PRÁCTICA 1

Metaheurísticas

Técnicas de Búsqueda Local
y algoritmos Greedy

Problema del Agrupamiento con Restricciones

Curso 2019/2020

Paula Iglesias Ahualli

75574877M piglesias@correo.ugr.es

Grupo de prácticas:
MH2 jueves 17:30

1 CONTENIDO

1. DESCRIPCIÓN DEL PROBLEMA

El problema del agrupamiento con restricciones, abreviado **PAR**, es una generalización del agrupamiento clásico, el llamado *clustering*, típico problema en el ámbito de *Machine Learning*.

El objetivo principal es la clasificación de puntos de acuerdo con su similitud sin una clasificación a priori de los datos. Los puntos que pertenezcan al mismo clúster habrán de tener características similares, y los que pertenezcan a grupos contrarios, propiedades muy distintas. Se trata de un método de aprendizaje no supervisado, aplicado en incontables campos [1].

En nuestro caso, se añadirán restricciones de instancia (pertenencia o exclusión a un determinado clúster) lo que hará que la técnica de aprendizaje pase a ser semi-supervisada.

Conocer estos grupos puede ser de gran utilidad para permitir una descripción sintética de un conjunto de datos multidimensional complejo, al reducirse la dimensionalidad del problema.

Algunos campos en los que se puede aplicar el PAR son la robótica, el marketing aplicado, la detección de comunidades terroristas o el diseño de distritos electorales, entre otros.

NUESTRO ENFOQUE AL PROBLEMA

Partiremos de 3 conjuntos de datos: iris, ecoli y rand, cada uno con datos de distinta dimensionalidad y naturaleza, que tendremos que agrupar en 3 clústers, en el caso de iris y rand, y 8 en el caso de ecoli.

Para cada conjunto de datos tendremos dos conjuntos de restricciones, del 10% y del 20%, por lo que contaremos con 6 instancias del PAR a partir de estos conjuntos.

El objetivo del problema es encontrar una partición C de clústeres que minimice la desviación general y cumpla con las restricciones de instancia de los conjuntos de restricciones.

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

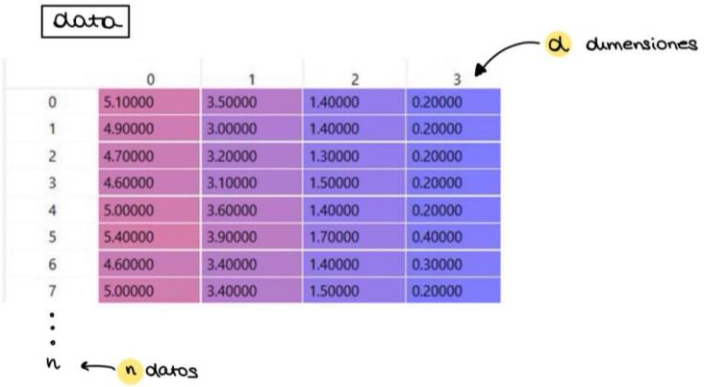
λ será el factor con el que le daremos a infeasibility la relevencia necesaria.

2. DESCRIPCIÓN DE LA APLICACIÓN DE LOS ALGORITMOS

1.1 ESQUEMA DE REPRESENTACIÓN DE LAS SOLUCIONES

Conjunto de datos

El conjunto de datos a agrupar se representa (Figura 1) con una matriz *data* de $n \times d$ valores reales:



	0	1	2	3
0	5.10000	3.50000	1.40000	0.20000
1	4.90000	3.00000	1.40000	0.20000
2	4.70000	3.20000	1.30000	0.20000
3	4.60000	3.10000	1.50000	0.20000
4	5.00000	3.60000	1.40000	0.20000
5	5.40000	3.90000	1.70000	0.40000
6	4.60000	3.40000	1.40000	0.30000
7	5.00000	3.40000	1.50000	0.20000
⋮				
n				

Figura 1: data

Restricciones

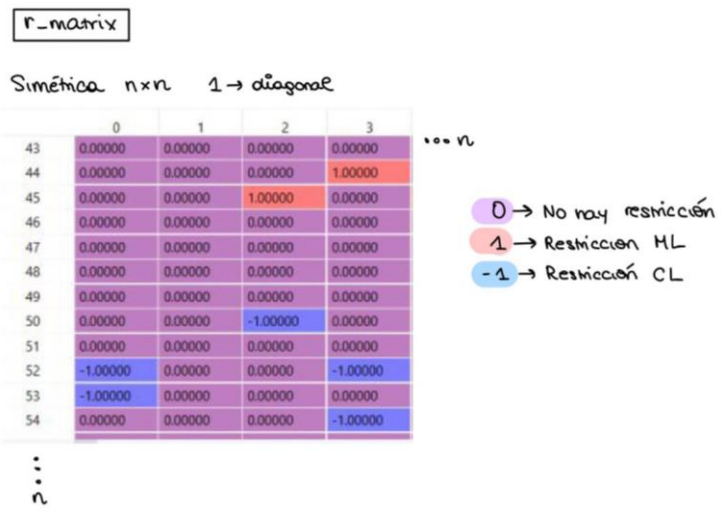
Existen dos tipos de restricciones de instancia (Tabla 1):

ML (Must-Link)	Deben pertenecer al mismo clúster	1
CL (Cannot-Link)	No pueden pertenecer al mismo clúster	-1

Tabla 1 Tipos de restricciones

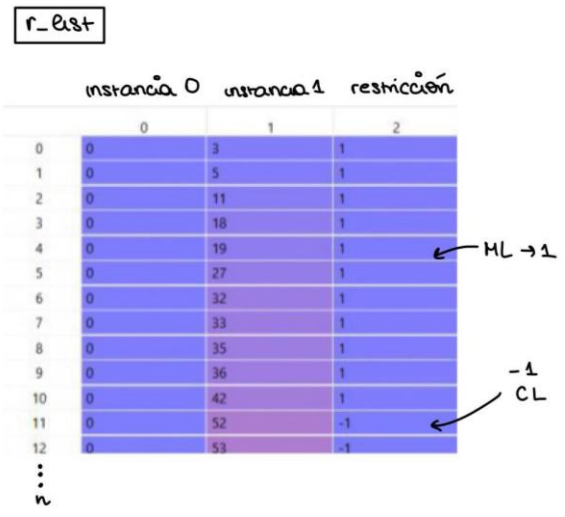
Utilizaremos dos estructuras para almacenar las restricciones:

- Una matriz simétrica de n^2 entradas *r_matrix (*Figura 2*)*
- Una lista de tripletas con R entradas *r_list* [identificador1, identificador2, restricción] (Figura 2)



	0	1	2	3
43	0.00000	0.00000	0.00000	0.00000
44	0.00000	0.00000	0.00000	1.00000
45	0.00000	0.00000	1.00000	0.00000
46	0.00000	0.00000	0.00000	0.00000
47	0.00000	0.00000	0.00000	0.00000
48	0.00000	0.00000	0.00000	0.00000
49	0.00000	0.00000	0.00000	0.00000
50	0.00000	0.00000	-1.00000	0.00000
51	0.00000	0.00000	0.00000	0.00000
52	-1.00000	0.00000	0.00000	-1.00000
53	-1.00000	0.00000	0.00000	0.00000
54	0.00000	0.00000	0.00000	-1.00000
⋮				
n				

Figura 2: r_matrix



	0	1	2
0	0	3	1
1	0	5	1
2	0	11	1
3	0	18	1
4	0	19	1
5	0	27	1
6	0	32	1
7	0	33	1
8	0	35	1
9	0	36	1
10	0	42	1
11	0	52	-1
12	0	53	-1
⋮			
n			

Figura 3: r_list

Solución

Se empleará un vector de n posiciones (Tabla 1). En cada posición se almacena la etiqueta de la instancia asociada a dicha posición (Figura 4).

	S0	S1	S2	S3	S4	…	SN
S	0	0	2	2	1		1

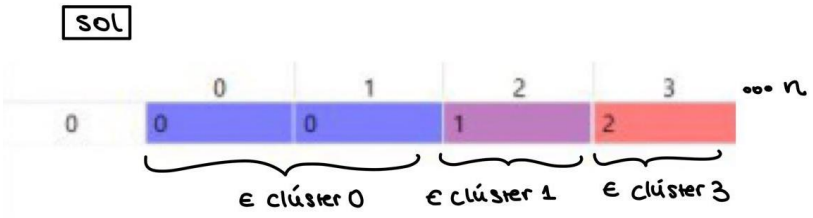


Figura 4: sol

1.2 Función Objetivo

Definimos la función objetivo como:

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

En pseudocódigo se define de la misma forma:

```
obj = c(sol, data, k) + infeasibility_total(sol, r_list) * l
```

Desviación General

La desviación general de la partición C se define como: $\bar{C} = \frac{1}{k}\sum_{c_i \epsilon C} \bar{c_i}$

Su cálculo gracias a *mean* de *numpy*, muy sencillo:

```
general_deviation = np.mean(intra_cluster_mean_distance)
```

Sin embargo, para calcular la distancia media intra-clúster, hay que elaborar un poco más:

En primer lugar, calculamos los centroides utilizando de nuevo la media

```
centroids = compute_centroids(data, sol, k)
```

Separamos los datos que corresponden a cada clúster c

```
data_cluster = np.array([data[np.where(sol == c)] for c in range(k)])
```

Para cada elemento perteneciente al clúster, calculamos la distancia al centroide de ese clúster

```
dis_cluster = [[distance.euclidean(data_cluster[c][i], centroids[c])
                        for i in range(data_cluster[c].shape[0])]
                                    for c in range(k)]
```

Finalmente calculamos la media y obtenemos la distancia media intra-clúster

```
intra_cluster_mean_distance = [np.mean(dis_cluster[c]) for c in range(k)]
```

Infeasibility

Para un cálculo óptimo de la infeasibility, recorremos la lista de restricciones y comprobamos si se cumplen o no:

```
np.count_nonzero([
                    (i[2] == -1 and sol[i[0]] == sol[i[1]]) or
                    (i[2] == 1 and sol[i[0]] != sol[i[1]])
                    for i in r_list
                ])
```

Para cada elemento, aumentamos el contador si se incumple:

- una condición CL → i[2] == -1 y los elementos están en el mismo clústeres
- una condición ML → i[2] == 1 y los elementos están en distintos clústeres

Lambda

Lambda será el factor con el que multiplicaremos la infeasibility, para minimizar un equilibrio entre la desviación y la infeasibility. Se define de la siguiente forma:

$$\lambda = \frac{[D]}{[R]} = \frac{mayor\ distancia\ en\ el\ conjunto\ de\ datos}{Número\ de\ restricciones\ del\ problema}$$

El valor **D** lo obtenemos así:

En primer lugar, obtenemos las combinaciones de todos los elementos, (1,2), (1,3), (2,3), …

```
comb = np.array(list(combinations(index_list, 2)))
```

Seguidamente, calculamos la distancia euclídea de cada par de valores y obtenemos el máximo

```
d = np.max([distance.euclidean(data[i[0]], data[i[1]]) for i in comb])
```

El valor **R** lo obtenemos de la longitud de la lista de restricciones

```
r = len(r_list)
```

Finalmente, $\lambda = \frac{[D]}{[R]}$

3. ESTRUCTURA DEL MÉTODO DE BÚSQUEDA

3.1. BÚSQUEDA LOCAL

Pseudocódigo principal

```
def local_search(data, r_list, k):
    # Calculamos lambda, ya que depende de los datos y no hace falta volver a
calcularla
    l = compute_lambda(data, r_list)

    # La solución inicial serán números aleatorios del número de clústeres para cada
punto inicial
    sol = initial_solution(k, n)

    # Generamos el vecindario virtual (los posibles cambios) de esa solución
    neighbourhood = generate_virtual_neighbourhood(n, k, sol)

    # Calculamos la función objetivo de esta primera solución
    objective_sol = objective(sol, data, k, r_list, l)

    while número de iteraciones <100000 and no se recorra el vecindario:

        # Calculamos el vecino del vecindario i
        neighbour = generate_neighbour(sol, neighbourhood[i])

        # Comprobamos que el vecino sea posible
        if todos los clusteres tienen un elemento:

            # Calculamos la función objetivo del vecino
            objective_neighbour = objective(neighbour, data, k, r_list, l)

            # Si objetivo(vecino) es mejor que objetivo(actual)
            if objective_neighbour < objective_sol:

                # Nuestra solución pasa a ser el vecino
                sol = neighbour

                # Nuestra función objetivo pasa a ser la del vecino
                objective_sol = objective_neighbour

                # Generamos un nuevo vecindario a partir de esta solución
                neighbourhood = generate_virtual_neighbourhood(n, k, sol)

                Recorremos el vecindario desde el principio
    return sol
```

La búsqueda local consiste en el muestreo de las soluciones vecinas mejores que la actual en el entorno de esta.

En nuestro caso utilizamos el método del **primer mejor**: se va generando paso a paso el entorno de la solución actual hasta que se obtiene una solución vecina que mejora a la actual o se construye el entorno completo (no se ha de generar el entorno completo de la solución actual, nos basta la primera que mejore la solución, no ha de ser la mejor del vecindario). Si se encuentra una

solución mejor, ésta sustituye a la actual y se continúa iterando. En caso contrario, finaliza la ejecución del algoritmo.

Para finalizar la ejecución del algoritmo, también hay una cota máxima de `100000` iteraciones, para evitar que el algoritmo dure demasiado.

Método de exploración del entorno

Con el fin de no ocupar mucha memoria, en lugar de generar todos los posibles vecinos a partir de una solución, generamos todos los posibles cambios, un vecindario virtual:

```
neighbourhood = [[i, c] for c in range(k) for i in range(n) if sol[i] != c]
```

[i, c] representará cambiar el clúster del elemento **i** por el nuevo clúster **c**. Es por esto que para que verdaderamente sea un cambio, hay que comprobar que el nuevo clúster sea distinto del anterior (sol[i] `!=` c)

El cambio de vecino consiste por tanto en realizar el cambio de clúster de una posición de la solución:

```
neighbour[i]=to_change[c]
```

4. DESCRIPCIÓN DEL ALGORITMO DE COMPARACIÓN

4.1. GREEDY

Pseudocódigo

```
def greedy(data, r_matrix, k):
    # Generamos los índices
    rsi = generar números desde 0 hasta n
    # Mezclamos los índices para conseguir aleatoriedad
    rsi = shuffle(rsi)
    # Generamos los centroides iniciales
    #   --> números aleatorios para cada dimensión entre el min y max
    centroids = initial_centroids(data, k)
    # La solución inicial tendrá el valor -1 para todos los elementos
    sol = [-1, -1, ... , -1]    # hasta n
    while True:
        old_sol = sol
        # Para cada i de los índices
        for i in rsi:
            # Calculo la infeasibility de añadir ese índice a cada clúster
            infeas = [infeasibility(i, ci, r_matrix, sol) for ci in range(k)]
            # Escojo los clústeres en los que esta sea la menor
            min_inf = min(infeas)
            if hay un único cluster con infeasibility mínima:
                # Este será el mejor clúster
                best_cluster = min_inf
            else:
                # Comparo las distancias
                # Calculo la distancia del elemento al centroide de cada clúster
                distances = [[distance.euclidean(data[i], centroids[c]), c] for c in
min_inf]
                # El mejor cluster será el que tenga la distancia menor de ellos
                best_cluster = min(distances)
            # Asigno el elemento al mejor clúster
            sol[i] = best_cluster
        # Actualizo los centroides con la media de las instancias que lo componen
        centroids = compute_centroids(data, sol, k)
        # Si no ha habido cambios en las soluciones salgo del bucle
        if old_sol == sol:
            break
    return sol
```

La solución Greedy propuesta es una modificación del archiconocido algoritmo **K-medias** de clustering. LA diferencia radica en tener en cuenta las restricciones de instancia.

El algoritmo parte de k centroides iniciales de forma aleatoria (*Aleatoriedad I*) dentro del dominio de los datos. Los índices de las instancias se barajan para recorrer sin repetición (*Aleatoriedad II*).

Al ser un algoritmo Greedy, en cada iteración se escoge la mejor opción: en este caso se tiene en cuenta la menor infeasibility, y en segundo lugar, aquello con centroide más cercano.

Las iteraciones continúan mientras que siga habiendo cambios.

Infeasibility

Pese a ser la misma fórmula que en la función objetivo, en este caso hay que calcular la infeasibility para cada clúster. Al acceder a cada elemento de forma individual, es mucho más eficiente utilizar la matriz de restricciones en lugar de la lista.

La infeasibility de un clúster ci:

```
np.count_nonzero([(ci == sol[i] and r_matrix[xi][i] == -1) or
                  (ci != sol[i] and r_matrix[xi][i] == 1)
                  for i in range(r_matrix.shape[0])])
```

5. PROCEDIMIENTO

Se ha utilizado el lenguaje **Python,** el lenguaje por excelencia del Machine Learning, y el entorno de desarrollo PyCharm.

Para implementar el algoritmo Greedy, ha bastado con el pseudocódigo de las transparencias.

En el caso de la búsqueda local han sido de gran ayuda las transparencias de [2] que explicaban el método del primer mejor de forma muy clara.

5.1. MANUAL DE USUARIO

Al tratarse de ficheros Python, no se necesita compilación y sólo hay que ejecutarlo con un interpréte de Python3.

Los paquetes que se han necesitado son los siguientes:

```
import numpy as np
import random
from scipy.spatial import distance
from itertools import combinations
import copy
```

En /bin se encuentra el fichero `clustering_test.py` cuyo método results() ejecuta 5 ejecuciones de cada dataset y cada % de restricciones del algoritmo especificado. Por defecto se ejecuta `local_search`, para ejecutar `greedy` hay que comentar la línea de local_search y descomentar la de greedy

Las semillas utilizadas se definen en el array seeds al principio de este mismo fichero, y son las siguientes: `seeds = [27, 33, 56, 22, 29]`

6. EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

Resultados del algoritmo Greedy

Resultados obtenidos por el algoritmo Greedy en el PAR con 10% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	3.6542	0.0000	3.6542	0.2682	66.1408	1.0000	66.1676	6.8201	2.3358	0.0000	2.3358	0.3826
Ejecución 2	3.6542	0.0000	3.6542	0.3799	66.7142	9.0000	66.9556	7.8788	2.3358	0.0000	2.3358	0.2510
Ejecución 3	3.6542	0.0000	3.6542	0.3321	66.0475	30.0000	66.8524	6.0619	2.3358	0.0000	2.3358	0.3513
Ejecución 4	3.6542	0.0000	3.6542	0.2523	66.1556	95.0000	68.7045	7.9121	2.3358	0.0000	2.3358	0.2521
Ejecuión 5	3.6542		3.6542	0.2530	64.7035	10.0000	64.9718	5.4432	2.3358	0.0000	2.3358	0.3055
Media	3.6542	0.0000	3.6542	0.2971	65.9523	29.0000	66.7304	6.8232	2.3358	0.0000	2.3358	0.3085

Resultados obtenidos por el algoritmo Greedy en el PAR con 20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	3.6542	0.0000	3.6542	0.2633	66.5108	2.0000	66.5377	5.0161	2.3358	0.0000	2.3358	0.2619
Ejecución 2	3.6542	0.0000	3.6542	0.2820	65.0495	0.0000	65.0495	5.9258	2.3358	0.0000	2.3358	0.1663
Ejecución 3	3.6542	0.0000	3.6542	0.2694	66.1916	41.0000	66.7416	4.3892	2.3358	0.0000	2.3358	0.2480
Ejecución 4	3.6542	0.0000	3.6542	0.2509	0.0000	0.0000	0.0000	5.5248	2.3358	0.0000	2.3358	0.1666
Ejecuión 5	3.6542	0.0000	3.6542	0.2602	66.3108	0.0000	66.3108	5.8984	2.3358	0.0000	2.3358	0.2499
Media	3.6542	0.0000	3.6542	0.2652	52.8125	8.6000	52.9279	5.3509	2.3358	0.0000	2.3358	0.2185

Tabla 2 Resultados globales Greedy

En (Tabla 2) podemos observar que para los conjuntos de datos Iris y Rand, tanto para el 10% como el 20% de restricciones, la infeasibility y la Tasa C se mantienen fijas en 0 y 3.6542. Estos conjuntos de datos son fáciles de agrupar y hacen que el Greedy obtenga muy buenos resultados.

En el caso de Ecoli, un conjunto de datos con mayor número de cluster y de mayor dificultad, los resultados son peores con una agregación muy superior (16 veces mayor).

Ecoli			
Tasa_C	*Tasa_inf*	*Agr.*	T
66.1408	1.0000	66.1676	6.8201
66.7142	9.0000	66.9556	7.8788
66.0475	30.0000	66.8524	6.0619
66.1556	95.0000	68.7045	7.9121
64.7035	10.0000	64.9718	5.4432
65.9523	29.0000	66.7304	6.8232

Tabla 3 Greedy Ecoli 10%

Sin embargo, en (Tabla 3) podemos comprobar como la aleatoriedad del método hace que en algunos casos la infeasibility sea muy dispar: puede ir desde 1.0 hasta 95, valores muy extremos. No afecta tanto en la función objetivo debido al bajo valor de lambda (excepto en casos raros donde ésta también es un *outlier* como en la ejecución 4 con el 20% de restricciones).

Un aspecto positivo del algoritmo es el bajo tiempo de ejecución necesario, inmediato para conjuntos de datos como Iris o Rand.

Resultados del algoritmo Búsqueda Local

Resultados obtenidos por el algoritmo Búsqueda Local en el PAR con 10% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	3.6542	0.0000	3.6542	8.9849	53.1490	42.0000	54.2759	461.9646	2.3358	0.0000	2.3358	11.1788
Ejecución 2	3.6542	0.0000	3.6542	9.4900	51.9759	101.0000	54.6857	350.9205	2.3358	0.0000	2.3358	10.8959
Ejecución 3	3.6542	0.0000	3.6542	8.7549	52.7977	42.0000	53.9245	272.9690	2.3358	0.0000	2.3358	8.3606
Ejecución 4	3.6542	0.0000	3.6542	8.6870	52.8674	48.0000	54.1553	276.0927	2.3358	0.0000	2.3358	9.5904
Ejecuión 5	3.6542	0.0000	3.6542	8.4362	52.5505	82.0000	54.7506	219.5131	2.3358	0.0000	2.3358	6.8458
Media	3.6542	0.0000	3.6542	8.8706	52.6681	63.0000	54.3584	316.2920	2.3358	0.0000	2.3358	9.3743

Resultados obtenidos por el algoritmo Búsqueda en el PAR con 20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Ejecución 1	3.6542	0.0000	3.6542	10.4459	54.8828	127.0000	53.1791	2698.4762	2.3358	0.0000	2.3358	12.3839
Ejecución 2	3.6542	0.0000	3.6542	16.2494	52.8825	126.0000	54.5728	364.4682	2.3358	0.0000	2.3358	8.7845
Ejecución 3	3.6542	0.0000	3.6542	8.6193	49.1395	314.0000	53.3517	373.4984	2.3358	0.0000	2.3358	7.7393
Ejecución 4	3.6542	0.0000	3.6542	11.9052	52.6441	122.0000	54.2807	418.1109	2.3358	0.0000	2.3358	14.0021
Ejecuión 5	3.6542	0.0000	3.6542	13.4214	52.8319	145.0000	54.7770	410.6877	2.3358	0.0000	2.3358	9.9095
Media	3.6542	0.0000	3.6542	12.1282	52.4762	166.8000	54.0323	853.0483	2.3358	0.0000	2.3358	10.5639

Tabla 4 Resultados globales Búsqueda Local

En (Tabla 4), al igual que con Greedy, se llega al infeasibility 0 sin problemas en conjuntos de datos como Iris o Rand. Y el algoritmo es bastante rápido, no pasando de los 10 segundos en la mayoría de las ejecuciones.

En cuanto a Ecoli, su dificultad queda patente en los tiempos de ejecución elevados y en los mayores valores de función objetivo, tasa c e infeasibility. En Ecoli además se puede apreciar especialmente, cómo los tiempos de ejecución difieren al aumentar el % de restricciones. Esto se debe a que el cuello de botella está, naturalmente, en la función objetivo, que debe recorrer la lista completa de restricciones, y al haber más restricciones se traduce en un mayor tiempo de cómputo.

Los datos son muy uniformes, indicativo de mayor fiabilidad del algoritmo y menor dependencia de la aleatoriedad respecto al greedy. No varía mucho entre ejecuciones.

No se habrá pasado por alto el elevado tiempo de ejecución en la primera ejecución de Ecoli con un 20% de restricciones: se trata de un valor inusual ajeno a la semilla de aleatoriedad empleada. Esto último se ha comprobado haciendo varias ejecuciones con la misma semilla, por lo que debe deberse a factores externos del hardware. En consecuencia, para la comparación de las medias no se ha tenido en cuenta, puesto que infla en exceso la media y puede hacer que las conclusiones sean erróneas. Sin embargo, se ha mantenido en la tabla para contrastar la fiabilidad de las medidas, que también implican valores atípicos.

Comparando los algoritmos

10% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Greedy	3.6542	0.0000	3.6542	0.2971	65.9523	29.0000	66.7304	6.8232	2.3358	0.0000	2.3358	0.3085
BL	3.6542	0.0000	3.6542	8.8706	52.6681	63.0000	54.3584	316.2920	2.3358	0.0000	2.3358	9.3743

20% de restricciones

	Iris				Ecoli				Rand			
	Tasa_C	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T	*Tasa_C*	*Tasa_inf*	*Agr.*	T
Greedy	3.6542	0.0000	3.6542	0.2652	52.8125	8.6000	52.9279	5.3509	2.3358	0.0000	2.3358	0.2185
BL	3.6542	0.0000	3.6542	12.1282	52.4762	166.8000	54.0323	391.6913	2.3358	0.0000	2.3358	10.5639

Tabla 5 Comparación global Greedy Búsqueda Local

(Tabla 5) Señalado en verde, se encuentran los valores que no varían entre algoritmos: se tratan de los resultados de los conjuntos Iris y Rand, que, debido a su baja complejidad, ambos algoritmos son capaces de encontrar el valor óptimo, eso sí, siendo el Greedy más rápido. En (Figura 5) podemos verlo de forma más gráfica centrándonos en el 10% de restricciones.

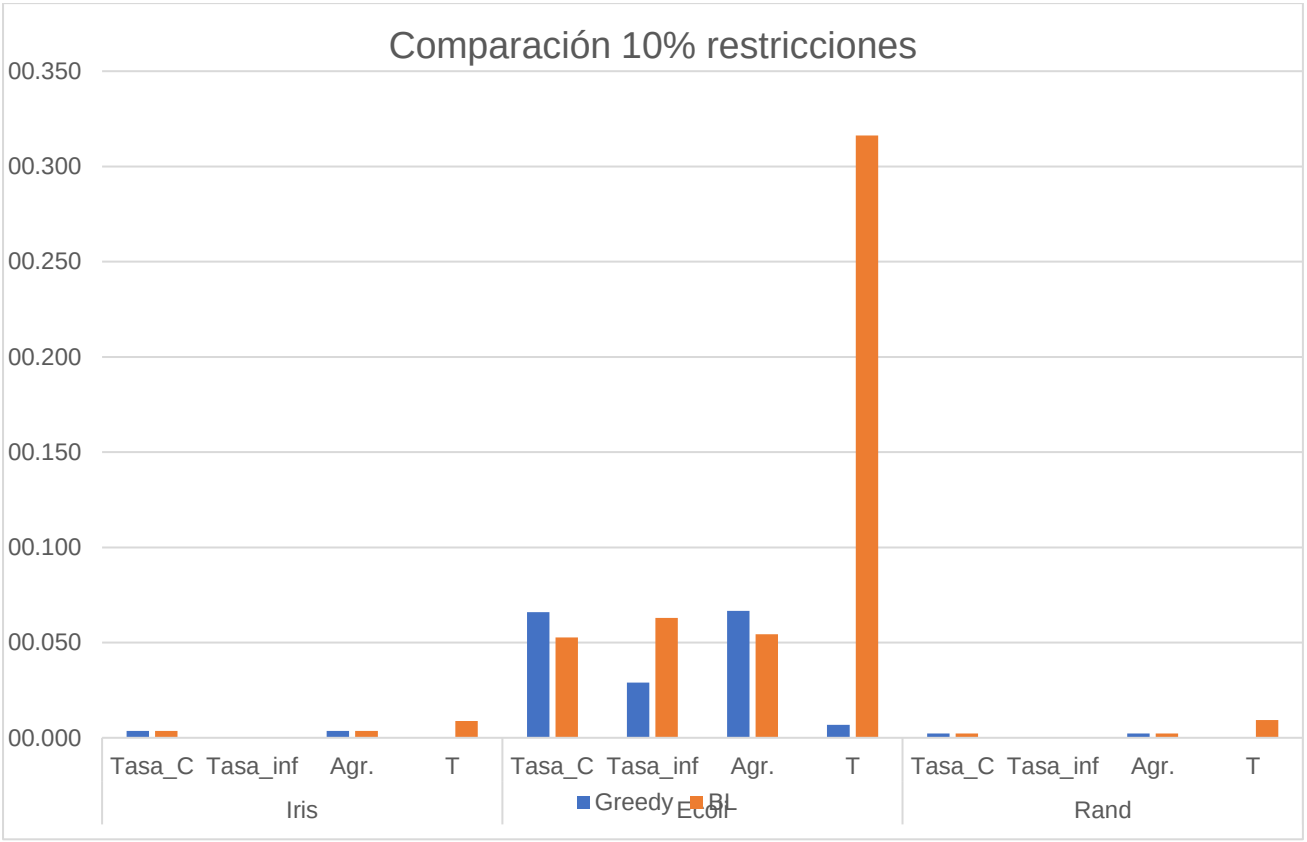


Figura 5 Comparación 10% restricciones

De nuevo, es en **Ecoli** donde podemos apreciar diferencias más interesantes, para ello graficamos el gráfico anterior sólo en este área de interés (Figura 6):

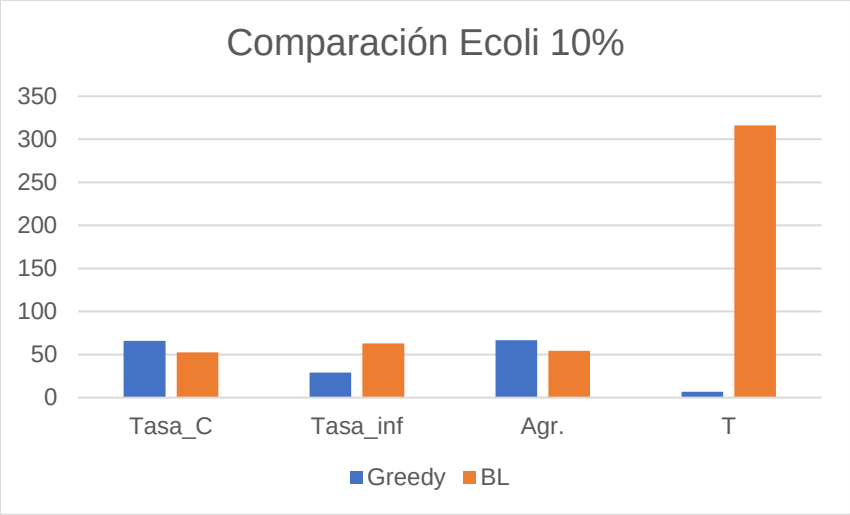


Figura 6 Comparación Ecoli 10%

En primer lugar, llama la atención el tiempo de ejecución muy superior en BL respecto a Greedy. Esto es debido a lo ya comentado: recorrer toda la lista de prioridades es un proceso costoso.

En segundo lugar, pasamos a fijarnos en la tasa de infeasibility: recordamos que en Greedy estos eran valores dispares, pero resultan en conjunto más bajos que en la Búsqueda Local. Esto se debe en gran medida a que lo que intenta minimizar cada algoritmo difiere: en Greedy calculamos la infeasibility para cada clúster y escogemos la menor. Sólo si hay empate comparamos las distancias entre los centroides. La función objetivo, que se emplea en la búsqueda local es más compleja y debe deberse al bajo valor de lambda a que se priorice a la tasa c respecto a la infeasibility. Si de verdad esto fuera lo que queremos minimizar, habría que cambiar el valor de lambda, o directamente centrarnos en la infeasibility.

Pasamos a fijarnos a los dos otros valores de comparación: No por mucho, pero la búsqueda local mejora a Greedy. No es una mejora muy significativa y de hecho si nos fijamos en el siguiente gráfico, del 20% (Figura 7), podemos comprobar que en ambos algoritmos la diferencia es mínima.

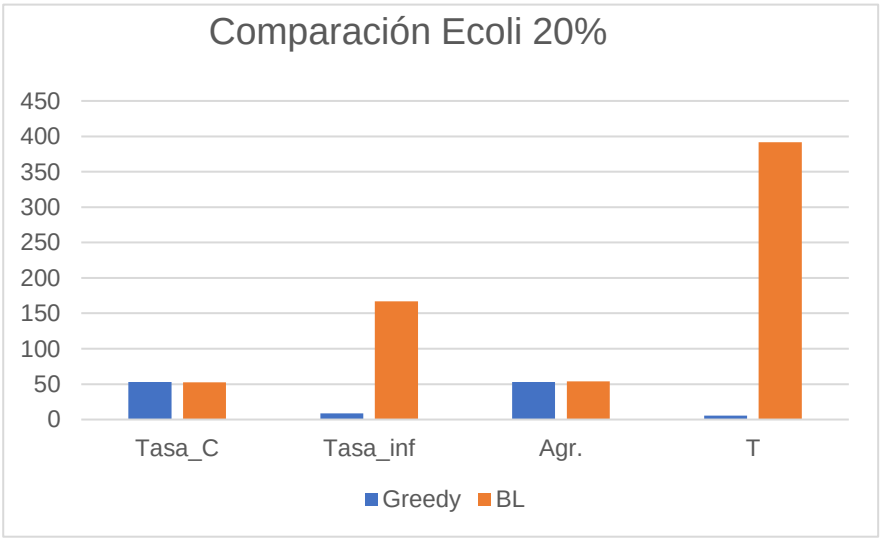


Figura 7 Comparación Ecoli 20%

La conclusión inmediata sería que conviene en término general, emplear el algoritmo Greedy, por su muy significativo tiempo de ejecución y a pesar de este, conseguir resultados similares a un algoritmo más costoso como es la búsqueda local.

Cabe mencionar, sin embargo, los defectos conocidos de los algoritmos k-medias: depende fuertemente de la aleatoriedad inicial, mientras que eso para la búsqueda local era menos relevante, en greedy las horquillas de infeasibility eran extremadamente amplias. Esto es crucial en la robustez del algoritmo, y un factor clave a la hora de no descartar de lleno la búsqueda local: si se consiguiera optimizar el tiempo de ejecución, los resultados de la búsqueda local son más estables, y si se quisiera una mejor infeasibility tan sólo habría que ajustar el factor lambda.

Por último, es curioso comprobar la convergencia de ambos algoritmos. Para ello, en cada iteración se ha calculado la función objetivo. Estas son las gráficas resultantes:

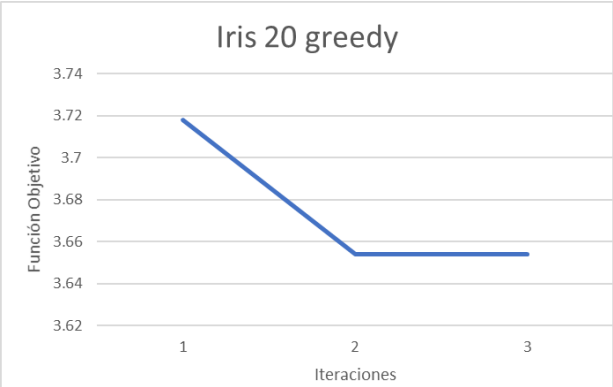


Figura 8: Iris 20% Greedy

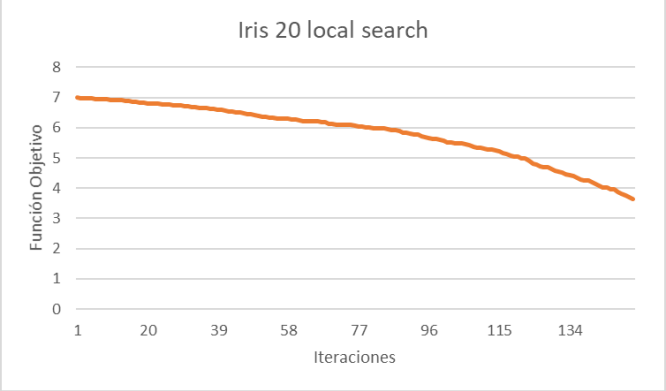


Figura 9: Iris 20% Búsqueda Local

Podemos ver como la diferencia entre el número de iteraciones necesario es muy significativa: Greedy con apenas 3 iteraciones converge, mientras que local search necesita unas 150 más.

Por otra parte, si nos fijamos en la tendencia de las curvas de la búsqueda local, éstas sugieren que todavía hay margen de mejora. En el caso de Iris sabemos que alcanzan el mínimo global, pero en Ecoli la tendencia también parece que podría encontrar mejores resultados. Podemos ver que no alcanza el número máximo de iteraciones, ha de pararse porque no encuentra en todo el vecindario un mejor vecino. Tal vez podría explorarse otra condición de parada alternativa que pueda proporcionar mejores resultados.

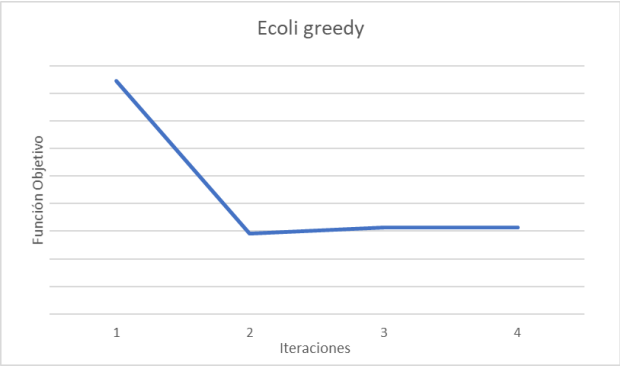


Ilustración 10 Ecoli 20% Greedy

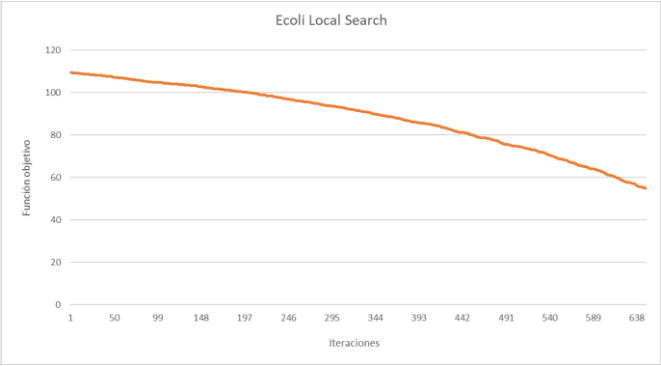


Ilustración 11 Ecoli 20% Búsqueda Local

7. BIBLIOGRAFÍA

[1]

„The 5 Clustering Algorithms Data Scientists Need to Know“, George Seif, Feb 5, 2018, Towards Data Science

https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68

[2]

Soft Computing and Intelligent Information Systems, UGR, Algorítmica Tema 2, 2012-2013

https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/Algoritmica/Tema02-BusquedaLocal-12-13.pdf