pc, hang = -1, cex=0.7)
rect.hclust(pc, k=3)
```

Podemos visualizar los dendrogramas con `factoextra`.

```
library(factoextra)
res <- hcut(data.protein, hc_func = "hclust",
```

```
            hc_method = "complete", k = 5)
fviz_dend(res)
fviz_dend(res, type = "circular")
```

Chapter 3

Self Organizing Maps

En esta práctica veremos un ejemplo de aplicación de los SOM. Comenzamos leyendo los datos

```
data.protein = read.csv("data/protein.csv")
rownames(data.protein) = data.protein$Country
data.protein$Country = NULL
data.protein = scale(data.protein)
```

3.1 Librería `kohonen`

Ahora crearemos el grid SOM. Generalmente especificamos el tamaño del grid antes de entrenar el modelo.

```
library(kohonen)
som_grid = somgrid(xdim = 4, ydim= 4, topo = "hexagonal")
```

Aquí especificamos la topología de la capa de salida. En este caso, será un grid de 4x4, donde cada neurona tiene 6 vecinos. Si elegimos `topo="rectangular"` entonces cada neurona tendrá 4 vecinos.

A continuación, entrenamos la red.

```
som_model = som(data.protein,
                grid = som_grid,
                rlen = 100,
                keep.data = T)
```

El parámetro `rlen`especifica el número de épocas (barridos sobre todo el dataset).

Podemos visualizar el proceso de entrenamiento.

```
plot(som_model, type = "changes")
```

Esto nos permite decidir si hay algún parámetro que debemos variar, como `rlen`, `alpha`, `radius`, etc.

```
som_model = som(data.protein,
                grid = som_grid,
                rlen = 600,
                keep.data = T)
plot(som_model, type = "changes")
```

También podemos ver de forma gráfica el número de veces que cada neurona es activada en el modelo final.

```
plot(som_model, type="count")
```

3.1.1 U-Matrix

La U-Matrix (Unified distance matrix o Neighboring distances matrix) es una representación de un SOM, donde las distancias euclídeas entre los pesos de neuronas vecinas son descritas en una imagen en escala de grises.

Veamos cómo construir la U-Matrix

```
plot(som_model, type="dist.neighbours", palette.name = gray.colors)
```

En un SOM los outliers se manifiestan como nodos cuyos pesos están muy alejados de los pesos de sus nodos vecinos. Por tanto, en la U-matrix los nodos outlies son los más claros. Todas las observaciones que activen nodos claros son observaciones que tienen un patrón muy diferente al resto.

Si queremos recuperar las filas que activan nodos más claros, hacemos lo siguiente.

```
Umat = plot(som_model, type="dist.neighbours")

data = merge(data.frame(obs = rownames(data.protein),
                        nodo = som_model$unit.classif),
             data.frame(nodo = 1:length(Umat),
                        u = Umat))

summary(data$u)

library(dplyr)
data %>% arrange(desc(u))
```

Podemos visualizar también las características (pesos) de cada nodo de salida.

```
plot(som_model, "codes")
```

3.1.2 Clusters

Una vez que tenemos entrenada la red, cada neurona de la capa de salida tendrá asociados unos pesos, a los que podemos acceder con

```
som_model$codes
```

Estos pesos pueden ser interpretados como puntos en el espacio de los datos originales, por lo que podemos aplicarle a estos las técnicas de segmentación que ya conocemos (e.g. kmeans o hclust).

En primer lugar, calculamos las distancias entre los nodos del mapa (en el espacio de los datos originales)

```
dist = object.distances(som_model, "codes")
```

A continuación, agrupamos estos nodos usando cluster jerárquico.

```
hc = hclust(dist, method = "ward.D2")
```

Visualizamos para encontrar el número de clusters.

```
library(factoextra)
fviz_dend(hc, show_labels = F)
som.hc = cutree(hc, k = 4)
```

```
plot(som_model, type="dist.neighbours", palette.name = gray.colors)
add.cluster.boundaries(som_model, som.hc)
```

```
colors = topo.colors(3)
plot(som_model, type="mapping", bgcol = colors[som.hc],
     pch = "", main = "Clusters")
add.cluster.boundaries(som_model, som.hc)
```

Chapter 4

Clústers basados en densidades

4.1 Librería `dbscan`

DBSCAN (Density-Based Spatial Clustering and Application with Noise) es un algoritmo de cluster basado en densidades, que puede ser utilizado para identificar clusters que contengan patrones, ruido y outliers. Mediante la grupación basada en densidades, se localizan áreas de mayor densidad con respecto al resto de los datos. Los puntos que quedan fuera de estas áreas de densidad son considerados ruido o puntos de frontera.

Si quisiéramos enseñar a una máquina a "observar" patrones como los seres humanos lo hacemos, probablemente deberíamos comenzar plantéandonos un método como DBSCAN.

El algoritmo DBSCAN se basa en la noción intuitiva detrás de lo que es cluster y lo que es ruido. Para los puntos que pertenecen a un cluster, hay una "vecindad" de radio dado, tal que esta vecindad contiene un número mínimo de puntos que también pertenecen al mismo cluster.

Por ejemplo, la siguiente imagen puede ser fácilmente segnemtada por un ser humano con una simple pasada. ¿Pudiésemos utilizar análisis cluster para determinar quiénes son los personajes de esta historia?

```
library(dplyr)
library(ggplot2)

points = read.table(file = "data/arte-rupestre.txt",
                    header = T)
ggplot(points, aes(x=x, y=y)) +
```

```
  geom_point(size = 0.5) +
  theme_void()
```

Veamos qué sucede si intentamos particionar estos puntos usando k-means.

```
library(factoextra)

set.seed(0)
km.res = kmeans(points, 3, nstart = 9)
fviz_cluster(km.res, points, geom = "point",
             ellipse = F, show.clust.cent = F,
             palette="jco", ggtheme = theme_classic() )
```

Como podemos observar, k-means no identifica apropiadamente estos clusters con formas arbitrarias, incluso cuando le decimos exactamente cuántos grupos verdaderos hay.

4.1.1 ¿Cómo funciona?

El objetivo de DBSCAN es identificar regiones densas, que pueden ser medidas según el número de objetos cercanos a un punto dado.

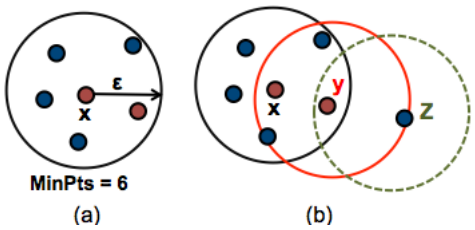
DBSCAN requiere dos hiperparámetros importantes,

- **epsilon** (`eps`) Define el radio de la vecindad alrededor de un punto x, comúnmente llamada $\epsilon-$vecindad.
- **minimum points** (`minPts`) Número mínimo de vecinos en un radio `eps`.

Esto da lugar a una clasificación en tres tipos de puntos. Dado x que pertenece al conjunto de datos,

- x es un punto **interior** si tiene más de `minPts` puntos en su $\epsilon-$vecindad.
- x es un punto **frontera** si no es interior, pero pertenece a la $\epsilon-$vecindad de algún punto interior.
- x es un **outlier** si no es punto interior ni frontera.

La siguiente figura muestra los diferentes tipos de puntos que podemos tener. En este ejemplo, x es un punto interior (porque tiene 6 puntos a distancia menor que `eps`), y es un punto frontera (porque pertenece a la $\epsilon-$vecindad de x) y z es un outlier.



El algoritmo DBSCAN se basa en las siguientes nociones de conectividad entre puntos.

- **Direct density reachable** ($\rightarrow$) Decimos que $B \rightarrow A$ si
  i. A pertenece a la $\epsilon-$vecindad de B
  ii. B es un punto **interior**
- **Density reachable** ($\rightarrow \cdot \rightarrow$) Decimos que $B \rightarrow \cdot \rightarrow A$ si existe un conjunto de puntos interiores que lleva de B a A.
- **Density connected** A y B son density connected si existe un punto interior C tal que $B \leftarrow \cdot \leftarrow C \rightarrow \cdot \rightarrow A$.

Un **cluster** basado en densidad se define como un conjunto de puntos density connected. La idea del algoritmo DBSCAN es la siguiente.

1. Para un punto inicial x_0 encontrar todos los puntos en su $\epsilon-$vecindad. Cada punto con un número de vecinos mayor o igual que `minPts` es marcado como **interior**, de lo contrario es marcado como **visitado**.
2. Para cada punto interior, si este no está en ningún cluster, crear un nuevo cluster. Recursivamente, encontrar todos los puntos density connected con él, y asignarlos a su mismo clúster.
3. Iterar 1-2 en los puntos restantes **no visitados**.
4. Los puntos que no pertenecen a ningún cluster son marcados como outliers.

Visualización paso a paso: https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/

4.1.2 Ventajas

1. A diferencia de k-means, DBSCAN no requiere un número prefijado de clusters.
2. DBSCAN puede lidiar con cualquier forma en los clústers, no necesariamente circular.
3. DBSCAN identifica los outliers.

4.1.3 Utilización

La función `dbscan`, del paquete del mismo nombre, es una implementación optimizada del algoritmo DBSCAN.

Para comprobar si el paquete no está instalado e instalarlo, ejecutamos la línea siguiente.

```
if(!require("dbscan"))install.packages("dbscan")
```

Vamos a aplicar el algoritmo DBSCAN al conjunto de puntos de nuestro ejemplo.

```
library(dbscan)
db = dbscan(points, eps = 15, minPts = 6)
fviz_cluster(db, data=points, stand = FALSE,
```

```
             ellipse = FALSE, show.clust.cent = FALSE,
             geom = "point",palette = "jco", ggtheme = theme_classic())
```

Ups! DBSCAN solamente identifica un cluster en este ejemplo. Veamos si podemos encontrar los hiperparámetros `eps` y `minPts` óptimos.

4.1.4 Determinando el `eps` óptimo.

El método para determinar el mejor `eps`consiste en, fijado el valor de `minPts`, calcular la distancia media de cada punto a sus k =`minPts` vecinos más cercanos. Luego este promedio para cada punto es mostrado en orden ascendente, y el valor de `eps` se escoje mirando el primer codo.

En nuestro ejemplo,

```
dist <- kNNdist(points, k=10)
dist <- sort(dist)

ggplot() +
  aes(x = 1:length(dist), y = dist) +
  geom_line()+
  geom_hline(yintercept = 6, linetype = "dashed")
```

```
library(dbscan)
db = dbscan(points, eps = 6, minPts = 10)
fviz_cluster(db, data=points, stand = FALSE,
             ellipse = FALSE, show.clust.cent = FALSE,
             geom = "point",palette = "jco", ggtheme = theme_classic())
```

4.1.5 Más ejemplos

```
data("multishapes", package = "factoextra")
df <- multishapes[, 1:2]
ggplot(df, aes(x=x, y=y)) +
  geom_point(size = 0.5) +
  theme_void()


set.seed(123)
db <- dbscan(df, eps = 0.15, minPts = 5)
# Plot DBSCAN results
library("factoextra")
fviz_cluster(db, data = df, stand = FALSE,
             ellipse = FALSE, show.clust.cent = FALSE,
             geom = "point",palette = "jco", ggtheme = theme_classic())
```

Chapter 5

Aprendizaje semi-supervisado

En esta sección estudiaremos un tipo de problema intermedio entre supervisado y no supervisado, que es muy común en la práctica.

5.1 Librería `s2net`

La librería `s2net` de R resuelve un problema de optimización similar al elastic-net, pero para tratar con datos semi-supervisados. No obstante, es más general, e incluye el caso de datos supervisados también. Fue concebido específicamente para resolver problemas de transfer-learning en datos de alta dimensión, donde además de ajustar un modelo lineal, se desea incluir solamente un subconjunto de variables explicativas en el modelo (como hace elastic-net).

Más información aquí: https://github.com/jlaria/s2net

You can install the released version of `s2net` from CRAN with:

```
install.packages("s2net")
```

The development version can be installed with:

```
devtools::install_github("jlaria/s2net", build_vignettes = TRUE)
```

This is a basic example which shows you how to use the package. Detailed examples can be found in the documentation and vignettes.

```
library(s2net)
# Auto-MPG dataset is included for benchmark
data("auto_mpg")
```

Semi-supervised data is made of a labeled dataset `xL`, the labels `yL`, and unlabeled data `xU`. Package `s2net` includes the function `s2Data` to process semi-supervised datasets.

```
head(auto_mpg$P2$xL, 2) # labeled data
head(auto_mpg$P2$yL, 2) # labels
head(auto_mpg$P2$xU, 2) # unlabeled data

train = s2Data(auto_mpg$P2$xL, auto_mpg$P2$yL, auto_mpg$P2$xU, preprocess = TRUE)

head(train$xL, 2)
```

The data is centered and scaled, and factor variables are automatically converted to numerical dummies. Constant columns are also removed. If we wanted to use validation/test data, we must pre-process it according to the training data, with:

```
valid = s2Data(auto_mpg$P2$xU, auto_mpg$P2$yU, preprocess = train)
```

There are two ways to fit a semi-supervised elastic-net using `s2net`. The easiest way is using the function `s2netR`, that returns an object of S3 class `s2netR`.

```
model = s2netR(train,
               params = s2Params(lambda1 = 0.01,
                                 lambda2 = 0.01,
                                 gamma1 = 0.01,
                                 gamma2 = 100,
                                 gamma3 = 0.1))

class(model)
model$beta

ypred = predict(model, valid$xL)
```

Chapter 6

Selección de hiperparámetros

En esta sección estudiaremos un enfoque moderno a la selección de hiperparámetros en algoritmos supervisados. Nos enfocaremos en el elastic-net, pero los resultados se pueden extender a cualquier algoritmo supervisado soportado por la librería `parsnip`.

Comenzamos cargando y preparando los datos. Usaremos unos datos simulados.

```
load("data/sim_data.RData")
```

Primero vamos a separar los datos en training y test. Esta vez, lo haremos con la librería `rsample`.

```
library(rsample)

set.seed(0)
split <- initial_split(datos, prop = 0.5)
train_data <- training(split)
test_data <- testing(split)
```

Para escalar los datos, utilizaremos la librería `recipes`.

```
library(recipes)

data_rec <- recipe(y~., data = train_data) %>%
  step_normalize(all_numeric(), -all_outcomes()) %>%
  prep(train_data)

train_data <- bake(data_rec, train_data)
test_data <- bake(data_rec, test_data)
```

A continuación creamos el modelo usando la librería `parsnip`. Podemos especificar los hyper-parámetros ahora o más adelante.

```
library(parsnip)

model <- linear_reg(penalty = 0.01, mixture = 0.5) %>% #elasticnet
          set_engine("glmnet")
```

Para ajustar el modelo, podemos hacer

```
model <- fit(model, y~., train_data)
```

Para obtener predicciones, hacemos

```
ypred <- predict(model, test_data)
```

En este caso, podemos calcular el error test usando la librería `yardstick`.

```
library(yardstick)
rmse(data.frame(truth = test_data$y, estimate = ypred$.pred), truth, estimate)
```

6.1 Librerías `tune` y `dials`

Los datos test en la práctica son desconocidos. ¿Cómo sabemos que valores de `penalty` y `mixture` nos darán mejores resultados?

Las librerías `tune` y `dials` nos brindan funcionalidades para seleccionar de forma óptima los hyperparámetros de un modelo.

6.1.1 Grid search

Para utilizar grid search, primero creamos el modelo `parsnip`, pero esta vez especificamos que no conocemos los hyper-parámetros óptimos, y los dejamos a la librería `tune` para escoger.

```
library(tune)
model <- linear_reg(penalty = tune(),
                    mixture = tune()) %>%
          set_engine("glmnet")
```

A continuación, creamos las particiones bootstrap para entrenar los diferentes modelos.

```
train_rs <- bootstraps(train_data, times = 2)
```

Especificamos qué metrica queremos optimizar, en este caso como es un problema de regresión, minimizaremos el `rmse`. Podemos incluir varias y luego optimizar con respecto a alguna.

```
metrics <- metric_set(rmse)
```

Finalmente, creamos el grid de hyper-parámetros los cuales vamos a probar en las diferentes particiones bootstrap.

```
library(dials)
params <- parameters( penalty(), mixture() )
grid <- grid_regular(params, levels = 10)
```

Para entrenar los modelos, usamos la función `tune_grid` de la librería `tune`.

```
ctrl <- control_grid(verbose = FALSE)

result <- tune_grid(
  y~.,
  model = model,
  resamples = train_rs,
  metrics = metrics,
  control = ctrl,
  grid = grid
)

result %>% show_best(maximize = FALSE)
```

Podemos evaluar cómo lo hace esta combinación de hyper-parámetros en los datos test.

```
best_config <- show_best(result, maximize = FALSE)
model <- linear_reg(penalty = best_config$penalty[1],
                    mixture = best_config$mixture[1]) %>%
          set_engine("glmnet") %>%
          fit(y~., train_data)
ypred <- predict(model, test_data)
rmse(data.frame(truth = test_data$y, estimate = ypred$.pred), truth, estimate)
```

6.1.2 Random Search

Otro enfoque (superior a grid search) es dejar a `tune` y `dials` escoger los valores de los hiperparámetros para cada iteración. Ambos grid y random search se pueden ejecutar en paralelo.

Para hacer random search procedemos como grid search pero en lugar de especificar un grid, especificamos un número de combinaciones.

```
model <- linear_reg(penalty = tune(),
                    mixture = tune()) %>%
          set_engine("glmnet")
```

```
result <- tune_grid(
  y~.,
  model = model,
  resamples = train_rs,
  metrics = metrics,
  control = ctrl,
  grid = 100
)

result %>% show_best(maximize = FALSE)
```

```
best_config <- show_best(result, maximize = FALSE)
model <- linear_reg(penalty = best_config$penalty[1],
                    mixture = best_config$mixture[1]) %>%
          set_engine("glmnet") %>%
          fit(y~., train_data)
ypred <- predict(model, test_data)
rmse(data.frame(truth = test_data$y, estimate = ypred$.pred), truth, estimate)
```

6.1.3 Optimización Bayesiana

```
model <- linear_reg(penalty = tune(),
                    mixture = tune()) %>%
          set_engine("glmnet")
train_rs <- bootstraps(train_data, times = 10)
metrics <- metric_set(rmse)
ctrl <- control_bayes(verbose = FALSE)

result <- tune_bayes(
  y~.,
  model = model,
  resamples = train_rs,
  metrics = metrics,
  iter = 100,
  control = ctrl
)

result %>% show_best(maximize = FALSE)
```

```
best_config <- show_best(result, maximize = FALSE)
model <- linear_reg(penalty = best_config$penalty[1],
                    mixture = best_config$mixture[1]) %>%
          set_engine("glmnet") %>%
          fit(y~., train_data)
ypred <- predict(model, test_data)
```

```
rmse(data.frame(truth = test_data$y, estimate = ypred$.pred), truth, estimate)
```

Bibliography

Bien, J. and Tibshirani, R. (2011). Hierarchical clustering with prototypes via minimax linkage. *Journal of the American Statistical Association*, 106(495):1075–1084.

Weber, A. and Weber, E. (1974). The structure of world protein consumption and future nitrogen requirements. *European Review of Agricultural Economics*, 2(2):169–192.