UNIVERSIDAD
DE GRANADA

METAHEURÍSTICAS

GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 1

PROBLEMA DEL AGRUPAMIENTO CON RESTRICCIONES (PAR)

Autor
Fernando Vallecillos Ruiz

DNI
77558520J

E-Mail
nandovallec@correo.ugr.es

Grupo de prácticas
MH1 Miércoles 17:30-19:30

Rama
Computación y Sistemas Inteligentes

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

CURSO 2019-2020

Índice

1. Descripción del problema

El problema que vamos a analizar en esta práctica se trata del Agrupamiento con Restricciones (PAR). Este es una variación del problema de agrupamiento o *clustering*, el cual persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos. De esta forma, el agrupamiento es una técnica de aprendizaje no supervisado ya que permite descubrir grupos (no conocidos) en un conjunto de datos y agrupar los datos similares. Cabe destacar que no es un algoritmo en específico, sino un problema pendiente de solución. Existen multitud de algoritmos que resuelven este problema los cuales difieren en su definición de *cluster* y su método de búsqueda.

El problema de Agrupamiento con Restricciones es una generalización del problema de agrupamiento. Incorpora al proceso nueva información: las restricciones. Esto provoca un cambio en el tipo de tarea que pasa a ser semi-supervisada. Intentaremos encontrar una partición $C = \{c_1, c_2, \ldots, c_k\}$ del conjunto de datos X con n instancias que minimice la desviación general y cumpla con las restricciones en el conjunto R.

1.1. Variantes del problema

Variantes según tipo de restricciones[1]:

- *Cluster-level constraints*: se definen requerimientos específicos a nivel de *clusters* como:
  - Número mínimo/máximo de elementos
  - Distancia mínima/máxima de elementos
- *Instance-level constraints*: se definen propiedades entre pares de objetos tales como la pertenencia o no de elementos al mismo cluster

Variantes según la interpretación de las restricciones:

- Basados en métricas(*metric-based*): El algoritmo de *clustering* utiliza una distancia métrica. Esta métrica sera diseñada para que cumpla con todas las restricciones.
- Basados en restricciones(*constraint-based*): El algoritmo es modificados de manera que las restricciones se utilizan para guiar el algoritmo hacia una partición C más apropiada. Se lleva a cabo mediante la modificación de la función objetivo del *clustering*.

1.2. Formalización del problema

El conjunto de datos es una matriz X de $n \times d$ valores reales. El conjunto de datos esta formado por n instancias en un espacio de d dimensiones notadas como:

$$\vec{x_i} = \{x_{[i,1]}, \cdots, x_{[i,d]}\}$$

Un *cluster* c_i consiste en un subconjunto de instancias de X. Cada *cluster* c_i tiene asociada una etiqueta (nombre de *cluster*) l_i.

Para cada *cluster* c_i se puede calcular su centroide asociado:

$$\vec{\mu_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$$

A partir de estos, se puede calcular la distancia media intra-cluster:

$$\overline{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \|\vec{x_j} - \vec{\mu_i}\|$$

Para calcular la desviación general de la partición C podemos:

$$\overline{C} = \frac{1}{k} \sum_{c_i \in C} \overline{c_i}$$

Dada una partición C y el conjunto de restricciones, definimos *infeasibility* como el número de restricciones que C incumple. Definimos $V(\vec{x_i}, \vec{x_j})$ como la función que, dada una pareja de instancias, devuelve 1 si viola una restricción y 0 en otro caso:

$$infeasibility = \sum_{i=0}^{n} \sum_{j=0}^{n} V(\vec{x_i}, \vec{x_j})$$

Entonces, podemos formularlo como:

$$\text{Minimizar } f = \overline{C} + (infeasibility * \lambda)$$

donde λ es un parámetro de escalado para dar relevancia a *infeasibility*. Si se establece correctamente, el algoritmo optimiza simultáneamente el número de restricciones incumplidas y la desviación general.

2. Descripción de los algoritmos

2.1. Consideraciones previas

Antes de realizar una descripción formal de cada algoritmo, necesitamos describir aspectos comunes entre estos: esquema de representación de soluciones, pseudocódigo de operadores comunes o la función objetivo.

Comenzaremos hablando de las estructuras de datos utilizadas. Se ha utilizado una matriz X de $n \times d$ para guardar los datos. Se ha querido aprovechar al máximo la capacidad de paralelización de operaciones, por ello se optó por añadir las columnas necesarias a X para concentrar en una sola estructura todos los datos necesarios para calcular nuestros parámetros. Por ello, en cada algoritmo se añadirán columnas diferentes dependiendo de las necesidades de este. No obstante, existen dos columnas añadidas comunes a cualquier algoritmo. Estas son la columna *cluster*, la cual representa a que *cluster* pertenece la instancia. Y la columna *Distance to Cluster (DC)* que representa la distancia de la instancia al centroide de su cluster. De aquí en adelante, serán referidas como $x_{cluster}$ y x_{DC} respectivamente.

Para los centroides se ha escogido una matriz M de $k \times d$ siendo k el número de *clusters*. De esta forma, para cualquier instancia x, $x_{cluster}$ indicara el índice en la matriz M. Por ello, M puede ser referido como vector de centroides. Para calcularlos, se han calculado los vectores promedios de las instancias.

Algorithm 1: Calcular centroides

```
1 CalcularCentroides (X)
      input  : Matriz de datos X
      output: Vector de centroides M
2     foreach x ∈ X do
3         sum_{l_{x_i}} ← sum_{l_{x_i}} + x
4         count_{l_{x_i}} ← count_{l_{x_i}} + 1
5     centroids = sum/count
6     return centroids
```

También se ha optado por representar el conjunto R de restricciones en una matriz de $n \times n$. Vamos a contar con solo 2 tipos de restricciones *Must-Link (ML)* y *Cannot-Link (CL)*. Dada una pareja de instancias, se establece una restricción *ML* si deben pertenecer al mismo *cluster* y de tipo *CL* si no pueden pertenecer al mismo *cluster*. Se representa una restricción *ML* entre las instancias x_i e x_j si $R[i][j] == 1$. Por otra parte, se representa una restricción *CL* entre las instancias x_i e x_j si $R[i][j] == -1$. Como se puede ver, la operación para calcular *infeasibility* dado

una matriz de datos X es trivial pero explicaremos brevemente su implementación y modularización.

Para calcular *infeasibility* dado una matriz de datos X, podríamos mirarlo como la suma de *infeasibility* de cada instancia $x \in X$.

Algorithm 2: Calcular Infeasibility

```
1  CalcularInfeasibility (X)
       input  : Matriz de datos X
       output: Valor de infeasibility
2      total ← 0
3      for i ← 0 to N − 1 do
4          total ← total + CalcularInfeasibilityRowPartial(X, i)
5      return total
```

Se necesita entonces, calcular el valor de *infeasibility* para cualquier $x \in X$. Paro ello, se recorren las restricciones asociadas a la instancia seleccionada. Se descarta la restricción de una instancia consigo misma. Si la restricción es de tipo *ML* o *CL* se comprueba si se cumple o no, acumulando puntos si fuese necesario.

Algorithm 3: Calcular Infeasibility Row

```
1  CalcularInfeasibilityRow (X, r)
       input  : Matriz de datos X, Índice r
       output: Valor de infeasibility asociado a x_r
2      total ← 0
3      for i ← 0 to N − 1 do
4          if R_r[i] == 1 then
5              if X[i]_cluster ≠ X[r]_cluster then
6                  total ← total + 1
7          else if R_r[i] == −1 then
8              if X[i]_cluster == X[r]_cluster then
9                  total ← total + 1
10     return total
```

Sin embargo, si se llamase a esta función para calcular *infeasibility* de toda la matriz X, se acabaría con el doble de puntuación. Este problema se puede solucionar de forma fácil devolviendo solo la mitad del valor acumulado. Subyace un problema de eficiencia, al ser una matriz, se calculan todas las inversas de las restricciones. Por ello, se puede crear una función para calcular de forma parcial el valor de una fila. Esta función sera llamada solo cuando queramos calcular

infeasibility de todo el conjunto.

Algorithm 4: Calcular Infeasibility Row Partial

```
1  CalcularInfeasibilityRowPartial (X, r)
       input  : Matriz de datos X, Índice r
       output: Valor de infeasibility asociado a x_r
2      total ← 0
3      for i ← r + 1 to N − 1 do
4          if R_r[i] == 1 then
5              if X[i]_cluster ≠ X[r]_cluster then
6                  total ← total + 1
7          else if R_r[i] == −1 then
8              if X[i]_cluster == X[r]_cluster then
9                  total ← total + 1
10     return total
```

Se describen a continuación las funciones necesarias para calcular los parámetros de la función objetivo. Como se ha descrito antes, esta tendrá la siguiente forma:

$$f = \overline{C} + (infeasibility * \lambda)$$

El valor de λ se ha calculado como el cociente entre la mayor distancia entre dos instancias del conjunto y el número de restricciones:

$$\lambda = \frac{\lceil D \rceil}{|R|}$$

El último parámetro es $\overline{C}$. Para calcularlo, primero se asignará la distancia desde cada instancia $x \in X$ a su *cluster*. Para ello, simplemente se calcula la

distancia euclídea entre ambos puntos y se asigna a su columna correspondiente.

Algorithm 5: Calcular Distancia a Centroides

CalcularDistanceCluster (X, M)
- **input :** Matriz de datos X, Vector de Centroides M
- **output:** Matriz de datos X
- **foreach** $x \in X$ **do**
  - $a \leftarrow x$
  - $b \leftarrow M\,[x_{cluster}]$
  - $x_{DC} \leftarrow \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}$
- **return** X

Con las distancias a mano, se puede calcular cada $\overline{c_i}$ y por tanto $\overline{C}$ como la media de estos.

Algorithm 6: Calcular Desviación General

CalcularDesviacion (X)
- **input :** Matriz de datos X
- **output:** Valor de Desviación General *average*
- $sum_{l_{x_i}} \leftarrow [k]$
- $count_{l_{x_i}} \leftarrow [k]$
- **foreach** $x \in X$ **do**
  - $sum_{l_{x_i}} \leftarrow sum_{l_{x_i}} + x_{DC}$
  - $count_{l_{x_i}} \leftarrow count_{l_{x_i}} + 1$
- $average = mean(sum/count)$
- **return** *average*

Como se puede ver, los dos últimos algoritmos están muy entrelazados entre si. Han sido escritos de forma separada para comprender su comportamiento de una forma fácil y simple. Sin embargo, han sido implementado de forma conjunta para mejorar la eficiencia temporal.

Con esto, se terminan las consideraciones comunes de los algoritmos usados. Se pasará entonces a explicar cada algoritmo en mayor profundidad.

2.2. Algoritmos de comparación: COPKM

Para comparar nuestras metaheurísticas, se ha decidido usar el algoritmo *COP-KMeans* introducido por Wagstaff et al.[2] con una interpretación débil de las

restricciones. Se comenzará explicando cada una de las funciones necesarias para la implementación, terminando con el pseudocódigo completo.

Primero, se explica como obtener los primeros centroides. Existen diferentes heurísticas solo para esta tarea pero se ha decidido crearlos de forma aleatoria. Para cada característica en la matriz de datos, se calcula el mínimo y máximo. De esta forma, podemos obtener centroides dentro del espacio de características. Sin embargo, pueden haber *outliers* en los datos que empeoren de manera significativa esta técnica, así que se opta por dividir el valor *random* para centrarlo de forma suave.

Algorithm 7: Generar Centroides Aleatorios

```
1 GenerateRandomCentroids (X)
    input  : Matriz de datos X
    output: Vector de centroides M
2   M ← [ ][ ]
3   for h ← 0 to d do
4       min ← MinValue(X_h)
5       max ← MaxValue(X_h)
6       for i ← 0 to k do
7           M ← RandomBetween(min, max)/1.5
8   return M
```

Teniendo ya unos centroides calculados, se puede pasar a la primera asignación de la matriz de datos. Aunque el pseudocódigo de la función parezca largo, ha sido diseñado de una forma simple y eficiente para tener en cuenta las situaciones

extremas posibles. Se irá descomponiendo y explicando en varias partes.

Algorithm 8: Inicialización Matriz

```
1  InicializarMatriz (X, M)
       input  : Matriz de datos X, Vector de centroides M
       output: Matriz de datos X actualizada
2      ClusterCount[k] ← 0
3      for w ← 0 to n − 1 do
4          i ← randomIndex[w]
5          x ← X_i
6          for CIndex ← 0 to k do
7              x_Cluster = CIndex
8              x_InfCIndex = CalcularInfeasibilityRowFirst(X, i)
9          own ← LeastInfeasibilityClusterOf(x)
10         x_cluster ← own
11         ClusterCount_own ← ClusterCount_own + 1
12         for CIndex ← 0 to k do
13             if CIndex ≠ own & & x_InfOwn == x_InfCIndex then
14                 ownDistance ← Distancia(x, M_own)
15                 otherDistance ← Distancia(x, M_CIndex)
16                 if otherDistance > ownDistance then
17                     ClusterCount_own ← ClusterCount_own − 1
18                     x_cluster ← CIndex
19                     own ← CIndex
20                     ClusterCount_own ← ClusterCount_own + 1
21     for i ← 0 to k do
22         if ClusterCount_i == 0 then
23             w ← i
24             while CountCluster_{X_{w_Cluster}} ≥ k & & w < n − 1 do
25                 w ← w + 1
26             X_{w_Cluster} ← i
27     return X
```

Primeramente, se define un bucle para iterar sobre los datos según un índice aleatorio. Por cada instancia x obtenida, le asignaremos en su columna correspondiente un *cluster*. Se calcula *infeasibility* suponiendo x pertenece a cada uno de los *clusters* y se guarda el resultado en una columna. Es decir, se han creado k columnas adicionales que guardan el valor $infeasibility$ con respecto a cada uno de los k *clusters*.

Esto sería suficiente para una gran parte de los casos. Se ha querido aumentar la seguridad de la heurística. Por ello, con el segundo bucle se comprueba si existe otro *cluster* con el mismo valor *infeasibility*. Si es así, se elige aquel con menor distancia al centroide respectivo.

Se ha querido asegurar otro caso extremo. Puede darse el caso de que algún *cluster* sea vacío. Por ello, nos aseguramos a partir de la linea **21**. En cada instancia, se ha llevado la cuenta a donde se asignaba en *ClusterCount*. Por ello, se puede asegurar que ningún *cluster* esté vacío. Si lo estuviera, se ha buscado la siguiente instancia cuyo *cluster* tenga k elementos o más y se ha cambiado su pertenencia al *cluster* vacío. Se realiza una búsqueda de esta forma para evitar que el cambio deje a este segundo *cluster* vacío (incluso cuando varios *cluster* tomen de él).

Se podría clarificar que la función *CalcularInfeasibilityRowFirst* es muy similar a la función explicada anteriormente *CalcularInfeasibilityRow*. Con una pequeña diferencia, comprueba si la segunda instancia x_i(a la que comparamos), ha sido asignada o no. Si no, no comprueba las restricciones pues no tiene nada con lo que comparar.

Por último, se explicará la función de asignación regular. Será la usada en bucle en nuestra implementación y es muy similar a la inicialización de matriz

previamente vista.

Algorithm 9: Asignación Regular

```
1  RegularAssignation (X, M)
       input  : Matriz de datos X, Vector de centroides M
       output: Matriz de datos X actualizada
2      ClusterCount ← CountClusters(X)
3      for w ← 0 to n − 1 do
4          i ← randomIndex[w]
5          x ← X_i
6          own ← x_cluster
7          if ClusterCount_own == 1 then
8              NextIterationFor()
9          ClusterCount_own ← ClusterCount_own − 1
10         for CIndex ← 0 to k do
11             x_Cluster = CIndex
12             x_InfCIndex = CalcularInfeasibilityRow(X, i)
13         own ← LeastInfeasibilityClusterOf(x)
14         x_cluster ← own
15         ClusterCount_own ← ClusterCount_own + 1
16         for CIndex ← 0 to k do
17             if CIndex ≠ own & & x_InfOwn == x_InfCIndex then
18                 ownDistance ← Distancia(x, M_own)
19                 otherDistance ← Distancia(x, M_CIndex)
20                 if otherDistance > ownDistance then
21                     ClusterCount_own ← ClusterCount_own − 1
22                     x_cluster ← CIndex
23                     own ← CIndex
24                     ClusterCount_own ← ClusterCount_own + 1
25     return X
```

Se puede apreciar que la estructura es similar a la inicialización de la matriz. Se explicará la función basándose en las diferencias. Se ha necesitado contar los el número de instancias de cada *cluster* al comienzo de la función. Se ha realizado este cambio ya que se necesita comprobar que no se deje ningún *cluster* vacío. Si se verifica que tiene más de un elemento, podemos comprobar los puntos *infeasibility* con respecto a otros *cluster*. El resto es igual que el previamente explicado.

Con todo lo necesario ya explicado, se pasa al pseudocódigo del programa principal. Toda la implementación se centra en torno a un bucle *while* principal. Primero se inicializa un índice aleatorio de tamaño n, el cual servirá para aleatori-

zar los acceso a X. Como se ha explicado, se inicializan el vector M de centroides y la estructura de datos X. Creamos un vector $OldClusters$. El bucle consiste de tres sencillos pasos:

1. Guardamos la columna con la asignación de *clusters* en nuestra variable $OldClusters$.
2. Calculamos los nuevos centroides a partir de la asignación de *clusters* actual.
3. Realizamos la nueva asignación de *clusters* a la instancias.

Como condición del bucle, se asegura que haya habido algún cambio en la asignación de *clusters* a las instancias.

Algorithm 10: COP-KMeans Débil

```
1  COPKmeansSoft (X)
       input  : Matriz de datos X, Matriz de restricciones R
       output: Matriz de datos X con asignaciones a clusters
2      randomIndex ← RandomShuffle(n)
3      M ← GenerateRandomCentroids(X)
4      X ← FirstAssignation(X, M)
5      OldClusters ← [ ]
6      while OldClusters ≠ X_Clusters do
7          OldClusters ← X_Clusters
8          M ← CalcularCentroides(X)
9          X ← RegularAssignation(X, M)
10     return X
```

Con esto, se acaba la explicación de nuestro de algoritmo de comparación. Como se ha visto, este algoritmo prioriza siempre *infeasibility* sobre la distancia (aunque la toma en cuenta en casos de empate). También es un algoritmo que luce por su simpleza y velocidad con la que alcanza soluciones factibles.

2.3. Algoritmo de Búsqueda Local

A continuación se describe el algoritmo de búsqueda local considerando el esquema del primer mejor. También se realiza una interpretación débil de las restricciones. Se han descrito anteriormente la mayoría de funciones y componentes de esta búsqueda. Sin embargo, se explicarán algunos componentes más y se describirá las modificaciones realizadas a las funciones así como un esquema general de la implementación.

Uno de los aspectos más importante la búsqueda local es el operador de vecino. Para aumentar la eficiencia en tiempo de nuestro algoritmo crearemos un vecindario virtual compuesto de parejas *índice, valor*. Es decir, cada una de estas parejas representará una solución (factible o no) resultado de asignar a la instancia x_indice el *cluster* con índice *valor*. Este vecindario virtual tendrá $n \times k$ componentes. Cabe destacar que al menos n componentes no tendrán utilidad, ya que asignan a una instancia el *cluster* que ya tiene asignado. Además se puede dar el caso de que una asignación deje un *cluster* vacío. Está solución no es factible y por lo tanto será descartada.
Ya que estas comprobaciones son de carácter trivial, se prefiere comprobar las parejas antes de usarlas que generar este vecindario virtual con solo parejas factibles. Estas parejas serán creadas y barajadas, posteriormente se accederán a ellas a través de la función *GetNeighbour*. Esta simplemente devuelve el primer vecino y rota el vector. De esta forma, el vecino devuelto puede ser devuelto de nuevo si y solo si, se han devuelto todos los otros vecinos una vez.

Algorithm 11: Get Neighbour

```
1 GetNeighbour (N)
      input  : Vector de Parejas N
      output: Vector de Parejas N, Pareja n
2     n ← N_0
3     N ← RotarVector(N)
4     return N, n
```

Debido a las optimizaciones realizadas en el código, es difícil marcar separaciones exactas sobre los módulos. Se ha realizado una factorización de las variables para ahorrar la máxima cantidad de operaciones posibles y evitar cualquier operación redundante. Se intentará dividir el pseudocódigo en lo posible y se explicará de forma gradual. Por aumentar la claridad de la explicación se han sacado pequeños módulos de código de la función principal. Estos se verán identificados por ausencia de *input* o *output*. En la implementación, se encontrarían dentro del código de la función principal.

Primero, se comenzará con la preparación de todas las variables necesarias antes del bucle principal. Se comienza creando el vecindario virtual de $n \times k$ elementos como explicado anteriormente y se baraja para aleatorizar los accesos. Se crea una solución aleatoria asignando un *cluster* aleatorio a cada instancia, y se comprueba que sea válida (ningún *cluster* vacío). Usamos entonces la función $CalculateDistanceSeparate$. Se explicó anteriormente en los algoritmos 5 y 6 cómo calcular las distancias a centroides y la desviación general. Se mencionó que las funciones se podían combinar en una sola. Lo cual devolvería X y *average*. Sin embargo, podemos no realizar el último paso y devolver *sum* y *count*. Siendo *sum* un vector con la suma de distancias por cada *cluster*. Y *count*, un simple conteo

de cuantas instancias pertenecen a cada *cluster*.
Siguiendo con la función, creamos una variable para contar el número de iteraciones y calculamos *infeasibility*. Se llama a una función para calcular el valor de la función objetivo con la fórmula ya explicada. Se crea la variable *repeated*, la cuál se utiliza como criterio de parada. Cambiará de valor solo cuando hayamos explorado todo el vecindario virtual y no se haya realizado ningún cambio. Por último, se crea la función $SumInstances$ la cual nos servirá para factorizar el cálculo de los centroides. Simplemente devuelve un vector de tamaño k con la suma de las instancias asignadas a cada *cluster*

Algorithm 12: Preparación Búsqueda Local

```
1  PrepareLocalSearch
       input :
       output:
2      N ← GenerateVirtualNeighbourhood(X)
3      N ← Shuffle(N)
4      X ← GenerateRandomSolution(X)
5      if SolutionNotValid(X) then
6          Exit()
7      M ← CalcularCentroides(X)
8      X, SumDist, ClusterCount ← CalculateDistanceSeparate(X, M)
9      nIterations ← 0
10     TotaInfeasibility ← CalcularInfeasibility(X)
11     ObjectiveValue =
         CalcularObjectiveValue(SumDist, ClusterCount, Lambda, TotalInfeasibility)
12     repeated ← False
13     SumValuesCluster ← SumInstances(X)
```

Se describirá entonces otro pequeño módulo sobre como factorizar el cálculo de *infeasibility* y el conteo de los *cluster*. Este módulo cambiará la asignación de una instancia $x \in X$ desde el *cluster* $OldCluster$ a $NewCluster$. Gracias al uso de la función $CalcularInfeasibilityRow$ se puede calcular la contribución en *infeasibility* de una instancia concreta. Se sustrae del valor de *infeasibility* total. Y se actualiza el conteo en $ClusterCount$ para reflejar que una instancia menos pertenece a ese *cluster*. Se cambia la asignación de x al nuevo *cluster* y se vuelven

a actualizar las variables de la misma forma.

Algorithm 13: Actualizar Count y Infeasibility

```
1 UpdateCountInfeasibility
    input :
    output:
2   TotalInfeasibility ←
      TotalInfeasibility − CalcularInfeasibilityRow(X, Index)
3   ClusterCount_OldCluster ← ClusterCount_OldCluster − 1
4   ChangeCluster(X, Index, NewCluster)
5   TotalInfeasibility ←
      TotalInfeasibility + CalcularInfeasibilityRow(X, Index)
6   ClusterCount_NewCluster ← ClusterCount_NewCluster + 1
```

Antes de describir el código principal se verán las nuevas variables creadas y como se usan para factorizar los parámetros. La factorización de *centroides* se ha conseguido gracias a las dos variables $SumValuesCluster$ y $ClusterCount$. Como se vio en el algoritmo 1, calculamos los centroides hallando los vectores promedios. Si se para un paso antes las variables *sum* y *count* pueden mantenerse separadas y ser actualizadas de forma simple. Al añadir/quitar una instancia de un determinado *cluster*, solo se necesita sumar/restar respectivamente de los valores acumulados.
La factorización de la desviación general se ha conseguido gracias a las variables $SumDist$ y $ClusterCount$. Como se vio en el algoritmo 6, la desviación general puede hallarse como la media de las desviaciones *intra-cluster*. A su vez, las desviaciones *intra-cluster* pueden hallarse como la media de las distancias en un *cluster*. Se puede mantener acumuladas la suma de las distancias *intra-cluster* en $SumDist$ y actualizar de manera individual los valores.

Con la mayoría de los componentes descritos, se comienza a describir el algoritmo final. Se muestra primero el pseudocódigo y luego se comenzará explicar por

partes:

Algorithm 14: Búsqueda Local Débil

```
1  LocalSearchSoft (X)
       input  : Matriz de datos X, Matriz de restricciones R
       output: Matriz de datos X con asignaciones a clusters
2      PrepareLocalSearch(···)
3      while nIterations < 100000 & & !repeated do
4          FirstIteration ← True
5          FirstNeighbour ← N_0
6          OldObjectiveValue ← ObjectiveValue
7          OldInfeasibility ← TotalInfeasibility
8          while OldObjectiveValue ≤ ObjectiveValue & & nIterations <
            100000 do
9              nIterations ← nIterations + 1
10             N, Neighbour ← GetNeighbour(N)
11             if Neighbour == FirstNeighbour & & !FirstIteration then
12                 repeated ← True
13                 Break
14             FirstIteration ← False
15             OldCluster ← GetCluster(X, Neighbour)
16             Index ← Neighbour_Index
17             NewCluster ← Neighbour_Value
18             if ClusterCount_OldCluster == 1 || OldCluster == NewCluster
                then
19                 Continue
20             M, SumValuesCluster ← CalculateCentroidesOptimizado(···)
21             X, SumDist, OldDist ← CalcularDistanciasOptimizado(···)
22             ObjectiveValue = CalcularObjectiveValue(···)
23             if OldObjectiveValue ≤ ObjectiveValue then
24                 RestoreValues(OldDist, ···)
25     return X
```

Se comienza realizando todas las preparaciones necesarias de variables explicadas anteriormente. Se entra al bucle principal de la función. Este bucle *while* continuará mientras se hayan realizado menos de 100.000 iteraciones o no se hayan explorado todos los vecinos de una solución. Las variables $FirstIteration$ y $FirstNeighbour$ son variables auxiliares usadas en la condición de parada. Como se ha dicho, ayudarán a parar el algoritmo si se ha explorado todo el vecindario. En las siguientes lineas se guardan los valores de *infeasibility* y la función objetivo para poder restaurarlos luego. Se entra entonces en el segundo bucle de la función.

Este bucle se utiliza para explorar todos los vecinos de la solución actual hasta que encontrar una mejor solución. Este bucle se parará al encontrar una mejor solución a la actual o al pasar las 100.000 iteraciones. Al principio se aumentan el número de iteraciones. Entonces se obtiene el primer vecino de la lista a partir de la función $GetNeighbour$. Se realiza una comprobación para ver que este vecino no ha sido usado anteriormente. Si lo ha sido, significa que se han probado todos los otros posibles vecinos y por lo tanto el programa termina. Se usan $OldCluster$, $NewCluster$ e $Index$ como variables auxiliares para acceder cómodamente a los valores del vecino y el mantener la antigua asignación de *cluster*. Se realiza una última comprobación para asegurar que el vecino obtenido será una solución válida. Es decir, no deja ningún *cluster* vacío y no nos devolverá la misma solución. Si lo hace, salta al siguiente vecino.

Las funciones $CalcularCentroidesOptimizado$ y $CalcularDistanciasOptimizado$ utilizan lo explicado anteriormente para calcular los centroides y la desviación general de la manera más eficiente posible. Con todos los parámetros ya calculados se puede llamar a la función $CalcularObjectiveValue$ y actualizar el valor de la función objetivo. Se realiza entonces una comparación con el valor de la función objetivo anterior. Si la solución es mejor, todos los valores han sido ya actualizados y se saldrá del bucle con la nueva solución. Si la solución es peor, se restauran las diferentes variables a como estaban al principio del bucle. Es decir, se restauran los valores de:

- El *cluster* asignado a la instancia $x_{Cluster}$
- El valor de *infeasibility*
- El valor de $ClusterCount$
- El valor de $SumDist$ gracias a $OldDist$
- El valor de los centroides M y la suma de valores $SumValuesCluster$

Con esto se termina de describir el algoritmo de búsqueda local. Este algoritmo ciclará hasta haber completado todas las iteraciones o haber explorado todos los vecinos locales de la solución actual. Se trata de un algoritmo fácil de entender pero algo más complicado de implementar. Los ajustes graduales para la factorización de los parámetros pueden ser objeto de confusiones y errores. Sin embargo, es posible alcanzar una solución buena en un tiempo corto gracias a la factorización.

3. Desarrollo de la práctica

La práctica se ha implementado en **Python3** y ha sido probada en la versión 3.6.9. Por tanto, se recomienda encarecidamente utilizar un intérprete de Python3. Se han utilizado diferentes paquetes con funciones de utilidad general: el módulo **time** para la medición de tiempos, el módulo **random** para generar números pseudoaleatorios, el módulo **scipy** para calcular λ, el módulo **math** para el cálculo de distancias euclidianas y el módulo **numpy** para gestionar matrices de forma eficiente. Se utilizó el módulo **pandas** pero finalmente se opta por dejarlo a parte. Se describirán los motivos en el apartado de optimización. Adicionalmente, se utiliza el módulo **matplotlib** para generar algunos de los gráficos.

En cuanto a la implementación, se ha creado un solo programa **main.py** el cual contiene todo el código para la búsqueda local y *greedy*. Se realizan todos las operaciones comunes y no relevantes (carga de datos, restricciones, semillas para números aleatorios) fuera del código en el cual se realizan medidas temporales. Se ha creado otro programa **comparison.py** para realizar las comparaciones. Este programa lanzará *main.py* múltiples veces y escribirá las medidas en un fichero '.csv'. De forma adicional, se crearon dos pequeños programas *extra_analysis.py* y *lambda_mod.py*, los cuales son modificaciones de *main.py* para realizar alguno de los experimentos.

4. Manual de usuario

Para poder ejecutar el programa, se necesita un intérprete de **Python3**, como se ha mencionado. Además, para instalar los módulos se necesita el gestor de paquetes **pip** (o **pip3**).

Para ejecutar el programa basta con ejecutar el siguiente comando:

```
$ python3 main.py
```

El programa se puede lanzar con o sin argumentos. Al lanzarse sin argumentos se realiza una ejecución por defecto. Para especificar una ejecución se lanza con los siguientes 5 argumentos:

```
$ python3 main.py [mode] [dataset]
                  [Restr.%] [Seed] [LambdaMod]
```

- **mode**: algoritmo de búsqueda ∈ { **local**, **greedy**}
- **dataset** ∈ { **ecoli**, **iris**, **rand**}

- **Restr. %**: nivel de restricción $\in \{\mathbf{10}, \mathbf{20}\}$
- **Seed**: semilla $\in \mathbb{Z}$
- **LambdaMod**: modificador de $\lambda \in \mathbb{Q}$ (default = 1)

Un ejemplo en una ejecución normal sería:

```
$ python3 main.py greedy ecoli 10 123 1
```

Por otra parte, se puede comentar brevemente **comparison.py**. Será utilizado para iterar ejecuciones de **main.py** y escribirlas en archivos. Se ha usado tanto para la obtención de valores de las tablas, como para los otros análisis y experimentos realizados. Se pueden modificar la lista de valores con los que se quiere iterar al principio del programa de una manera sencilla y rápida.

5. Experimentación y análisis de resultados

5.1. Descripción de los casos del problema

Para analizar el rendimiento de los algoritmos, se han realizado pruebas sobre 3 conjuntos de datos:

- **Iris**: Conjunto de datos clásico en la ciencia de datos. Contiene información sobre las características de tres tipos de flor Iris. Debe ser clasificado en 3 clases.
- **Ecoli**: Conjunto de datos que contiene medidas sobre ciertas características de diferentes tipos de células que pueden ser empleadas para predecir la localización de cierta proteínas. Debe ser clasificado en 8 clases.
- **Rand**: Conjunto de datos artificial generado en base a distribuciones normales. Debe ser clasificado en 3 clases.

A cada conjunto de datos le corresponde 2 conjuntos de restricciones, correspondientes al 10 % y 20 % del total de restricciones posibles. Con esto, el total de casos de estudio principal es 6.

5.2. Optimización

Desde la primera versión del programa hasta la actual, han habido multitud de cambios en las librerías, métodos, cálculos y estructuras de datos. Se procederá a dar un breve resumen de estas y los cambios que sufrieron. Todas las versiones han sido mantenidas aunque no se incluirán con la versión actual para no confundir al lector.

Como se menciono anteriormente, se utilizó el módulo **pandas** el cual se utiliza en la creación de *dataframes* para la manipulación de datos y análisis. En estos *dataframes* se pueden realizar operaciones incluyendo filas, columnas o cualquier otro tipo de combinación con gran facilidad. Además calcula estadísticas sobre los datos automáticamente, lo cual puede utilizarse para el calculo de variables como centroides. Sin embargo, al ser un *dataframe* muy potente, lleva consigo una gran cantidad de carga computacional y de memoria. Cada vez que se realizasen cambios en este, actualizaba estadísticos de forma automática. Teniendo en cuenta la estructura del programa, esto lo ralentizaba de forma enorme. Sin embargo, sirvió para tener una primera solución. El tiempo medio para ejecutar *Ecoli* en modo *greedy* era de 2 minutos, en modo búsqueda local era de 15 minutos.

Al terminar esta primera fase, se comenzó a contemplar la posibilidad de optimizar el código. Existían dos opciones, factorizar las funciones, lo cuál no era muy lógico ya que *pandas* volvía a calcular los estadísticos automáticamente. O cambiar completamente de *dataframe* a uno mas "rudimentario". Se optó por lo último. Se cambió entonces el *dataframe* de *pandas* por una matriz de datos de *numpy*. *Numpy* se caracteriza por ser uno de los módulos más famosos en Python por su soporte de arrays y matrices, y su larga colección de funciones matemáticas. Esta nueva versión entonces cambió la estructura de datos hacia matrices y dejó intacta la estructura de las funciones. Se produjo un gran cambio en los tiempo: el tiempo medio para ejecutar *Ecoli* en modo *greedy* pasó a 20 segundos, en modo búsqueda local a 2.5 minutos.

Se empieza a plantearse que variables factorizar. Existen dos posibilidades claras, factorizar el calculo de las distancias o de los centroides. Estas modificaciones solo afectarán al algoritmo de búsqueda local. Se mide cuanto tiempo pasa en cada parte del programa y se ve un que el calculo de distancias toma mayor tiempo. En este proceso de factorización, se encuentra una forma más eficiente de calcular distancias euclidianas que la actual. Utilizando el módulo *math* y creando iteradores paralelos de columnas. Esto último se aplica también al algoritmo *greedy*. La mejora de tiempo es menor que la última vez pero aun así significante: el tiempo medio para ejecutar *Ecoli* en modo *greedy* pasó a 10-15 segundos, en modo búsqueda local a 1 minuto.

Se lleva a cabo entonces la segunda factorización para seguir reduciendo el tiempo. De la misma forma, la factorización del calculo de los centroides se realizó solo para el algoritmo de búsqueda local. También se encontró una nueva forma de mejorar ligeramente la suma para el cálculo de centroides. Se aprecia una mejora en la búsqueda local sobre todo: el tiempo medio para ejecutar *Ecoli* en modo *greedy* pasó a 8-10 segundos, en modo búsqueda local a 10 segundos.

Como se ve, ambos algoritmos tardaban lo mismo en *Ecoli* (en los otros no). Se volvieron a realizar medidas de tiempo en las diferentes partes del programa. Los resultados nos decían que una gran parte del tiempo se pasaba en *CalcularInfeasibilityRow*. Se investigó como mejorar la eficiencia de está función y se encontró una manera. Gracias a la eficiencia de listas de *numpy*, se puede 'iterar' entre las restricciones no nulas (distintas de 0). De esta forma, se llega a que el tiempo en modo *greedy* pasa a 2 segundos, en modo búsqueda local a 4 segundos.

Finalmente, se termino de pulir el programa. Quitando variables innecesarias, mejorando generación de posibles soluciones y realizando pequeños cambios para mejorar ligeramente los cálculos gracias de nuevo a la optimización de listas. Con esto se llega a nuestra solucion actual: 1 segundo en modo *greedy* y 2.5 segundos en modo búsqueda local.

5.3. Análisis de los resultados

Se comienza observando los resultados generales de los casos de estudio. Como se ha descrito, se han realizado un total de 6 casos de estudio distintos. Debido a la variabilidad de resultados dependiendo del punto inicial, cada caso de estudio se ejecutará con 5 semillas distintas. Estas nos aseguran una solución inicial distinta y acceso aleatorio diferente en cada ejecución. Además, como medida de precaución, cada una de estas 30 ejecuciones diferentes se han ejecutado 10 veces. Se ha hecho para tomar una media del tiempo y rebajar cantidad de variables externas que pudiesen afectar a este (ocupación de CPU, datos en memoria, etc).

5.3.1. Análisis del algoritmo *greedy*

Se empieza por los resultados del algoritmo de comparación. Por cada *dataset* se incluye la desviación general, *infeasibility* y tiempo de ejecución (en segundos):

	Iris			Ecoli			Rand		
	Tasa_C	Tasa_inf	T	Tasa_C	Tasa_inf	T	Tasa_C	Tasa_inf	T
Seed 123	0.6693	0.0000	0.0999	31.6957	0.0000	1.0325	0.7573	0.0000	0.0981
Seed 456	0.6693	0.0000	0.0996	34.2347	4.0000	0.9138	0.7573	0.0000	0.0834
Seed 789	0.6693	0.0000	0.1135	29.9626	0.0000	1.0134	0.7573	0.0000	0.0828
Seed 101112	0.6693	0.0000	0.0996	33.0444	0.0000	0.8955	0.7573	0.0000	0.0985
Seed 131415	0.6693	0.0000	0.0840	35.1608	6.0000	1.2642	0.7573	0.0000	0.0822
Media	0.6693	0.0000	0.0993	32.8196	2.0000	1.0239	0.7573	0.0000	0.0890

Cuadro 1: Resultados logrados por el algoritmo greedy (10 % de restricciones)

	Iris			Ecoli			Rand		
	Tasa_C	Tasa_inf	T	Tasa_C	Tasa_inf	T	Tasa_C	Tasa_inf	T
Seed 123	0.6693	0.0000	0.1176	28.3973	0.0000	1.5115	0.7573	0.0000	0.1154
Seed 456	0.6693	0.0000	0.1154	32.2893	0.0000	1.2838	0.7573	0.0000	0.0944
Seed 789	0.6693	0.0000	0.1159	33.4576	0.0000	1.4874	0.7573	0.0000	0.1153
Seed 101112	0.6693	0.0000	0.1161	28.3973	0.0000	1.2713	0.7573	0.0000	0.1358
Seed 131415	0.6693	0.0000	0.1371	29.6081	0.0000	1.4887	0.7573	0.0000	0.1164
Media	0.6693	0.0000	0.1204	30.4299	0.0000	1.4086	0.7573	0.0000	0.1155

Cuadro 2: Resultados logrados por el algoritmo greedy (20 % de restricciones)

Como era de esperar, el aumento del número de restricciones incrementa el tiempo de cálculo. Se puede ver que los *datasets Iris* y *Rand* alcanzan lo que se podría asumir como su solución óptima. También se puede observar como el incremento de tiempo no es lineal. El *dataset Ecoli* incrementa su tiempo de ejecución en casi un 40 %. Esto se puede deber al aumento de instancias produce un aumento exponencial en el número de restricciones (lo que implica una mayor cantidad de

cálculos e iteraciones). Este aumento de restricciones puede ser la causa de la diferencia de resultados entre los diferentes niveles de restricciones. En este caso, las restricciones están compuestas por aquellas que reducen la desviación general. Por ello, y dado que el algoritmo *greedy* solo busca reducir *infeasibility*, estas restricciones han funcionado en favor del usuario dando una desviación general menor. Sin embargo, este podría no ser el caso. Al no tener en cuenta (en gran parte) la desviación general en la asignación de *clusters*, este algoritmo *greedy* puede llegar a producir resultados no deseables.

5.3.2. Análisis sobre número de restricciones

Se realiza un pequeño experimento sobre para confirmar si el numero de restricciones influye en las soluciones obtenidas en el algoritmo *greedy*. Para esto, ejecutaremos los programas pero eliminaremos una porción de las restricciones totales. Se cree que cuanto menor sea el número de restricciones, peor será la solución final (normalmente). Se ejecutan los programas con un cuarto y la mitad de restricciones al 10 %. También se ejecutará con la mitad de restricciones al 20 %, para comprobar si quitando diferentes restricciones el resultado final cambia:

	Restrc. 10 % / 4		Restrc. 10 % / 2		Restrc. 20 % / 2	
	Tasa_C	Tasa_inf	Tasa_C	Tasa_inf	Tasa_C	Tasa_inf
Seed 123	35.7745	1	36.8303	3	34.53399	29
Seed 456	36.6186	11	33.0776	12	32.1409	2
Seed 789	36.3976	13	33.2074	3	34.5773	4
Seed 101112	35.7974	8	35.0781	4	28.3973	0
Seed 131415	35.9106	9	34.5411	5	25.4889	0
Media	36.09974	8.4	34.5469	5.4	31.027678	7

Cuadro 3: Resultados de comparación de restricciones

Como se ven, los resultados del test son bastante ilustrativos. Se ha muestran los resultados de estas ejecuciones solo respecto al *dataset Ecoli* ya que los otros no cambian. Puede que debido al pequeño número de instancias de estos *datasets*, estos converjan a la solución óptima aunque quitemos restricciones. Se comienza analizando el caso de la mitad de restricciones al 20 %. Es un caso peculiar ya que se han obtenido no uno, sino dos *outliers*. Se puede ver que *infeasibility* en el primer caso es mucho mayor de lo habitual. Pero también, la última solución es la mejor obtenida hasta ahora en cualquier test. Esto podría confirmar las sospechas de la aleatoriedad en las soluciones dependiendo del punto de salida y las restricciones usadas. Estas servirán como guía para una solución muy buena o indeseable. Con respecto a los otros resultados, se pueden ver que eran parecido a lo esperado,

similares a unas restricciones al 10 %.

Observando las otras columnas y el conjunto en general, se puede ver una mejora de las soluciones de izquierda a derecha. Esto podría confirmar la hipótesis de que, las restricciones impuestas son utilizadas como una guía hacia la solución "óptima"(respecto a desviación general) y por tanto, el algoritmo *greedy* las usa para alcanzarla. Podría no darse la misma situación en todos los casos, por ello se debe realizar un estudio y análisis de las restricciones dándole su debida importancia.

5.3.3. Análisis del algoritmo de búsqueda local

Se sigue entonces con los resultados para el algoritmo de búsqueda local. Se continua la misma estructura que los datos anteriores pero añadiendo una columna para el valor de la función objetivo:

	Iris				Ecoli				Rand			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Seed 123	0.67	0.00	0.67	0.19	22.85	55.00	24.37	2.57	0.76	0.00	0.76	0.15
Seed 456	0.67	0.00	0.67	0.14	22.74	43.00	23.93	2.77	0.76	0.00	0.76	0.17
Seed 789	0.67	0.00	0.67	0.18	22.82	40.00	23.92	2.29	0.76	0.00	0.76	0.18
Seed 101112	0.67	0.00	0.67	0.17	23.90	44.00	25.12	3.33	0.76	0.00	0.76	0.15
Seed 131415	0.67	0.00	0.67	0.17	22.82	44.00	24.04	2.45	0.76	0.00	0.76	0.15
Media	0.67	0.00	0.67	0.17	23.02	45.20	24.27	2.68	0.76	0.00	0.76	0.16

Cuadro 4: Resultados logrados por el algoritmo búsqueda local (10 % de restricciones)

	Iris				Ecoli				Rand			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Seed 123	0.67	0.00	0.67	0.16	22.93	139.00	24.83	2.36	0.76	0.00	0.76	0.16
Seed 456	0.67	0.00	0.67	0.15	22.68	112.00	24.21	2.70	0.76	0.00	0.76	0.16
Seed 789	0.67	0.00	0.67	0.23	22.66	124.00	24.35	2.73	0.76	0.00	0.76	0.17
Seed 101112	0.67	0.00	0.67	0.15	22.93	136.00	24.78	2.48	0.76	0.00	0.76	0.16
Seed 131415	0.67	0.00	0.67	0.17	22.89	140.00	24.80	2.36	0.76	0.00	0.76	0.19
Media	0.67	0.00	0.67	0.17	22.82	130.20	24.59	2.53	0.76	0.00	0.76	0.17

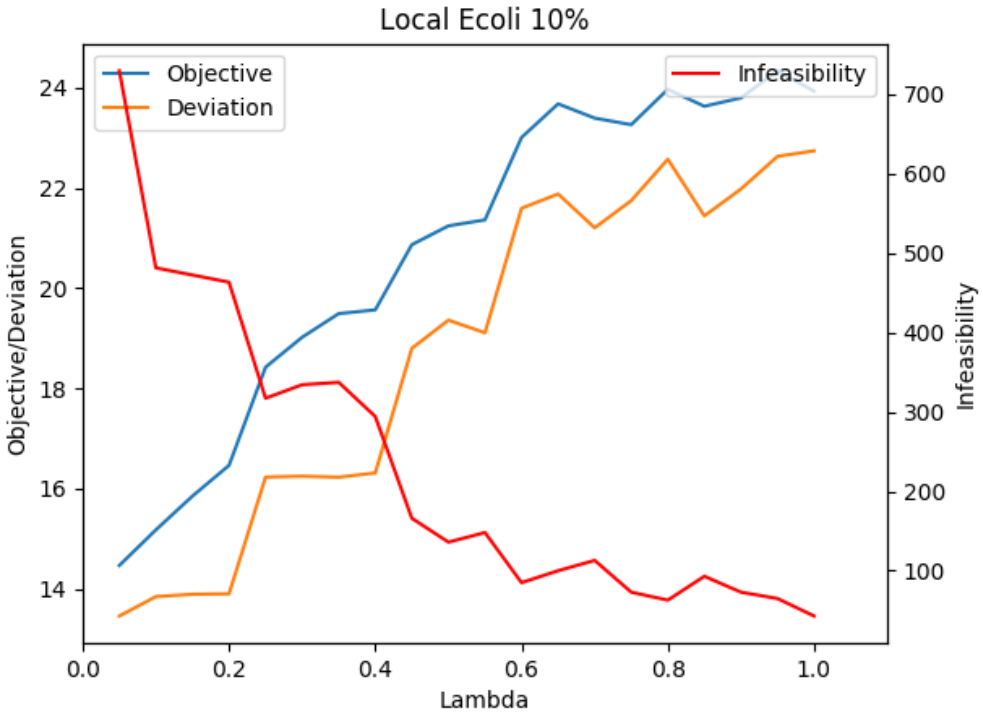
Cuadro 5: Resultados logrados por el algoritmo búsqueda local (20 % de restricciones)

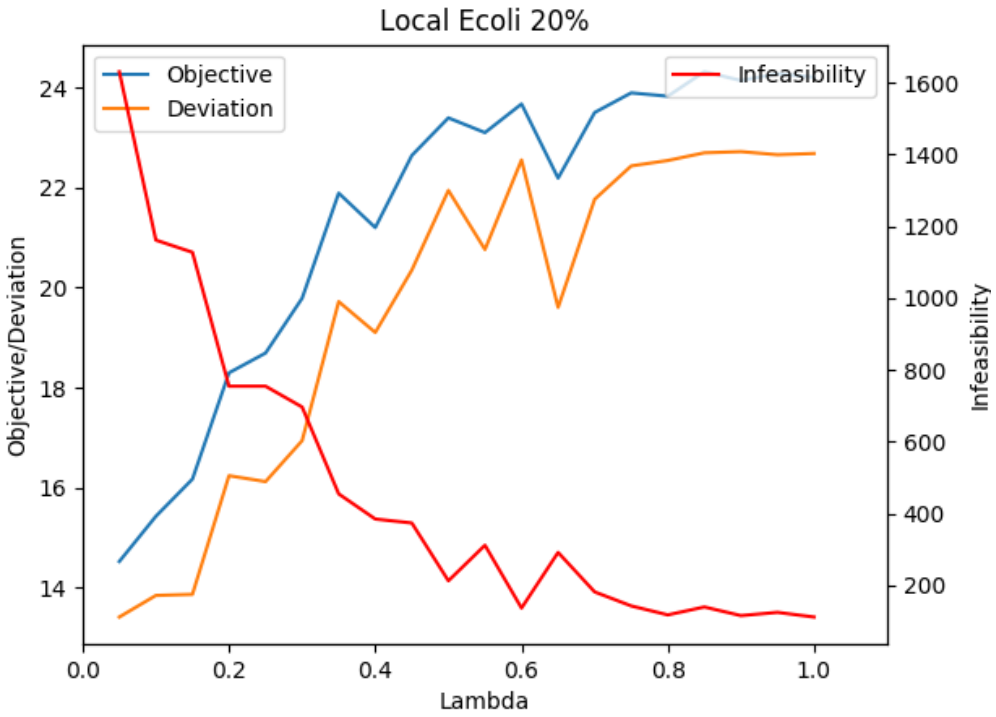
Como se puede apreciar, se reciben datos contradictorios con respecto al algoritmo *greedy*. Primeramente los tiempos entre los dos conjuntos de ejecuciones no parecen cambiar. Segundo, no parece haber demasiada diferencias en los resultados finales. Existen múltiples explicaciones para estos hechos. La opción más probable para explicar la ausencia del aumento de tiempo de ejecución es una cantidad menor de iteraciones. Dado que las condiciones de parada son iguales, esto podría significar que el algoritmo converge de una forma más abrupta cuantas más restricciones se le imponen. Con respecto a la ausencia de variabilidad en la

función objetivo con el cambio de nivel de restricción, se pueden sospechar dos posibles opciones: se ha alcanzado un óptimo local en ambos casos. Sin embargo, comparando los resultados se aprecia un cambio en *infeasibility*, lo cual significa que son soluciones distintas. La otra opción sobre el cálculo de la función objetivo. Como se explicó, esta tiene en cuenta la desviación general y *infeasibility*. Ya que el rango de *infeasibility* cambia entre estos conjuntos, se utiliza λ para dar a cada uno de estos parámetros la misma importancia. De esta forma, se comenzará realizando un estudio sobre esta variable.

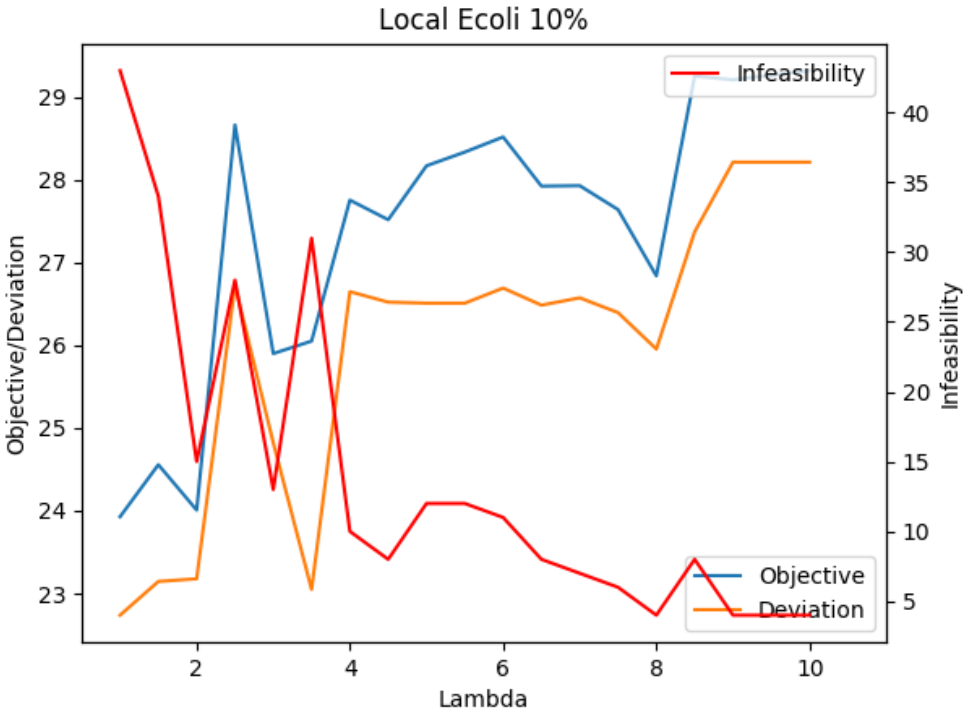
5.3.4. Análisis sobre parámetro λ

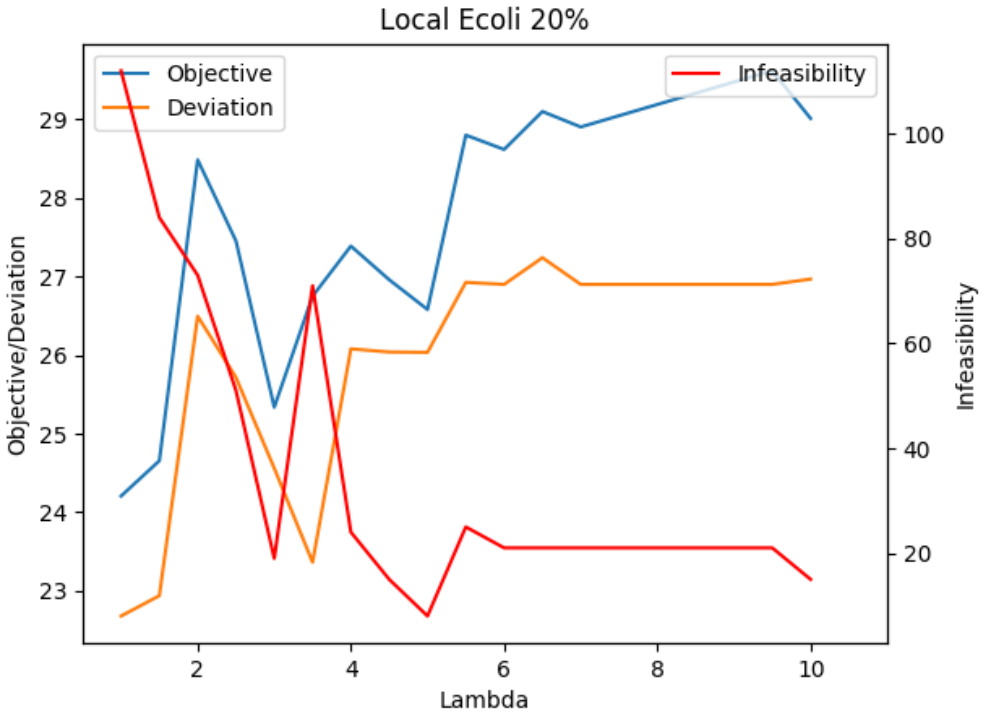
La variable λ se utiliza para crear un equilibro en la función objetivo entre la desviación general y las restricciones violadas. Se optó por una forma sencilla de calcularlo sin embargo, no es la única forma. Existen artículos de investigación muy similares como el descrito por Bise Ryoma et al.[3] en el que simplemente se decide un valor arbitrario para cada *dataset*. Se realiza un experimento en el que se tomará el λ ya calculado como referencia y se irá modificando y analizando los cambios. Primero, se reducirá λ a valores menores y se expresan los resultados en una gráfica. Para ello se utiliza una fracción o % de *lambda*. En este caso se comienza con un 10 % y se realizan las ejecuciones en incrementos de 5 %. Se realiza para el *Ecoli* por ser el más descriptivo y con ambos niveles de restricción:


Local Ecoli 10%
Objective
Deviation
Infeasibility
Objective/Deviation
Infeasibility
Lambda
24
22
20
18
16
14
700
600
500
400
300
200
100
0.0
0.2
0.4
0.6
0.8
1.0


Local Ecoli 20%
Objective
Deviation
Infeasibility
Objective/Deviation
Infeasibility
Lambda
24
22
20
18
16
14
1600
1400
1200
1000
800
600
400
200
0.0
0.2
0.4
0.6
0.8
1.0

Los dirección general de los resultados era la esperada. Conforme menor sea el valor de λ, menos se tiene en cuenta el número de restricciones violadas y mayor es el énfasis en mejorar la desviación general. También se ha realizado el mismo experimento pero con mayores valores de λ. En este caso, se ha ido incrementando en intervalos de 5 % hasta llegar a 10 veces su valor, es decir 1000 %.



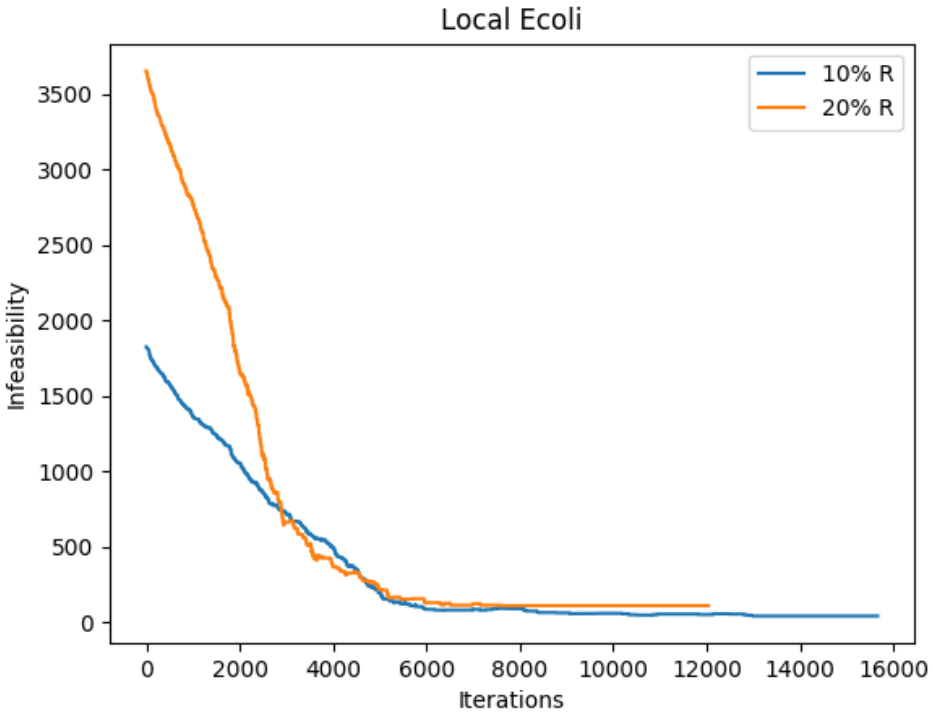


Estas gráficas también sigue la misma tendencia que las anteriores. Si aumentamos el valor de λ, el numero de restricciones violadas se reduce pero el numero la desviación general aumenta. Con estos pequeños estudios se puede concluir la importancia de buscar la mejor función objetivo posible teniendo en cuenta todos sus parámetros. Estas gráficas son muy variables ya que aún persiste el grado de aleatoriedad dependiente de la solución inicial y el orden de exploración. Pero también se puede ver la tendencia general y en que zonas los resultados tienden a ser mejores. Por cada *dataset*, se podría realizar un estudio como este para ajustar los parámetros a un rango que encaje mejor y devuelva mejores soluciones.

Volviendo al origen de este problema (la ausencia de diferencia de soluciones entre los conjuntos con diferentes niveles de restricciones). Se ha visto como ha pesar de tener más restricciones, se intenta compensar cambiando el valor de λ para obtener una función objetivo más "general". Se sospechó ya que existían diferencias en los resultados del algoritmo *greedy*. En este último, ambos conjuntos alcanzar soluciones sin *infeasibility*. Si se quisiese comparar de forma directa, se podría buscar un valor de λ que consiga obtener soluciones con la misma puntuación *infeasibility* en ambos conjuntos.

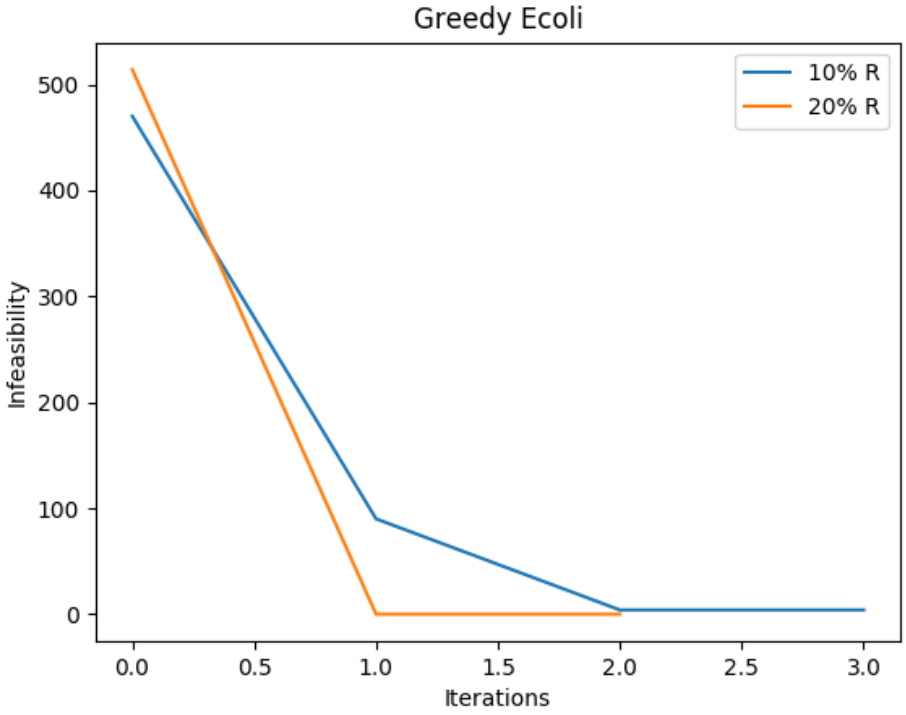
5.3.5. Análisis sobre convergencia

Quedaría resolver la segunda cuestión planteada anteriormente. Se sabe que cuanto más nivel de restricciones, mayor es el número restricciones que se tiene que comprobar. En los resultados del algoritmo *greedy* se aprecia un aumento de alrededor de 20 % en el tiempo de ejecución. Ya que la condición de parada es la misma y se necesitan más cálculos por cada iteración, se tiene la sospecha de haber una reducción en el número de iteraciones. Se puede comprobar con un pequeño experimento donde se plasma la convergencia de *infeasibility* con respecto a las iteraciones realizadas. Se escoge el *dataset Ecoli* por ser el más representativo:



Con este gráfico se confirman las sospechas. El aumento de número de restricciones provoca una convergencia mayor. Esto provoca llegar a un punto común con un nivel de restricción al 10 % de manera relativamente rápida. Tras este punto, ambas partes continúan convergiendo de forma similar. Sin embargo, el tener mayor número de restricciones provoca que el número de vecinos mejores que el actual se reduzca de manera exponencial. Esto resulta en que llegue al óptimo local antes y, por tanto, termine.

De esta forma podemos afirmar que en la búsqueda local de nivel 20, a pesar de realizar más cálculos por iteración que en el nivel 10. Consume el mismo tiempo por el aumento de iteraciones correspondiente. Este mismo análisis se podría realizar con el algoritmo *greedy*:



Como se puede ver, en este caso el mayor número de restricciones también provoca el tener una iteración menos. A diferencia del anterior, este ahorro en iteraciones no compensa completamente el tiempo de ejecución gastado en el aumento de cálculo. Por ello, sigue tomando más tiempo en obtener la solución final con mayor número de restricciones.

5.4. Conclusión

Como conclusión de nuestros experimentos y análisis, se puede decir que por el momento, el mejor algoritmo es el de búsqueda local. El algoritmo *greedy* sería indicado para aquellos casos donde el *dataset* sea pequeño o haya muchas restricciones que puedan guiar al algoritmo. Este es más rápido que la búsqueda local pero ofrece resultados peores para el *dataset Ecoli*, aún siendo este de tamaño "pequeño". Conforme aumente el tamaño de estos, la diferencia entre ambos algoritmos crece, dando el caso de que el algoritmo *greedy* de soluciones demasiado malas para el problema. Sin embargo, se podría aprovechar su gran eficiencia para dar soluciones iniciales a otros algoritmos.

Referencias

[1] TFM: Clustering de documentos con restricciones de tamaño
https://pdfs.semanticscholar.org/812b/edf1c055c37d03b5a2acd655fead801bd215.pdf

[2] Constrained K-means Clustering with Background Knowledge
https://www.cs.cmu.edu/~./dgovinda/pdf/icml-2001.pdf

[3] Efficient Soft-Constrained Clusteringfor Group-Based Labeling
https://www.researchgate.net/publication/336392965_Efficient_Soft-Constrained_Clustering_for_Group-Based_Labeling