Problema de Agrupamiento con Restricciones (PAR)

Técnicas de búsqueda local y algoritmos greedy

Metaheurísticas: Práctica 1.b, Grupo MH2 (Jueves de 17:30h a 19:30h)

Jorge Sánchez González - 75569829V
jorgesg97@correo.ugr.es

13 de marzo de 2020

Índice

1. El problema

1.1. Descripción del problema

El **agrupamiento** o **análisis de clusters** clásico (en inglés, *clustering*) es un problema que persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos. Así, se trata de una técnica de aprendizaje no automático que permite clasificar en grupos (desconocidos a priori) objetos de un conjunto de datos que tienen características similares.

El **Problema del Agrupamiento con Restricciones (PAR)** (en inglés, Constrained Clustering, CC) es una generalización del problema del agrupamiento clásico. Permite incorporar al proceso de agrupamiento un nuevo tipo de información: las restricciones. Al incorporar esta información la tarea de aprendizaje deja de ser no supervisada y se convierte en semi-supervisada.

El problema consiste en, dado un conjunto de datos X con n instancias, encontrar una partición C del mismo que minimice la desviación general y cumpla con las restricciones del conjunto de restricciones R. En nuestro caso concreto, solo consideramos restricciones de instancia (Must-Link o Cannot-Link) y todas ellas las interpretaremos como restricciones débiles (Soft); con lo que la partición C del conjunto de datos X debe minimizar el número de restricciones incumplidas pero puede incumplir algunas. Formalmente, buscamos

$$\text{Minimizar } f = \hat{C} + \lambda * infeasibility$$

donde:

- C es una partición (una solución al problema), que consiste en una asignación de cada instancia a un cluster.
- $\hat{C}$ es la desviación general de la partición C, que se define como la media de las desviaciones intra-cluster.
- $infeasibility$ es el número de restricciones que C incumple.
- λ es un hyperparámetro a ajustar (en función de si le queremos dar más importancia a las restricciones o a la desviación general).

1.2. Conjuntos de datos considerados

Trabajaremos con 6 instancias del PAR generadas a partir de los 3 conjuntos de datos siguientes:

1. Iris: Información sobre las características de tres tipos de flor de Iris. Se trata de 150 instancias con 4 características por cada una de ellas. Tiene 3 clases ($k = 3$).
2. Ecoli: Contiene medidas sobre ciertas características de diferentes tipos de células. Se trata de 336 instancias con 7 características. Tiene 8 clases ($k = 8$).
3. Rand: Está formado por tres agrupamientos bien diferenciados generados en base a distribuciones normales ($k = 3$). Se trata de 150 instancias con 2 características.

Para cada conjunto de datos se trabaja con 2 conjuntos de restricciones, correspondientes al 10 % y 20 % del total de restricciones posibles.

2. Descripción de la aplicación de los algoritmos

En esta sección se describen las consideraciones comunes a los distintos algoritmos. Se incluyen la representación de las soluciones, la función objetivo y los operadores comunes a los distintos algoritmos. Ya que los únicos puntos en común de la búsqueda local y la técnica greedy son la función objetivo y la representación de las soluciones, no estudiaremos ningún operador común. No se han incluido tampoco los detalles específicos de ninguno de los algoritmos en esta sección.

Una primera consideración general es el hecho de que la distancia usada como medida de cercanía (o similitud) entre las distintas instancias y clusters es la distancia euclídea.

El lenguaje utilizado para la implementación de la práctica ha sido **Python**.

2.1. Representación de la soluciones

Las soluciones a nuestro problema serán las llamadas *particiones*, que asignan a cada instancia del conjunto de datos uno de los k clusters. Aunque existen varias formas de representar una partición, por simplicidad se ha decidido utilizar la misma forma para ambos algoritmos (tanto el Greedy como el de búsqueda local). En concreto, respresentamos una partición de un conjunto de n instancias en k clusters con una lista S de tamaño n cuyos valores son enteros $S_i \in \{0, 1, 2, ..., k-1\}$. Así, el valor de la posición i del vector, S_i, indica el cluster al que la i-ésima instancia, x_i, ha sido asignada.

Por verlo con un ejemplo, la representación será de la siguiente forma:

```
partition = [0,0,0,5,0,1,1,1,1,1,2,3,1,1,1,...]
```

Esta partición indicaría, por ejemplo, que la instancia x_0 se ha asignado al cluster número 0, c_0, y que la instancia x_3 se ha asignado al cluster c_5.

Cabe mencionar también que durante la ejecución de los algoritmos, las particiones en proceso de construcción tendrán sus valores en $\{-1, 0, 1, 2, ..., k-1\}$. Cuando una posición i tenga el valor -1, esto indicará que aún no se le ha asignado ningún cluster a la instancia x_i.

2.2. Función objetivo

Como hemos visto en la seccion anterior (1.1) para calcular la función objetivo se necesita saber tanto la desviación general de la partición, $\hat{C}$, como el número de restricciones que esta viola, $infeasibility$.

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

En pseudocódigo:

Algorithm 1: objective_function

Data: Conjunto de datos X, partición en forma de vector S, lista de restricciones *constraints_list*, lista de centroides *centroids*, hiperparámetro *lambda*

Result: function_value

begin

```
dev ← general_deviation(X, S, centroids)
inf ← infeasibility(S, constraints_list)
function_value ← dev + lambda * inf
return function_value
```

end

El hyperparámetro λ que se usará por defecto será el mínimo indicado en las diapositivas del Seminario 2, es decir, el cociente entre la distancia máxima existente en el conjunto de datos y el número de restricciones del problema.

Cabe mencionar que la función objetivo como tal solo será usada por el algoritmo de búsqueda local, ya que el algoritmo Greedy usará la desviación general y la infactibilidad en diferentes etapas (y por separado). De cualquier manera, las soluciones de ambos algoritmos van a ser evaluadas con esta función.

2.2.1. Desviación general de una partición

La desviación general de una partición se define como la media de las distancias medias intra-cluster, esto es,

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

Donde las distancias medias intra-cluster se definen a su vez como

$$\hat{c_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} distance(\vec{x_j}, \vec{\mu_i}).$$

En pseudocódigo:

Algorithm 2: mean_dist_intra_cluster

```
Data: Conjunto de datos X, partición S, cluster del que queremos calcular su distancia
      intra cluster cluster_id, centroide del cluster centroid
Result: distance_intra_cluster
begin
    Y ← instancias_asignadas_al_cluster(X, S, cluster_id)
    distances ← list([ ])
    foreach y ∈ Y do
        distances.append(distance(y, centroid))
    end
    distance_intra_cluster ← media(distances)
    return distance_intra_cluster
end
```

Algorithm 3: general_deviation

```
Data: Conjunto de datos X, partición S, lista de centroides centroids
Result: deviation
begin
    cluster_ids ← {0,1,...,longitud(centroids)-1}
    intra_cluster_distances ← list([ ])
    foreach c_id ∈ cluster_ids do
        d ← mean_dist_intra_cluster(X, S, c_id, centroids[c_id])
        intra_cluster_distances.append(d)
    end
    deviation ← media(intra_cluster_distances)
    return deviation
end
```

2.2.2. Infactibilidad

La infactibilidad de una partición ($infeasibility$) se define como el número de restricciones que incumple. Por ello para calcularla utilizamos una función auxiliar, V, que nos va a decir, dada una partición, y un par de instancias con un valor de restricción (-1,0 o 1), si la partición incumple alguna restricción asociada a las dos instancias. Por otro lado, cabe destacar también que usamos la lista de restricciones, y no la matriz de restricciones, para recorrer todas las restricciones para calcular el $infeasibility$ de una manera más eficiente.

Algorithm 4: V (si se incumple alguna restricción o no)

Data: Índice i de la instancia x_i, índice j de la instancia x_j, partición S, valor de restricción $constraint\_value$)
Result: incumplimiento
begin
  incumplimiento_ML $\leftarrow$ ($constraint\_value == 1$ and $S[i] \neq S[j]$)
  incumplimiento_CL $\leftarrow$ ($constraint\_value == -1$ and $S[i] == S[j]$)
  incumplimiento $\leftarrow$ (incumplimiento_ML or incumplimiento_CL)
  return incumplimiento
end

Algorithm 5: $infeasibility$

Data: Partición S, lista de restricciones $constraints\_list$
Result: infeasibility
begin
  infeasibility $\leftarrow$ 0
  foreach $(i, j, contraint\_value) \in constraints\_list$ **do**
    infeasibility $\leftarrow$ infeasibility + V(i, j, S, constraint_value)
  end
  return infeasibility
end

3. Algoritmos

En esta práctica se han usado dos algoritmos, el Greedy y el de Búsqueda Local. A continuación se describirán mostrando pseudocódigo y comentando los aspectos más importantes.

3.1. Algoritmo Greedy

La solución Greedy para el PAR consiste en modificar el algoritmo k-medias para tener en cuenta las restricciones. Para la solución greedy (al igual que para la de búsqueda local) se hace una interpretación débil de las restricciones. De esta forma, para cada instancia se llevará a cabo la asignación que menos restricciones viole y que más disminuya la desviación general. Las particiones se construirán como se ha comentado anteriormente (en forma de lista S de tamaño $n = numero\_de\_instancias$).

Como se ha discutido en clase, este algoritmo puede llegar a formar ciclos e iterar de forma infinita. Es decir, puede saltar de una partición a otra y luego volver a la misma y así sucesivamente. En concreto, podría formar ciclos de una longitud arbitraria N,

$$(particion_1 \longrightarrow particion_2 \longrightarrow ... \longrightarrow particion_N \longrightarrow particion_1 \longrightarrow ...).$$

En la siguiente implementación se contemplan los ciclos de longitud 1 (debido a que experimentalmente son los que más han surgido y a que su control no constituye una gran desventaja en eficiencia). La descripción en pseudocódigo se ha dividido en dos partes. En primer lugar se describe el algoritmo dados los centroides iniciales. En segundo lugar se describe la forma de generar dichos centroides iniciales.

Algorithm 6: Algoritmo Greedy - COPKM

```
Data: Conjunto de datos X, matriz de restricciones const_matrix, lista de restricciones
      const_list, lista de centroides iniciales initial_centroids
Result: Partición solución S
begin
    k ← longitud(initial_centroids)
    n ← longitud(X)
    RSI ← list([0,1,...,n-1])
    RSI ← RandomShuffle(RSI)
    S ← list(longitud = n, valores = -1)
    S_prev ← list(longitud = n, valores = -1)
    change ← True
    cycle ← False
    while change and not cycle do
        change ← False
        S_new = list(longitud = n, valores = -1)
        1.-Asignamos las instancias a cada cluster:
        foreach i ∈ RSI do
            1.1 Calculamos el incremento en infeasibility que produce la asignación de x_i
             a cada cluster c_j y guardamos los índices j que producen menos incremento.
            incrementos ← incrementos_en_inf(i, S_new, const_matrix)
            less_incr_clusters ← argmin(incrementos)
            1.2-De entre las asignaciones (js) que producen menos incremento en
             infeasibility, seleccionamos la asociada con el centroide μ_j mas cercano a x_i
            distances ← [distance(X[i], centroids[j]) for j in less_incr_clusters]
            closest ← less_incr_clusters[argmin(distances)]
            S_new[i] ← closest
            if S_new[i] ≠ S[i] then
                change ← True
            end
        end
        2.-Comprobamos que no estamos atrapados en un ciclo de longitud 1
        if listas_son_iguales(S_prev, S_new) then
            cycle ← True
        end
        3.-Actualizamos la partición y los centroides de cada cluster
        S_prev ← S
        S ← S_new
        foreach j ∈ {0, 1, 2, ..., k − 1} do
            centroids[j] ← media_de_instancias_en_el_cluster(X, S, cluster_id = j)
        end
    end
    return S
end
```

Algorithm 7: centroids_initialization

```
Data: Conjunto de datos X, número de clusters k
Result: Lista de centroides centroids
begin
    - Dominio de una caracteristica son su valor mínimo y su máximo en el dataset
    dominios ← dominios_de_cada_caracteristica(X)
    centroids ← list([ ])
    foreach j ∈ {0, 1, ..., k − 1} do
        centroid_j = list([ ])
        foreach d ∈ dominios do
            centroid_j.append(aleatorio_entre(min(d),max(d)))
        end
        centroids.append(centroid_j)
    end
    return centroids
end
```

3.2. Búsqueda local

Tratamos ahora el algoritmo búsqueda local paso por paso. Primero describimos como generar una partición de forma aleatoria que nos servirá como punto de partida.

Algorithm 8: generate_initial_sol

```
Data: Conjunto de datos X, número de clusters k
Result: Una partición S
begin
    n ← longitud(X) S ← list([ ])
    foreach i ∈ {0, 1, ..., n − 1} do
        S.append(entero_aleatorio_entre(0,k-1))
    end
    return S
end
```

Una vez tengamos la particion inicial, en cada iteración exploramos el vecindario hasta encontrar una solución mejor y sustituimos la actual por la encontrada. Repetiremos este proceso hasta que exploremos todo el vecindario y no encontremos una solución mejor o hasta que hayamos evaluado la función objetivo 100000 veces.

Cabe destacar dos detalles del algoritmo. El primero es la utilización de un diccionario *assignations_counter* con el objetivo de comprobar que ninguna partición de las que aceptamos durante las distintas iteraciones asigna cero instancias a algún cluster (condición que se requería).

El segundo detalle está en la exploración de los vecindarios. Como se indica en el pseudocódigo, se usa una lista de pares *(indice_de_instancia, adición)*, donde $indice\_de\_instancia \in \{0, 1, ..., n-1\}$ y $adicion \in \{1, 2, ..., k-1\}$. Esto es una manera ingeniosa de representar el vecindario de forma virtual. De esta forma no tenemos que generar el vecindario en cada iteración, sino simplemente barajar esta lista y generar todos los vecinos mediante ella. En concreto lo que hacemos es

$$\text{particion[indice\_de\_instancia]} = (\text{particion[indice\_de\_instancia]} + \text{adición}) \bmod k$$

para generar una partición vecina. De esta forma cubrimos todas las particiones del vecindario asociado a la operación que se nos indica en el guión.

Algorithm 9: Algoritmo de Búsqueda Local

Data: Conjunto de datos X, lista de restricciones *const_list*, número de clusters k, hiperparámetro *lambda*
Result: Partición solución S
begin
n ← longitud(X)
-Declaramos un diccionario 'assignations_counter' para vigilar cuantas instancias tiene cada cluster asignadas (para asegurar que nunca tienen 0)
valid_partition ← False
while *not valid_partition* **do**
S ← generate_initial_sol(X, k)
assignations_counter ← assignations_counts(S)
if *no_zero_values(assignations_counter)* **then**
valid_partition ← True
end
end
current_func_value ← objective_function(X, S, const_list, lambda)
counter ← 1 (Número de veces que se evalua la función objetivo)
-Creamos una lista de parejas [(0,+1),(0,+2),..]. Esta lista representará todas las operaciones de movimiento posibles desde una partición dada.
virtual_neighborhood ← [(index, add) for index in {0,1,...,n-1} for add in {1,2,...,k}]
found_better_sol ← True
while *counter* $\leq$ *100000 and found_better_sol* **do**
found_better_sol ← False
virtual_neighborhood ← RandomShuffle(virtual_neighborhood)
i ← 0
while *counter* $\leq$ *100000 and not found_better_sol and* $i < n$ **do**
operation ← virtual_neighborhood[i]
-Ejecutamos la operación
tmp ← S[operation[0]]
S[operation[0]] ← (S[operation[0]] + operation[1]) mod k
func_val ← objective_function(X, S, const_list, lambda)
counter ← counter + 1
-Si llegamos a una mejor partición que sea válida, la elegimos.
if *func_val < current_func_value and assignations_counter[tmp] > 1* **then**
assignations_counter[tmp] ← assignations_counter[tmp] - 1
assignations_counter[partition[operation[0]]] ←
assignations_counter[partition[operation[0]]] + 1
current_func_value ← func_val
found_better_sol ← True
else
-Si no, volvemos a la particición anterior
S[operation[0]] ← tmp
end
i ← i + 1
end
end
return S
end

4. Procedimiento de desarrollo

Todo el código, desde la lectura de datos hasta los algoritmos, se ha implementado en *Python* y se encuentra en la carpeta *Software*. En concreto, los algoritmos se encuentran en el fichero *algoritmos.py*. La función objetivo y estadísticos comunes a ambos algoritmos explicados en la sección 2 se encuentra en el fichero *funciones_auxiliares_y_estadisticos.py*. Por otro lado, las funciones dedicadas a la lectura y carga de los conjuntos de datos se encuentran en el fichero *leer_datos.py*.

Finalmente, se han desarrollado dos Jupyter Notebook (en *Python* también). La primera con el objetivo de, usando las funciones definidas en los mencionados ficheros, hacer las ejecuciones que se nos requieren en el guión de una forma clara y sin distracciones y obtener las tablas que se nos pide (así como exportarlas al formato *Excel*). Esta Notebook se llama *Ejecución_y_resultados.ipynb* y para la **replica de los resultados**, se recomienda ejecutarla directamente (las semillas están fijadas en ella). La segunda Notebook, *Análisis_resultados.ipynb*, se ha usado para analizar los resultados y realizar varias visualizaciones; se adjunta parte de ella en las siguientes páginas de esta memoria.

4.1. Entorno y paquetes

Para el desarrollo del proyecto se ha trabajado con Anaconda en Windows 10; en concreto con Python y Jupyter Notebook. Un ordenador con una versión instalada de Python 3 será requerido. Específicamente, se ha usado la version Python 3.7.3. Para instalarla, puedes ejecutar en Anaconda Prompt:

```
conda install python=3.7.3
```

Los paquetes NumPy (1.16.2) y Matplotlib (3.0.3) son usados, los puedes obtener con la herramienta pip:

```
pip3 install numpy
pip3 install matplotlib
```

Los siguientes paquetes son también requeridos para ejecutar todo el código:

- seaborn (0.9.0) para mejorar las visualizaciones.
- pandas (0.24.2) para tratar con los resultados, crear las tablas y trabajar con ellas.

Los puedes instalar con conda:

```
conda install -c anaconda seaborn
conda install -c anaconda pandas
```

5.- Análisis de los resultados

En esta sección describiremos los experimentos realizados y estudiaremos los resultados obtenidos.

En el PAR consideraremos 5 ejecuciones con semillas de generación de números aleatorios diferentes para cada algoritmo en cada conjunto de datos con su conjunto de restricciones. Esto quiere decir que un algoritmo cualquiera, se ejecutará 5 veces por cada conjunto de datos, por lo que supondrá un total de 30 ejecuciones por algoritmo. Las semillas elegidas para cada una de las 5 ejecuciones de los algoritmos, como se puede comprobar en la Notebook `Ejecucion_y_resultados.ipynb` , son `seeds = [0, 14, 17, 25, 31]` . Estas han sido elegidas de forma arbitraria siempre comprobando que la ejecución del algoritmo Greedy (COPKM) no cicla de forma infinita con la inicialización pseudoaleatoria de los centroides dada por cada semilla.

El resultado final de las medidas de validación será calculado como la media de los 5 valores obtenidos para cada conjunto de datos y restricciones. Como se indica en el guión, para facilitar la comparación de algoritmos en las prácticas del PAR se considerarán cuatro estadísticos distintos denominados Tasa_C (la desviación general de la partición solución en las distintas ejecuciones), Tasa_inf (el $infeasibility$), Agregado (la evaluación de la función objetivo) y el tiempo de ejecución T.

Recordamos que la función objetivo venía dada por:

$$f = \hat{C} + \lambda * infeasibility$$

Así, es importante mencionar que en los siguientes resultados el hyperparámetro λ utilizado ha sido el cociente entre la distancia máxima existente en el conjunto de datos, $D = \max_{x_i, x_j \in X}\{distance(x_i - x_j)\}$, y el número de restricciones del problema $|R|$:

$\lambda = \frac{D}{|R|}$.

In [1]:
```
import os
import pandas as pd
from leer_datos import *
from funciones_auxiliares_y_estadisticos import *
from algoritmos import *
```

In [2]:
```
#Cargamos las tablas
results_folder = os.pardir + "/Results/"
dataframes = np.load(results_folder + 'dataframes_algorithms.npy',allow_pickle='TRUE').item()
global_results_dfs = np.load(results_folder + 'dataframes_global_comparison.npy',allow_pickle='TRUE').item()
```

Resultados obtenidos con el algoritmos Greedy (COPKM)

Resultados obtenidos en cada conjunto de datos dado el 10% del total de restricciones:

In [3]:
```
dataframes[("COPKM", 10)]
```

Out[3]:

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.173456	35.000000	0.395463	0.180694	201.296981	124.00000	204.623845	3.581947	0.426085	0.0	0.426085	0.167952
Ejecución 2	0.148829	0.000000	0.148829	0.231419	222.752358	126.00000	226.132881	10.429566	0.426085	0.0	0.426085	0.178700
Ejecución 3	0.243392	83.000000	0.769865	0.167850	232.058982	115.00000	235.144379	14.684767	0.593006	31.0	0.816521	0.167218
Ejecución 4	0.148829	0.000000	0.148829	0.167587	240.091561	301.00000	248.167254	8.954754	0.426085	0.0	0.426085	0.167369
Ejecución 5	0.148829	0.000000	0.148829	0.163595	234.233805	365.00000	244.026590	9.654126	0.426085	0.0	0.426085	0.166170
Media	0.172667	23.600000	0.322363	0.182229	226.086737	206.20000	231.618990	9.461032	0.459469	6.2	0.504172	0.169482
Desviación típica	0.036626	32.647205	0.243288	0.025260	13.592848	105.55643	15.485846	3.551977	0.066768	12.4	0.156175	0.004645

Resultados obtenidos en cada conjunto de datos dado el 20% del total de restricciones:

In [4]:
```
dataframes[("COPKM", 20)]
```

Out[4]:

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.144022	21.000000	0.210594	0.174871	240.088037	90.000000	241.295366	3.699347	0.426085	0.0	0.426085	0.172562
Ejecución 2	0.148829	0.000000	0.148829	0.189957	235.489955	250.000000	238.843648	9.898984	0.426085	0.0	0.426085	0.123466
Ejecución 3	0.186499	60.000000	0.376706	0.297462	177.463791	59.000000	178.255263	2.910767	0.420902	18.0	0.485765	0.116006
Ejecución 4	0.148829	0.000000	0.148829	0.175524	249.287102	422.000000	254.948136	15.851435	0.426085	0.0	0.426085	0.172000
Ejecución 5	0.148829	0.000000	0.148829	0.171347	197.459443	78.000000	198.505795	3.559698	0.426085	0.0	0.426085	0.172146
Media	0.155402	16.200000	0.206757	0.201832	219.957665	179.800000	222.369642	7.184046	0.425048	3.6	0.438021	0.151236
Desviación típica	0.015660	23.361507	0.088277	0.048238	27.635481	139.002734	29.005158	5.020647	0.002073	7.2	0.023872	0.025828

Análisis

Las dos tablas previamente mostradas nos permiten comparar la efectividad del algoritmo Greedy en los distintos conjuntos de datos. En primer lugar, observamos (tanto en la primera como en la segunda tabla) que los peores resultados medios en los 4 estadísticos se obtienen con el conjunto de datos `Ecoli` . Para explicar este fenómeno es importante recordar los tamaños de cada conjunto de datos:

- Ecoli: 336 instancias con 7 características. Tiene 8 clases ($k = 8$).
- Iris: 150 instancias con 4 características. Tiene 3 clases ($k = 3$).
- Rand: 150 instancias con 2 características. Tiene 3 clases ($k = 3$).

El mayor tamaño del dataset Ecoli, unido con el mayor número de clusters ($k = 8$), parece ser una explicación clara para que sus tiempos de ejecución sean bastante mayores y sus desviaciones generales (Tasa_C) peores. Además, un mayor número de instancias implicará que el 10% (o 20%) del total de restricciones en el dataset Ecoli son muchas más que el 10% (o 20%) del total de restricciones de los otros dos datasets (pues estos tienen un menor número de instancias y por tanto el total de restricciones será menor). Este hecho parece ser la razón por la que la $infeasibility$ (Tasa_inf) es bastante mayor con el dataset Ecoli.

Los datasets `Iris y Rand` parecen ser más parecidos en tamaño y vemos que sus resultados son más similares. Sin embargo, podemos comprobar también que la $infeasibility$ es considerablemente menor de media en el dataset Rand, y que la Tasa_C es menor de media en Iris. En cuanto al tiempo de ejecución, vemos que es mayor para el dataset Iris. Esto podría ser simplemente porque el dataset Iris tiene 4 características por instancia frente a 2 que tiene el dataset Rand.

Finalmente, estas dos tablas nos permiten ver también la diferencia de los resultados cuando usamos el 10% de las restricciones y cuando usamos el 20% de estas. Vemos que con todos los datasets obtenemos una mejora en todos los estadísticos excepto en el tiempo, donde no se aprecian muchas diferencias. Cuando le damos más restricciones a nuestro algoritmo lo que estamos haciendo es darle parte de la solución, una información que puede ser clave para formar los clusters correctos. Esto podría explicar la mejoría en la Tasa_C y la Tasa_inf (y por ende en la función objetivo (Agr.)).

Resultados obtenidos por el algoritmo de Búsqueda Local

Resultados obtenidos en cada conjunto de datos dado el 10% del total de restricciones:

In [6]:
```
dataframes[("BL", 10)]
```

Out[6]:

	Iris				Ecoli				Rand			
	Tasa_C	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**	**Tasa_C**	**Tasa_inf**	**Agr.**	**T**
Ejecución 1	0.173064	11.000000	0.242838	3.549502	90.488749	1076.000000	119.357342	164.824874	0.567493	17.000000	0.690066	5.421015
Ejecución 2	0.147961	10.000000	0.211392	5.472892	88.731481	870.000000	112.073186	246.646611	0.471133	9.000000	0.536025	4.206658
Ejecución 3	0.164955	14.000000	0.253758	4.246338	82.376373	882.000000	106.040033	185.752572	0.488334	5.000000	0.524385	4.489495
Ejecución 4	0.148829	0.000000	0.148829	4.815099	88.213647	653.000000	105.733341	194.677624	0.426085	0.000000	0.426085	4.098372
Ejecución 5	0.148829	0.000000	0.148829	6.975041	73.927553	1154.000000	104.888850	231.527741	0.481247	7.000000	0.531718	4.364523
Media	0.156728	7.000000	0.201129	5.011774	84.747561	927.000000	109.618550	204.685884	0.486858	7.600000	0.541656	4.516013
Desviación típica	0.010356	5.865151	0.044912	1.168748	6.058741	175.544866	5.498000	30.095593	0.045774	5.571355	0.084630	0.471764

Resultados obtenidos en cada conjunto de datos dado el 20% del total de restricciones:

In [7]:
```
dataframes[("BL", 20)]
```

Out[7]:

	Iris				Ecoli				Rand			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0.164327	13.000000	0.205538	7.089431	85.653598	1799.000000	109.786775	493.439849	0.476127	44.0	0.634680	10.438419
Ejecución 2	0.164935	19.000000	0.225167	7.804216	93.670943	1459.000000	113.243097	344.433477	0.546218	34.0	0.668736	7.594651
Ejecución 3	0.172664	58.000000	0.356531	6.142935	87.998345	1733.000000	111.246147	328.455965	0.426085	0.0	0.426085	8.187196
Ejecución 4	0.148829	0.000000	0.148829	7.712220	81.396033	2436.000000	114.074420	352.515751	0.536014	45.0	0.698171	6.599174
Ejecución 5	0.181723	75.000000	0.419481	7.238607	93.999634	1456.000000	113.531544	467.763391	0.619710	54.0	0.814298	7.056082
Media	0.166496	33.000000	0.271109	7.197482	88.543710	1776.600000	112.376397	397.321687	0.520831	35.4	0.648394	7.975105
Desviación típica	0.010855	28.544702	0.100672	0.593046	4.812382	358.012625	1.609498	68.917279	0.065768	18.8	0.126511	1.341300

Análisis

Recordamos que este algorimo buscaba minimizar la función objetivo $f = \hat{C} + \lambda * infeasibility$. Así, en este caso el hiperparámetro λ no solo juega su papel a la hora de calcular el Agregado (Agr.) que vemos en las tablas, sino que también juega un papel crucial durante la ejecución del algoritmo. Un λ alto dará mayor importancia a minimizar la $infeasibility$, mientras que un λ bajo hará que el algoritmo se centre en minimizar la desviación general (Tasa_C). Como se comentó al principio de esta sección, en estos resultado se ha usado $\lambda = \frac{D}{|R|}$. En general, comprobamos que los valores de la Tasa_inf son bastante altos, esto quizás se podría corregir dandole un valor más alto al hiperparámetro. Vemos los valores específicos de λ para cada dataset (se despejan facilmente de cualquier fila de las tablas, pues este no hiperparámetro cambia en las distintas ejecuciones):

- Ecoli con 10%: $\lambda = 2.682 * 10^{-2}$
- Ecoli con 20%: $\lambda = 1.341 * 10^{-2}$
- Iris con 10%:$\lambda = 6.34 * 10^{-3}$
- Iris con 20%:$\lambda = 3.17 * 10^{-3}$
- Rand con 10%:$\lambda = 7.21 * 10^{-3}$
- Rand con 20%:$\lambda = 3.60 * 10^{-3}$

(Comprobamos que tiene sentido, pues cuando aumentamos las restricciones del 10% al 20%, estamos multiplicando el número de restricciones por 2, y por tanto dividiendo el parámetro λ entre 2).

Centrándonos en comparar los resultados según el dataset, nos encontramos con una situación parecida a la que se explica más detalladamente en el análisis anterior (de los resultados del algoritmo Greedy). Los resultados para el dataset de mayor tamaño, el `Ecoli` , son los peores. Cabe mencionar que los tiempos de ejecución con este algoritmo son del todo apreciables, en concreto cuando lo ejecutamos en el dataset Ecoli.

Comparación de ambos algoritmos

Resultados obtenidos en cada conjunto de datos dado el 10% del total de restricciones:

In [8]:
```
global_results_dfs[10]
```

Out[8]:

	Iris				Ecoli				Rand			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	0.172667	23.6	0.322363	0.182229	226.086737	206.2	231.61899	9.461032	0.459469	6.2	0.504172	0.169482
BL	0.156728	7.0	0.201129	5.011774	84.747561	927.0	109.61855	204.685884	0.486858	7.6	0.541656	4.516013

Resultados obtenidos en cada conjunto de datos dado el 20% del total de restricciones:

In [9]:
```
global_results_dfs[20]
```

Out[9]:

	Iris				Ecoli				Rand			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	0.155402	16.2	0.206757	0.201832	219.957665	179.8	222.369642	7.184046	0.425048	3.6	0.438021	0.151236
BL	0.166496	33.0	0.271109	7.197482	88.543710	1776.6	112.376397	397.321687	0.520831	35.4	0.648394	7.975105

Análisis:

Comparamos a continuación la eficacia de ambos algoritmos en términos de los 4 estadísticos. En primer lugar, vemos que en general para todos los datasets, y dadas ya sea el 10% o el 20% de las restricciones, se tiene que los tiempos de ejecución son mucho mayores en el caso del algoritmo de búsqueda local. Esto se debe a que el algoritmo greedy, cuando no cae en ciclos, necesita menos iteraciones para converger de las que necesita el algoritmo de búsqueda local para para llegar a la mejor solución de un entorno.

Por otro lado, en el caso de 10% de restricciones, vemos que el algoritmo de COPKM obtiene unos resultados peores en términos de Tasa_C, Tasa_inf (y por ende Agr.) en los dos primeros datasets, Iris y Ecoli, no siendo así en el último dataset Rand, en el que el algoritmo Greedy obtiene resultados un poco mejores. Si vamos al caso del 20%, nos encontramos con que el Greedy tiene mejores resultados en los datasets Iris y Rand, mientras que vuelve a ser poco efectivo (en comparación) con el dataset Ecoli. Esto nos podría indicar que el algoritmo de Búsqueda Local funciona considerablemente mejor para datasets de gran tamaño como el de Ecoli, mientras que para datasets de menor tamaño no hay un algoritmo claramente vencedor.

Comparando ahora ambas tablas, vemos que el algoritmo que mejores resultados, en términos del Agregado, obtiene para los datasets Iri y Rand es el COPKM cuando se ejecuta con el 20% de las restricciones. Mientras tanto, el mejor Agregado obtenido para el dataset de mayor tamaño, el Ecoli, es el que obtiene el algoritmo de Búsqueda Local con el 10% de las restricciones. Cabe señalar que las diferencias entre los mejores resultados del COPKM y la Búsqueda Local en los datasets Iris y Rand son bastante menores (incluso en proporción) que las diferencias en los resultados del dataset Ecoli.

Representación gráfica de las soluciones

Para tener una idea gráfica de los clusters que están construyendo nuestros algoritmos, se proporcionan dos visualizaciones a continuación de las instancias y los centroides representados según cada par de características. Los puntos respresentarán las instancias del conjunto de datos; su color, el cluster al que han sido asignadas. Los puntos de mayor grosor y de color gris representarán los centroides. Por desgracia, estas representaciones no nos proporcionan una visualización de las restricciones.

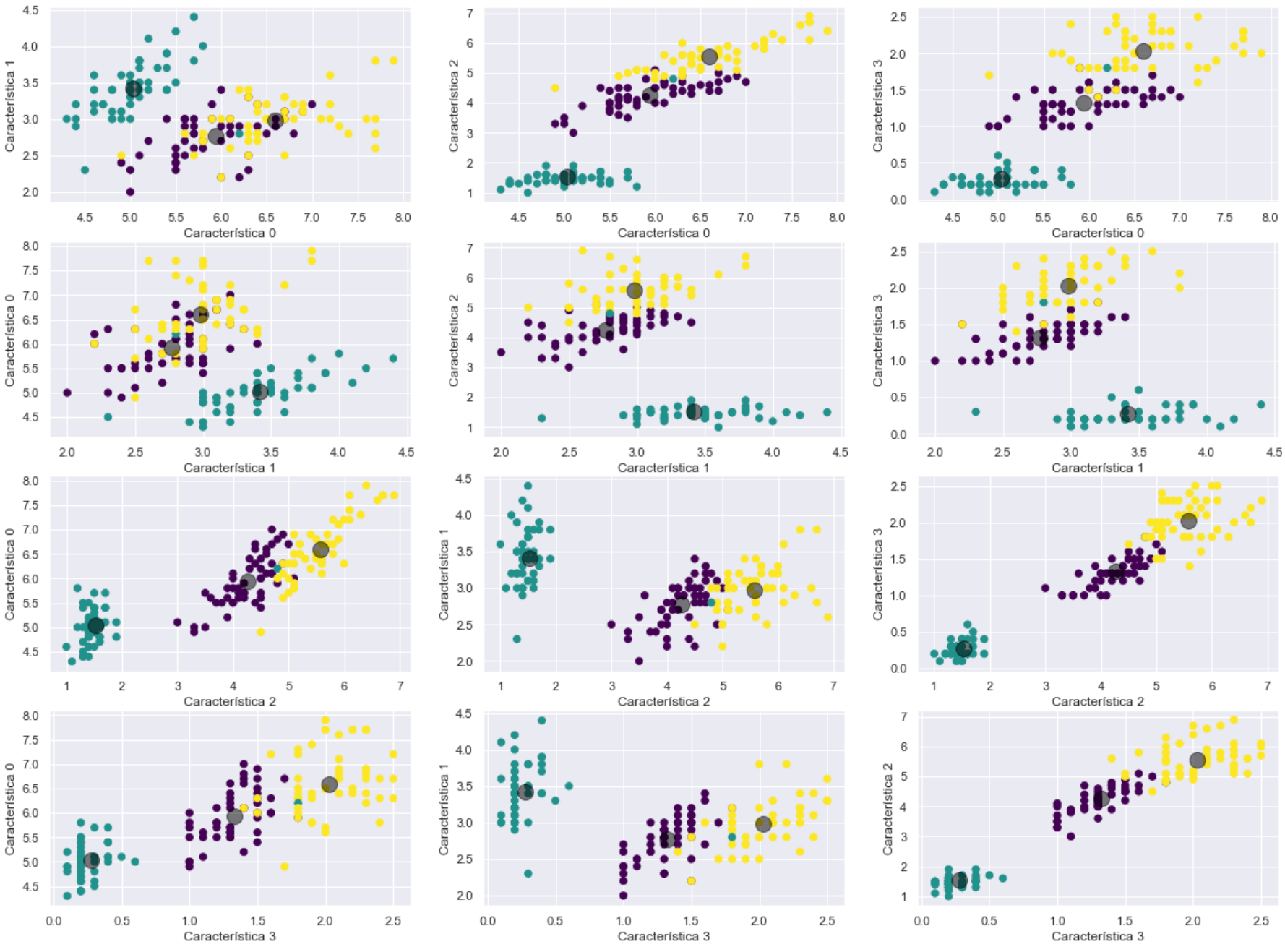
Representación de la partición solución obtenida por el algoritmo de **COPKM** en el dataset **Rand** con el 10% de las restricciones:

In [10]:
```
#Leemos los datos
X = read_dat("./data/rand_set.dat")
const_matrix = read_constraints_matrix("./data/rand_set_const_10.const")
const_list = constraints_matrix_to_list(const_matrix)
#Fijamos semilla
np.random.seed(0)
#Ejecutamos el algoritmo:
partition_sol = copkm_algorithm_with_ini(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_rand_clusters(X, partition_sol)
```

Representación de la partición solución obtenida por el algoritmo **Búsqueda Local** en el dataset **Iris** con el 10% de las restricciones:

In [11]:
```
#Leemos los datos
X = read_dat("./data/iris_set.dat")
const_matrix = read_constraints_matrix("./data/iris_set_const_10.const")
const_list = constraints_matrix_to_list(const_matrix)
#Fijamos semilla
np.random.seed(0)
#Ejecutamos el algoritmo:
partition_sol = local_search(X, const_matrix, const_list, k=3)
# Visualizamos la solución obtenida
visualise_iris_clusters(X, partition_sol)
```

Representamos las instancias y los clusters según las diferentes características (2D)



6.- Extra: Experimentando con el hiperparámetro λ

Comprobamos con un rápido ejemplo que comportamiento siguen los estadísticos aplicados a los resultados del algoritmo de búsqueda local cuando cambiamos la lambda.

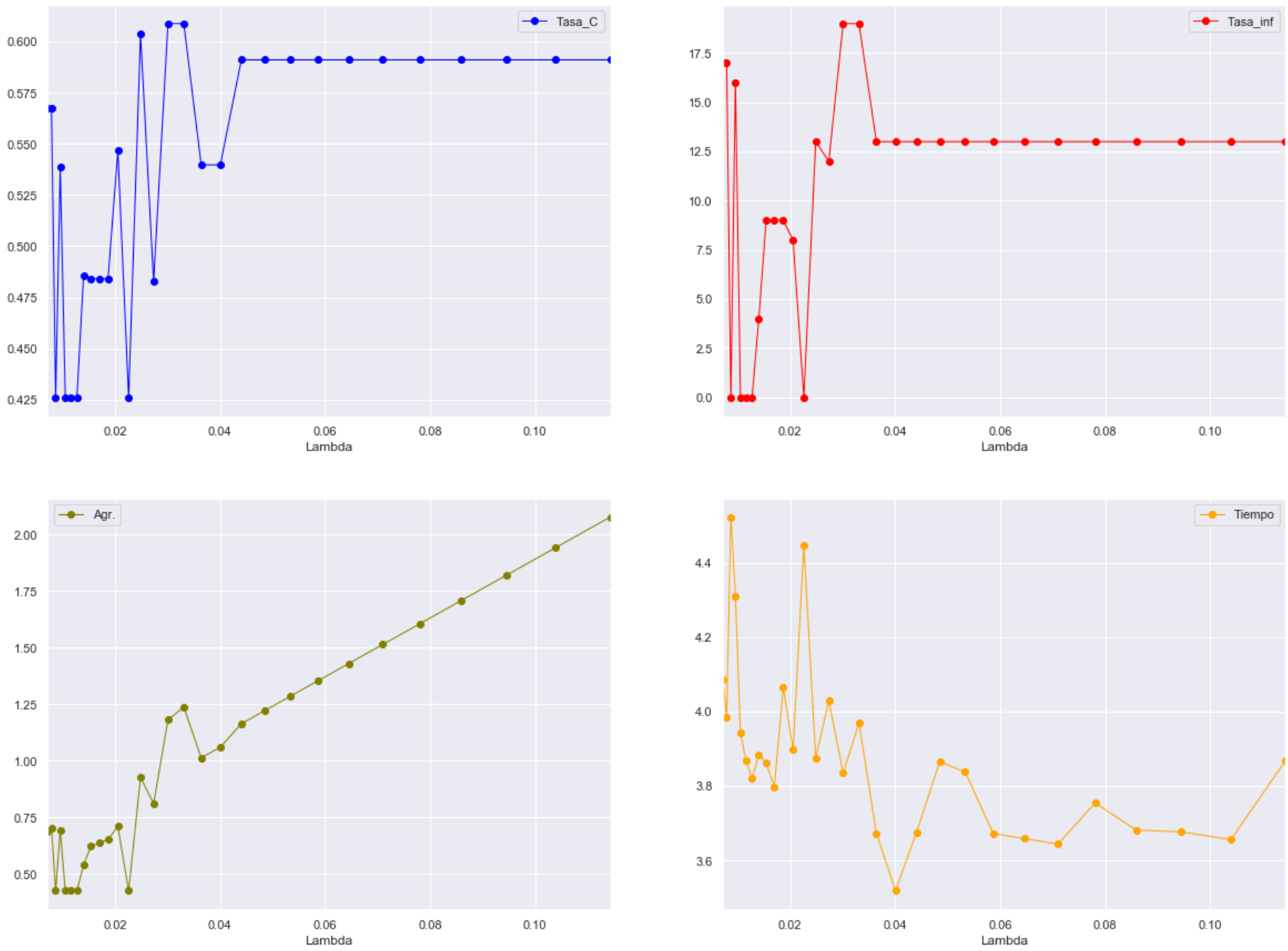
In [13]:
```
#Extra: Change lambda and plot
def local_search_with_different_lambdas(dat_file, const_file, k, seed, n_lambdas, factor_to_add = 1.3):
    results = {}
    #Leemos los datos
    X = read_dat(dat_file)
    const_matrix = read_constraints_matrix(const_file)
    const_list = constraints_matrix_to_list(const_matrix)
    lambda_ini = max_dist(X) / len(const_list)
    lambdas = [lambda_ini * factor_to_add**n for n in range(n_lambdas)]
    for lambda_ in lambdas:
        np.random.seed(seed)
        #Ejecutamos el algoritmo
        t0 = time.perf_counter()
        partition_sol = local_search(X, const_matrix, const_list, k, lambda_)
        t1 = time.perf_counter()
        tiempo = t1 - t0
        tasa_C = general_deviation(X, partition_sol)
        tasa_inf = infeasibility(partition_sol, const_list)
        agr =  objective_func(X, partition_sol, const_list, lambda_ = lambda_)
        results[lambda_] = tasa_C, tasa_inf, agr, tiempo
    return pd.DataFrame.from_dict(results, orient='index', columns = ["Tasa_C", "Tasa_inf", "Agr.", "Tiempo"])
```

In [14]:
```
results_lambda = local_search_with_different_lambdas("./data/rand_set.dat", "./data/rand_set_const_10.const", k = 3,
                                        seed = 0, n_lambdas = 30, factor_to_add = 1.1)
```

Representamos en gráficos los estadísticos con respecto a los diferentes lambdas:

In [25]:
```
results_lambda.index.name = 'Lambda'
fig, axes = plt.subplots(nrows=2, ncols=2, figsize = (20,15))
results_lambda.reset_index().plot(x='Lambda', y='Tasa_C', linewidth=1, color = 'blue', marker='o', ax=axes[0,0])
results_lambda.reset_index().plot(x='Lambda', y='Tasa_inf', linewidth=1, color = 'red', marker='o', ax=axes[0,1])
results_lambda.reset_index().plot(x='Lambda', y='Agr.', linewidth=1, color = 'olive', marker='o', ax=axes[1,0])
results_lambda.reset_index().plot(x='Lambda', y='Tiempo', linewidth=1, color = 'orange', marker='o', ax=axes[1,1])
```

Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x16acac30f60>



En las gráficas de los estadísticos Tasa_inf y Tasa_C (por ende Agr. también) comprobamos que cuando nos movemos por valores de λ pequeños cercanos al valor que se ha fijado por defecto durante las ejecuciones del análisis (apartado anterior), ($\lambda = \frac{D}{|R|}$), es cuando se obtienen los mejores resultados. También observamos que cuando el valor de lambda sigue creciendo, los resultados (de Tasa_inf y Tasa_C) tienden a ser bastante peores. Esto puede ser debido a que la $infeasibility$ toma todo el protagonismo en la función objetivo y por tanto el algoritmo no tiene en cuenta apenas la desviación general.

In [ ]:
```
-
```