Explicacion paso a paso del algoritmo

Pasaremos a explicar la funcion **clustering** paso por paso, siguiendo un ejemplo con el dataset **iris** discretizado (convirtiendo los valores numericos a categoricos, separandolos en 3 rangos: 'Bajo', 'Medio', 'Alto').

Durante la explicación iremos mostrando parte del código, el cual se encontrara completo al final del documento, junto con la url del GitHub donde se podrá visitar las implementaciones y los test hechos.

```
head(iris_cluster)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1         Bajo       Medio         Bajo        Bajo
## 2         Bajo       Medio         Bajo        Bajo
## 3         Bajo       Medio         Bajo        Bajo
## 4         Bajo       Medio         Bajo        Bajo
## 5         Bajo       Medio         Bajo        Bajo
## 6         Bajo        Alto         Bajo        Bajo
```

Llamado de la función

A la función se le pasan 5 parámetros:

1. **datos.ini** = Dataset con datos categoricos sin etiquetar
2. **min_diviciones** = Cantidad mínima de division de grupos
3. **metodo** = Distancia usada para elegir el grupo a dividir ('KL', 'Wasserstein')
4. **max_clusters** = Cantidad maxima de clusters
5. **distrib** = Distribucion a usar para la elección del grupo a dividir ('unif', 'global')

```
#clustering = function(datos.ini, min_diviciones, metodo, max_clusters=15, distrib)
```

Primeras variables

En las primeras lineas del algoritmo se convierten los valores del dataframe a factores y se le agrega una nueva columna "C" donde se le ira asignando el numero de grupo a cada observacion. Esto se guarda en una nueva variable "datos" con una asignacion inicial aleatoria de 2 grupos para todas las observaciones (como los primeros 2 grupos con 0 y 1, $C_i \sim Be(0,5)$).

Luego se crea el grafo "g" donde se indica la dependencia de las variables a la columna C (Figura 1):

```
variables = colnames(datos)
g = empty.graph(variables)
for (i in 1:(length(variables) - 1)) {
  g = set.arc(g, "C", variables[i])
}
```

Este será usado para las funciones **bn.fit** y **score**.

Bucle exterior

Legamos al núcleo del algoritmo: los dos bucles.

Para el **bucle exterior**, que tiene la funcionalidad de aumentar el número de clusters, se incializan las siguientes variables:

1. **c.valores** = Se irán guardando las etiquetas de los grupos ya asignados, agregando una nueva por ciclo.
2. **i_ciclo_exterior** = Contador del ciclo.
3. **mejor.modelo** = Lista de los mejores modelos para cada número de clusters.
4. **bic.mej.ant** = Mejor BIC para el ciclo anterior.

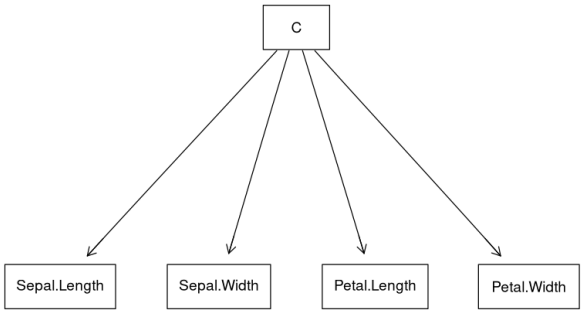


Figure 1: Grafo de dependencias para el dataset iris.

5. **bic.mej.post** = Mejor BIC para el ciclo actual.
6. **bic.mejores** = Vector con el mejor BIC para cada cantidad de clusters.

Las condiciones del ciclo son:

1. Que el bic siga mejorando a medida que se agregan grupos, o que no se haya iterado la cantidad minima de veces indicada en **min_diviciones**
2. Que no se haya superado una cantidad máxima de iteraciones indicada en **max_clusters**

```
#while ((bic.mej.ant < bic.mej.post || i_ciclo_exterior <= min_diviciones) &&
#         i_ciclo_exterior <= max_clusters) {
```

Bucle interior

Dentro se encuentra el **bucle interior**, que tiene el objetivo de encontrar el mejor modelo posible para una cierta cantidad de clusters. Las variables que se inicializan para este bucle son:

1. **i_ciclo_interior** = Contador del ciclo
2. **lista.modelos** = Lista de los modelos de cada iteración
3. **bic.antiguo** = BIC de la iteracion anterior
4. **bic.nuevo** = BIC actual
5. **bic.grupos** = Lista de los BIC del modelo estimado en cada iteración

Las condiciones del ciclo son:

1. Que el BIC siga mejorando respecto al anterior, o que no se haya llegado a las 20 iteraciones.
2. No haberse exedido de las 150 iteraciones (condicion de seguridad para evitar bucle infinito)

```
# while ((bic.antiguo < bic.nuevo || i_ciclo_interior <= 20) && i_ciclo_interior <= 150)
```

Lo primero que se hace al entrar al ciclo es guardar el modelo actual (asignado en la iteracion anterior) en la variable **lista.modelos**, el cual se encuentra en la columna **"C"** de la variable **datos**. Luego se procede a calcular las probabilidades necesarias para, mediante **Naive Bayes**, estimar un nuevo modelo que intente superar al anterior (en términos de la metrica **BIC**).

```
# guardar modelo actual
lista.modelos[[i_ciclo_interior]] = datos["C"]
```

Cálculo de probabilidades

Abora viene la parte **bayesiana** del algoritmo. Se calculan las probabilidades del modelo usando **bn.fit**, de la librería **bnlearn**.

```
bn = bn.fit(g, data = datos, method = "bayes", iss = 10)
```

Como parámetros le pasamos el grafo **g**, la variable **datos** y le indicamos que use el método **bayes**.

Esto calcula las probabilidades que necesitamos para poder usar la formula de **Naive Bayes** y calcular la **probabilidad de cada observacion** X_i **de pertenecer al grupo** C_i, y con esto poder crear un nuevo modelo.

Observemos algunas salidas para entender un poco que devuelve esta función:

1. Indexando en la columna **"C"** obtenemos $P(c_j)$:

```
bn["C"]
```

```
## $C
##
##   Parameters of node C (multinomial distribution)
##
## Conditional probability table:
##        0       1
## 0.46875 0.53125
```

2. Indexando en otra variable X_i obtenemos $P(X_i = x_i | C_i = c_i)$. Probamos con la columna **Sepal.Length**

```
bn["Sepal.Length"]
```

```
## $Sepal.Length
##
##   Parameters of node Sepal.Length (multinomial distribution)
##
## Conditional probability table:
##
##             C
## Sepal.Length         0         1
##        Bajo  0.4222222 0.3607843
##        Medio 0.4488889 0.4784314
##        Alto  0.1288889 0.1607843
```

Con esto tenemos lo suficiente para calcular $P(C_i = c_i | X_i = x_1, ..., X_n = x_n)$

En la variable **pc** guardaremos $P(c_j)$; en **px.c** la $P(x_i|c_i)$; y en **pc.x** la $P(c_i|x_i, ..., x_n)$

```
pc = bn[[length(variables)]]$prob
px.c = matrix(nrow = length(datos[, 1]), ncol = length(variables) - 1)
pc.x = matrix(nrow = length(datos[, 1]), ncol = length(levels(datos[, length(variables)])))
```

Para asignar los valores a **pc** y **px.c** solo hace falta recorrer los valores en el modelo ajustado **bn**. Para calcular los valores de **pc.x** usamos la formula

$$P(C = k \mid \mathbf{x}) \propto P(C = k) \prod_{j=1}^{n} P(x_j \mid C = k)$$

y luego normalizamos los resultados.

Creación del nuevo modelo

En este paso es donde se ve el carácter difuso del algoritmo, ya que para asignarle un grupo nuevo a cada observación usamos una multinomial de la siguiente manera:

$$C_i \sim \text{Multinomial}(1; P(C = 1 \mid \mathbf{x}_i), P(C = 2 \mid \mathbf{x}_i), \ldots, P(C = K \mid \mathbf{x}_i))$$

De esta forma, cada X_i tiene probabilidad mayor que cero de pertencer a cualquier grupo C_i.

El nuevo modelo lo guardamos en la columna **"C"** de la variable **datos**, asignando para cada X_i el grupo k, donde k es la posicion de C_i que se encuentra un 1.

Lo ultimo que hacemos antes de terminar el **ciclo interior** es calcular el **BIC** del modelo actual y guardarlo dentro de la lista **bic.grupos**.

División del siguiente grupo

Al salir del **bucle interior** ya tenemos el mejor modelo para un numero **n** de grupos. Ahora lo que haremos es elegir uno para dividirlo, asignar un nuevo modelo al azar dentro de este y volver a iterar para encontrar un nuevo modelo que se ajuste para **n+1** grupos.

En este trabajo, como dijimos anteriormente, hemos implementado un nuevo método usando la **distancia de Wasserstein** para comparar la distribucion de cada grupo con otra fija (que ahora especificaremos) y asi encontrar el grupo "menos definido".

Las distribuciones con las que medimos a cada grupo son:

1. Distribucion uniforme: La comparación con una distribución uniforme mide qué tan lejos está la distribución actual del grupo de una situación completamente "indefinida" o aleatoria. Una distribución uniforme indica que todos los valores tienen la misma probabilidad, lo que refleja un grupo sin estructura o patrones claros.
2. Distribución global: La comparación con la distribución global mide qué tan representativo es un grupo respecto a todo el conjunto de datos. Si un grupo es similar a la distribución global, significa que no aporta información nueva o distintiva.

La idea de comparar con una distribución global también fue implementada por nosotros, y encontramos que en algunos experimentos se vieron mejoras con esta implementación.

Entonces tenemos 2 funciones de disimilaridad (**Kullback-Leibler** y **Wasserstein**) y dos distribuciones para realizarla (**uniforme** y **global**)

En todos los casos el procedimiento es el siguiente:

1. Se inicializa un vector donde se guardarán las medidas de distancia de las variables X_i pertenecientes al grupo C_i con respecto a una distribución de referencia (uniforme o global).
2. Se calcula la distribución observada de cada variable X_i en el grupo C_i, representando las proporciones de cada categoría en el grupo. Dependiendo el caso:
    - **Distribución Uniforme:** La distribución de referencia es uniforme, es decir, cada categoría tiene igual probabilidad.

- **Distribución Global:** La distribución de referencia se calcula como las proporciones globales de cada categoría en todo el conjunto de datos.

3. Se calcula la distancia entre la distribución observada y la de referencia utilizando una métrica específica:
   - **Distancia de Wasserstein:** Para medir la diferencia entre las distribuciones como el "transporte óptimo".
   - **Divergencia KL (Kullback-Leibler):** Para medir la disimilitud entre la distribución observada y la de referencia.
4. Se suma la distancia calculada para todas las variables dentro del grupo, obteniendo un puntaje total para el grupo.
5. Se repite este proceso para todos los grupos C_i actuales.
6. El próximo grupo a dividir será aquel con la menor distancia total al comparar con la distribución de referencia, ya que indica que el grupo es menos definido o tiene mayor dispersión en comparación con los demás.

Con el grupo i elegido, este se divide en 2 y se crea un nuevo modelo asignando aleatoriamente un nuevo grupo para cada X_i dentro del grupo i.

Selección del mejor modelo

Recordemos las siguientes variables:

1. **bic.mejores[n]** = mejor BIC para **n** clusters.
2. **mejor.modelo[n]** = modelo con mejor BIC para **n** clusters

Con esto ya tenemos todo para poder devolver el modelo mejor ajustado segun la métrica **BIC**.

Nosotros modificamos el algoritmo para que si **min_divisiones** = **max_clusters** = **n** el algoritmo devuelva un modelo de **n** clusters, para usarlo en casos donde sabemos la cantidad de grupos a estimar.

```
# Creación de la salida
#  if(min_divisiones == max_clusters){
#    mejor.mejor = mejor.modelo[[n]]
#  } else {
#    mejor.mejor = mejor.modelo[[which(bic.mejores == max(bic.mejores))]]
#  }
#  salida = list(table(mejor.mejor), mejor.mejor, bic.mejores)
#  return(salida)
```

Resultados

Iris

Para terminar con el ejemplo del principio probamos la funcion sobre este dataset discretizado:

```
set.seed(49)
etiquetas_reales <- as.numeric(iris$Species) - 1  # Convertir a 0, 1, 2,...

resultados <- clustering(iris_cluster, min_divisiones = 5, metodo = 'Wasserstein', max_clusters = 10, di

print(resultados[[1]])  # Tabla de conteo
```

```
## C
##  0  1  2
## 52 50 48
```

Medimos el accuracy del modelo comparando las etiquetas estimadas con las reales:

```
etiquetas_estimadas <- as.numeric(unlist(resultados[[2]])) - 1

confusion_matrix <- table(Real = etiquetas_reales, Estimado = etiquetas_estimadas)

# usamos matriz de confusion para reasignar clusters
asignacion <- solve_LSAP(confusion_matrix, maximum = TRUE)
# reasignar etiquetas estimadas
etiquetas_reasignadas <- as.numeric(asignacion)[etiquetas_estimadas + 1] - 1
confusion_matrix <- table(Real = etiquetas_reales, Estimado = etiquetas_reasignadas)
print(confusion_matrix)
```

```
##     Estimado
## Real  0  1  2
##    0 50  0  0
##    1  0 48  2
##    2  0  4 46
```

```
# Calcular el accuracy
accuracy <- mean(etiquetas_reasignadas == etiquetas_reales)
print(paste("Accuracy:", accuracy))
```

```
## [1] "Accuracy: 0.96"
```

Como vemos en este dataset funciona bastante bien el algoritmo, aunque sea bastante de juguete y lo hayamos discretizado.

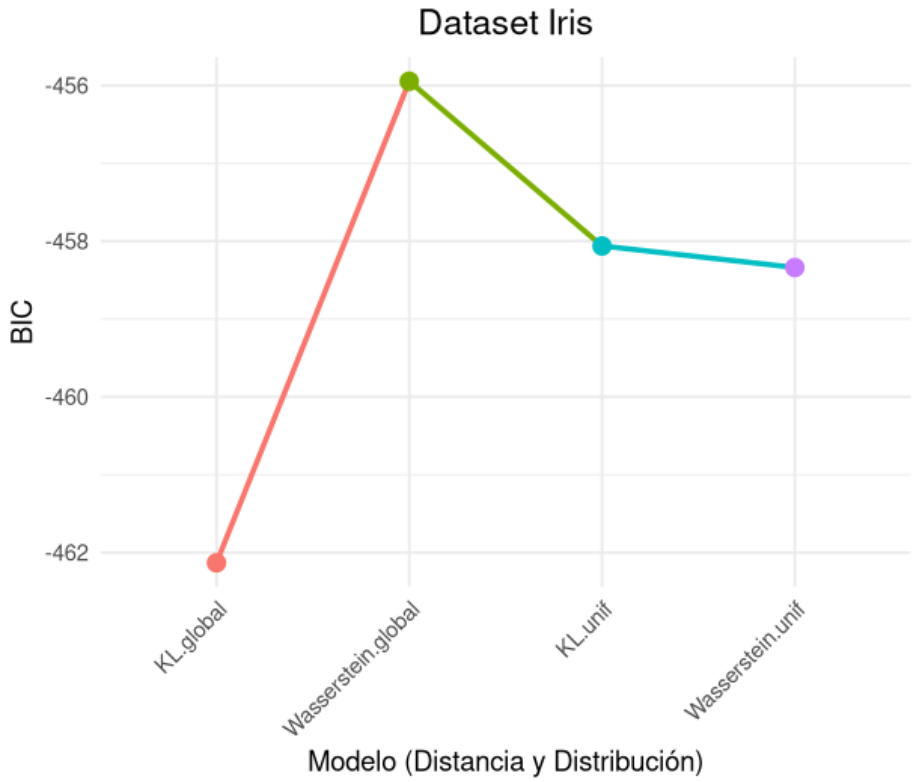
Resultados con experimentos

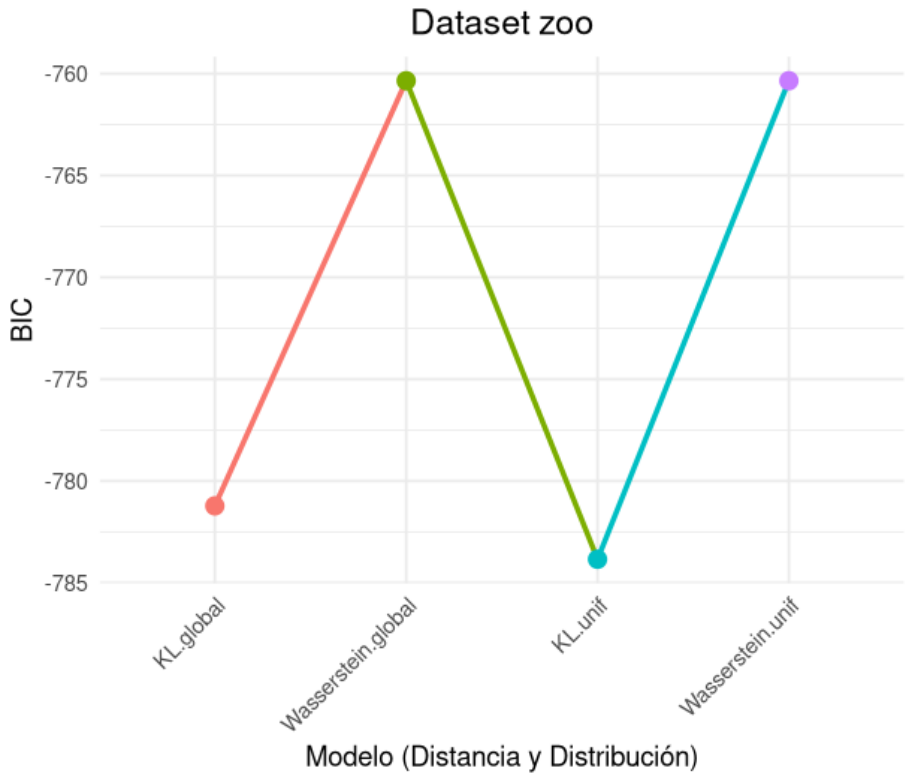
Experimentamos con 3 datasets reales categoricos (ademas de iris) y comparamos los BIC obtenidos para las distintas combinaciones de parametros entre distancias y distribuciones.

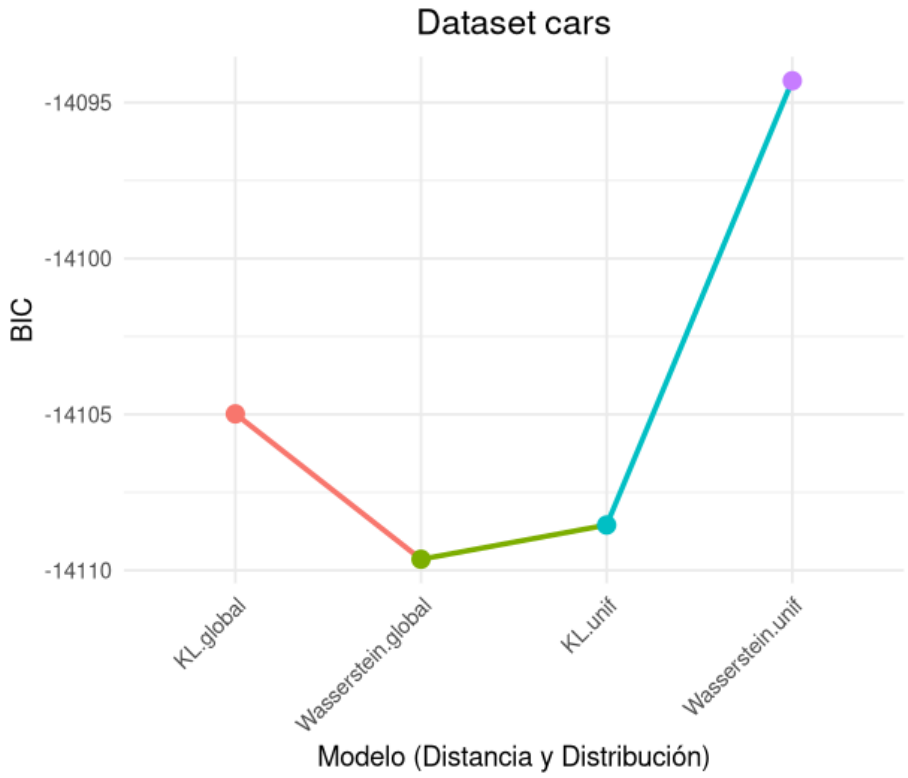
Datasets:

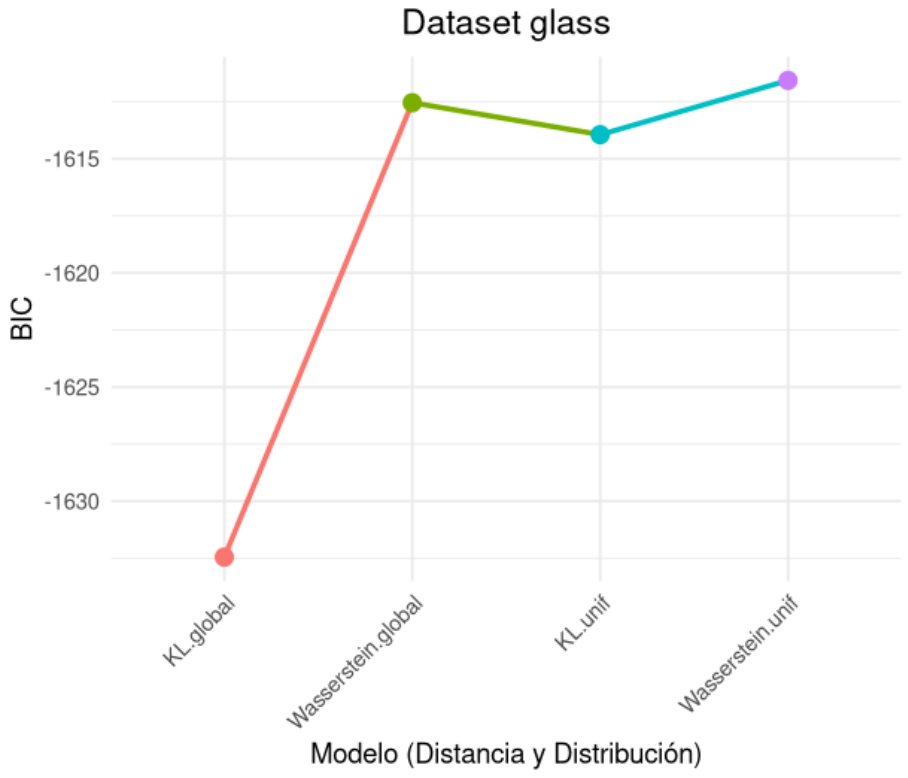
1. **Zoo Dataset:** Contiene características de 101 animales divididos en 7 clases taxonómicas diferentes. Cada animal está descrito por 16 atributos categóricos y un atributo numérico (número de patas). Fuente: UCI Machine Learning Repository.
2. **Cars Dataset:** Clasifica automóviles en función de su aceptabilidad. Cada registro representa una combinación de características del automóvil. Fuente: UCI Machine Learning Repository.
3. **Glass Dataset:** incluye datos químicos sobre 214 muestras de vidrio. Cada muestra está clasificada en una de 7 categorías de vidrio. Los atributos son medidas numéricas de óxidos químicos presentes en el vidrio, como SiO2 y Na2O. Fuente: UCI Machine Learning Repository.

Graficos comparativos de modelos:









En los 4 tests se ve que la distancia de Wasserstein consigue las mejores métricas. Aunque no son pruebas suficientes para demostrar que es consistentemente mejor, da un buen indicio de que se podría llegar a buenos

resultados con este enfoque.

Código en R:

Como comentamos anteriormente, nos basamos en el algoritmo del trabajo "Clustering en Redes Bayesianas"[1]. El siguiente código es una modificacion del mismo, con el agregado de la **distancia de Wasserstein** y la **distribucion global** para el aumento de clusters.

```
#clustering = function(datos.ini, min_divisiones, metodo, max_clusters=15, distrib) {
#
#  # Se transforma la matriz de datos en factores.
#  datos.ini = as.data.frame(datos.ini)
#  datosfac = as.data.frame(lapply(datos.ini, factor))
#
#  # Se añade la variable de clase y se genera el primer modelo al azar.
#  datos = cbind(datosfac, as.factor(rbinom(length(datosfac[, 1]), 1, 0.5)))
#  names(datos)[length(names(datos))] = "C"
#
#  # Se crea el grafo con la estructura del modelo Naive-Bayes.
#  variables = colnames(datos)
#  g = empty.graph(variables)
#  for (i in 1:(length(variables) - 1)) {
#    g = set.arc(g, "C", variables[i])
#  }
#
#  # Inicialización de variables
#  c.valores = c(0, 1)
#  i_ciclo_exterior = 1
#  mejor.modelo = list()
#  bic.mej.ant = -Inf
#  bic.mej.post = 0
#  bic.mejores = vector()
#
#  # Inicio del bucle exterior
#  while ((bic.mej.ant < bic.mej.post || i_ciclo_exterior <= min_divisiones) &&
#         i_ciclo_exterior <= max_clusters) {
#
#    # Inicialización de variables
#    i_ciclo_interior = 1
#    lista.modelos = list()
#    bic.antiguo = -Inf
#    bic.nuevo = score(g, data = datos, type = "bic")
#    bic.grupos = bic.nuevo
#
#    # Inicio del bucle interior
#    while ((bic.antiguo < bic.nuevo || i_ciclo_interior <= 20) && i_ciclo_interior <= 150) {
#
#      lista.modelos[[i_ciclo_interior]] = datos["C"]
#
#      # Cálculo de probabilidades en el modelo
#      bn = bn.fit(g, data = datos, method = "bayes", iss = 10)
#      # P(Ci)
#      pc = bn[[length(variables)]]$prob
#      # P(Xi = xi | Ci)
```

```
#       px.c = matrix(nrow = length(datos[, 1]), ncol = length(variables) - 1)
#       # P(ci | x1, ..., xn)
#       pc.x = matrix(nrow = length(datos[, 1]), ncol = length(levels(datos[, length(variables)])))
#
#       # Recorremos los resultados de bn y asignamos las probabilidades a las variables
#       for (k in 1:length(c.valores)) {
#         for (j in 1:(length(variables) - 1)) {
#           px.c[, j] = bn[[j]]$prob[datos[, j], k]
#         }
#         pc.x[, k] = pc[k] * apply(px.c, 1, prod)
#       }
#       pc.x = pc.x / apply(pc.x, 1, sum)
#
#       # Generación del modelo nuevo a partir de pc.x
#       # usando pc.x como parametros de la multinomial
#       rand = rmultinom(1, 1, pc.x[1, ])
#       for (i in 2:length(datos[, 1])) {
#         rand = cbind(rand, rmultinom(1, 1, pc.x[i, ]))
#       }
#       for (l in 1:length(c.valores)) {
#         datos[which(rand[l, ] == 1), length(variables)] = l - 1
#       }
#
#       # Cálculo del BIC
#       bic.antiguo = bic.nuevo
#       bic.nuevo = score(g, data = datos, type = "bic")
#       bic.grupos = append(bic.grupos, bic.nuevo)
#
#       i_ciclo_interior = i_ciclo_interior + 1
#     }
#     # Final del bucle interior
#
#     # Selección del mejor modelo
#     bic.mejores[i_ciclo_exterior] = max(bic.grupos)
#     mejor.modelo[[i_ciclo_exterior]] = lista.modelos[[min(which(bic.grupos == bic.mejores[i_ciclo_exte
#
#     dividir = -1
#     # PARTE DONDE SE ELIJE EL PROXIMO CLUSTER A DIVIDIR
#     if (metodo == "Wasserstein") {
#       if(distrib == 'unif'){
#       # EL PROXIMO GRUPO A DIVIDIR ES AQUEL CON DISTANCIA DE WASSERSTEIN
#       # MAS CERCANA A LA DISTRIBUCION UNIFORME
#         disim = vector(length = length(variables) - 1) # distancias de cada variable
#         # suma de las distancias de las variables de cada grupo
#         wasserstein.total = vector(length = length(c.valores))
#
#         for (i in 1:length(c.valores)) {
#           grupo = datos[which(datos$C == i - 1), ]
#           for (q in 1:(length(variables) - 1)) {
#             distrib_observada <- table(grupo[[variables[q]]]) / nrow(grupo)
#             distrib_uniforme <- rep(1 / length(distrib_observada), length(distrib_observada))  # Unifo
#
#             disim[q] <- wasserstein1d(distrib_observada, distrib_uniforme)
```

```
#           }
#           wasserstein.total[i] <- sum(disim)
#         }
#         dividir <- which.min(wasserstein.total) - 1
#
#       } else if(distrib == 'global'){
#       # EL PROXIMO GRUPO A DIVIDIR ES AQUEL CON DISTANCIA DE WASSERSTEIN
#       # MAS CERCANA A LA DISTRIBUCION GLOBAL DE CADA VARIABLE
#         disim = vector(length = length(variables) - 1) # distancias de cada variable
#         # suma de las distancias de las variables de cada grupo
#         wasserstein.total = vector(length = length(c.valores))
#
#         # calcular la distribución global para cada variable
#         distrib_global <- lapply(variables, function(var) {
#           table(datos[[var]]) / nrow(datos)
#         })
#
#         for (i in 1:length(c.valores)) {
#           grupo <- datos[which(datos$C == i - 1), ]
#           for (q in 1:(length(variables) - 1)) {
#             distrib_observada <- table(grupo[[variables[q]]]) / nrow(grupo)
#             distrib_referencia <- distrib_global[[q]]
#
#             disim[q] <- wasserstein1d(distrib_observada, distrib_referencia)
#           }
#           wasserstein.total[i] <- sum(disim)
#         }
#         dividir <- which.min(wasserstein.total) - 1
#       }
#     } else if (metodo == "KL") {
#       if(distrib == 'unif'){
#       # EL PROXIMO GRUPO A DIVIDIR ES AQUEL CON DISIMILITUD DE KL
#       # MENOR A LA DISTRIBUCION UNIFORME
#         disim = vector(length = length(variables) - 1)
#         kl.total = vector(length = length(c.valores))
#         for (i in 1:length(c.valores)) {
#           grupo = datos[which(datos$C == i - 1), ]
#           for (q in 1:(length(variables) - 1)) {
#             probs = table(grupo[, q]) / length(grupo[, q])
#             disim[q] = kl.dist(probs, rep(1 / length(probs), length(probs)), base = exp(1))[[3]]
#           }
#           kl.total[i] = sum(disim)
#         }
#         dividir = which.min(kl.total) - 1
#       } else if(metodo == 'global'){
#       # EL PROXIMO GRUPO A DIVIDIR ES AQUEL CON DISIMILITUD DE KL
#       # MENOR A LA DISTRIBUCION GLOBAL DE CADA VARIABLE
#         disim = vector(length = length(variables) - 1)
#         kl.total = vector(length = length(c.valores))
#         # calcular la distribución global para cada variable
#         distrib_global <- lapply(variables, function(var) {
#           table(datos[[var]]) / nrow(datos)
#         })
```

```
#         for (i in 1:length(c.valores)) {
#           grupo = datos[which(datos$C == i - 1), ]
#           for (q in 1:(length(variables) - 1)) {
#             probs = table(grupo[, q]) / length(grupo[, q])
#             disim[q] = kl.dist(probs, , base = exp(1))[[3]]
#           }
#           kl.total[i] = sum(disim)
#         }
#         dividir = which.min(kl.total) - 1
#       }
#     } else {
#       stop("Por favor inserte un método válido")
#     }
#
#     # Adición del nuevo cluster
#     c.valores = append(c.valores, length(c.valores))
#     levels(datos$C) = c.valores
#
#     datos[datos$C == dividir, "C"] = as.factor(sample(c(dividir, length(c.valores) - 1), length(datos[
#
#     # Actualización de BIC de los mejores modelos
#     if (i_ciclo_exterior > 1) {
#       bic.mej.ant = bic.mejores[i_ciclo_exterior - 1]
#       bic.mej.post = bic.mejores[i_ciclo_exterior]
#     }
#
#     i_ciclo_exterior = i_ciclo_exterior + 1
#   }
#   # Final del bucle exterior
#
#   # Creación de la salida
#   if(min_divisiones == max_clusters){
#     mejor.mejor = mejor.modelo[[min_divisiones]]
#   } else {
#     mejor.mejor = mejor.modelo[[which(bic.mejores == max(bic.mejores))]]
#   }
#   salida = list(table(mejor.mejor), mejor.mejor, bic.mejores)
#   return(salida)
# }
```