METAHEURÍSTICAS

PRÁCTICA 1: Técnicas de Búsqueda Local y Algoritmos Greedy

UNIVERSIDAD
DE GRANADA

Problema: Agrupamiento con Restricciones

Alba Casillas Rodríguez
76738108B
3ºA – Grupo de Prácticas 1 (Miércoles 17:30-19:30)
albacaro@correo.ugr.es

Índice

1 - Descripción del problema

PROBLEMA DEL AGRUPAMIENTO CON RESTRICCIONES

El agrupamiento, también llamado ***clustering***, es una tarea de aprendizaje no supervisado que consiste en agrupar un conjunto de objetos de tal manera que los objetos que estén en el mismo ***grupo (cluster)*** sean más ***similares*** entre sí que con los de otros grupos (clusters).

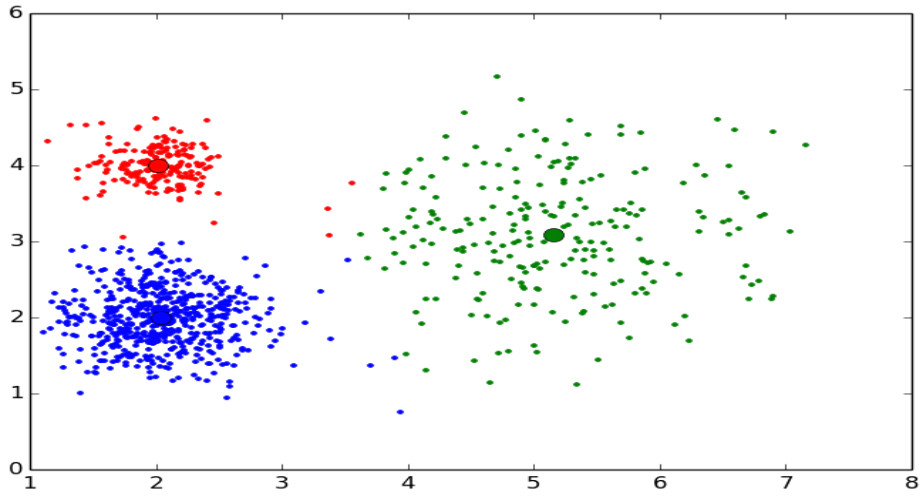
Esta similitud entre los objetos está definido mediante una ***distancia***, normalmente la Euclídea; por lo que haremos uso del término ***centroide***, siendo este el centro geométrico del cluster.

En nuestro problema, al clustering clásico le añadiremos restricciones; convirtiendo este problema en uno de aprendizaje semi-supervisado.

La partición de nuestro grupo de datos, deberá tener en cuenta ***restricciones fuertes***, que obligan a que todos los clusters deben contener al menos una instancia, cada instancia debe pertenecer a un único cluster, y la unión de los clusters debe ser el conjunto de datos.

Además, también deberán cumplirse unas ***restricciones débiles***, donde la partición del conjunto de datos debe minimizar el número de restricciones incumplidas (pudiendo incumplir algunas). Estas restricciones consisten en que dada una pareja de instancias, algunas deberán pertenecer al mismo cluster (***must-link***); y otras, por el contrario, no podrán estar en un mismo cluster (***cannot-link***).

El clustering es una tarea muy importante en exploración de data mining, y una técnica común en el análisis estadístico de datos, aprendizaje automático, reconocimiento de imágenes, etc.



2 - Consideraciones comunes de los algoritmos

- *2.1 - Carga de datos :* Leeremos el conjunto de datos y el conjunto de restricciones desde los ficheros proporcionados; de forma que los datos leídos se guardaran en una matrix nxd (para "n" el número de instancias de datos en un espacio de "d" dimensiones), y una matriz nxn ; para el conjunto de datos y restricciones, respectivamente.

Tendremos en cuenta que en los ficheros, los datos estarán separados por comas (","), las cuales deberemos omitir.

```
def carga_datos(fichero_datos, fichero_restricciones):
        f_datos = open(fichero_datos)
        f_restric = open(fichero_restricciones)

        para cada linea del fichero de datos:
                #Leeremos hasta el salto de linea omitiendo las comas
                linea.rstrip('\n').split(",")
                matriz_datos ← linea

        n = longitud(matriz_datos) #numero de filas/instancias
        d = longitud(linea) #numero de columnas/dimensiones

        #Repetir el mismo proceso para el conjunto de restricciones
```

- *2.2 - Representación de la solución :* Representamos nuestra solución como un vector de tamaño "n", cuyos valores van desde 1 hasta "k", siendo este el número de clusters definido.

$$\text{Solución} = \{S_1 , S_2 , \dots , S_n\} \setminus S_i \in K$$

Inicialmente, esta solución es creada mediante un generador de números aleatorios.

- *2.3 – Función objetivo :* Será nuestra función a minimizar. Está se calculará mediante la fórmula:

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

- C será la desviación general.
- Infeasibility será el número de restricciones incumplidas.
- λ(lambda) será yb parámetro de escalado para dar relevancia al infeasibility.

- *2.4 – Cálculo del Infeasibility :* Llamaremos infeasibility al número de restricciones incumplidas dada una pareja de instancias del conjunto de datos.

$$\text{Infeasibility} = \Sigma^{n}_{i=0}\ \Sigma^{n}_{j=i+1} V(x_i, x_j)$$

Por tanto, recorreremos el vector solución en un doble bucle para formar parejas, asegurándonos previamente de que ya tienen un clúster asignado; con las cuales buscar qué restricciones ha de cumplir en la matriz de restricciones. Si se incumple, el valor de infeasibility aumenta.

```
def calcular_infeasibility(datos, v_solucion, restricciones):
    para i ∈ {0...n}:
        si v_solucion[i] != 0 : #Si tiene un cluster asignado
            for j ∈ {i...n}:
                si v_solucion[j] != 0:
                    # INCUMPLE CANNOT-LINK
                    si restricciones[i][j] == '-1' && v_solucion[i] == v_solucion[j]
                        infeasibility++

                    # INCUMPLE MUST-LINK
                    si restricciones[i][j] == '1' && v_solucion[i] != v_solucion[j]
                        infeasibility++
```

- *2.5 – Distancia y desviación :* Uno de los criterios esenciales a la hora de asignar a una instancia del conjunto de datos un clúster, será la distancia media intra-cluster; con la que podremos calcular a su vez la desviación general requerida en la función objetivo.

Para ello, implementaremos una función que calcule la distancia media intra-cluster, siguiendo la fórmula proporcionada:

$$\text{Distancia intra-cluster} = c_i = 1/|c_i|\ \Sigma_{x_j \epsilon c_i} ||x_j - \mu_i||_2$$

De manera que recorreremos el conjunto de datos para calcular la distancia euclídea de las instancias correspondientes a un clúster con el centroide del mismo; obteniendo después la distancia media.

```
def calcular_distancia(datos, cluster, centroides, d, v_solucion):
    for i ∈ {0...n}:
        si v_solucion[i] == cluster #si pertenece al cluster indicado
            for j ∈ {0...n}:
                distancia ← sqrt( (centroides[cluster][j] – datos[i][j])²
        contador++

    distancia_media  ← distancia / contador
```

La desviación general será la media de las distancias intra-cluster:

$$\text{Desviacion genereral} = C = 1/k \; \Sigma_{ci \in C} C_i$$

Para cada cluster c_i calcularemos su distancia media intra-cluster. La desviación será la media de estos resultados

```
def calcular_desviacion(datos, centroides, d, v_solucion):
    for i ∈ {0...k}:
        desviacion ← distancia_media(i)
    desviacion_general ← desviacion/k
```

- *2.6 – Lambda :* Este parámetro se encargará de dar relevancia al valor de infeasibility de nuestra función objetivo para así optimizar el número de restricciones incumplidas antes que la desviación general.

Lambda será el cociente entre el valor de la máxima distancia del conjunto de datos y el número de restricciones del problema.

$$\lambda = D/|R|$$

Para ello, primero debemos calcular la distancia máxima de nuestro conjunto. Recorro el conjunto de datos y calculo para cada pareja de instancias la distancia que hay entre ellas, quedándome con el máximo valor.

```
def distancia_maxima(datos):
    for i ∈ {0...longitud(datos)}:
        for j ∈ {i+1...longitud(datos)}:
            for c ∈ {0...d}:
                distancia ← sqrt( (datos[i][c] - datos[j][c])²)
        si distancia > maximo
            maximo ←  distancia

def lamda(datos, num_restricciones):

    distancia <-  distancia_maxima(datos)

    #Para calcular lamda hacemos un casting a "int" ya que en la fórmula se indica que debe ser
    # la parte entera de la distancia máxima
    lamda ←  (int(distancia))/num_restricciones
```

- *2.7 – Cálculo de centroides :* Calcular el centroide asociado a un cluster c_i se hará hallando el vector promedio de las instancias de datos que lo componen

$$\text{centroide} = \mu_i = 1 \ / \ |c_i| \Sigma_{x_j \epsilon i} x_j$$

Recorreremos todos los clústeres, y para cada uno de ellos se hallará su centroide formado por la suma de los valores de las instancias del conjunto de datos que pertenezcan a ese mismo clúster entre el número de clústeres total.

```
def recalcular_centroides(datos, v_solucion, centroides, d):

    for i∈ {0...k}:
        #Calcularemos cada centroide  μi siguiendo la fórmula descrita
        for i∈ {0...n}:
            si v_solucion[j] == i #Si este punto pertenece a al cluster i
                for l ∈ {0...d}:
                    nuevo_centroide[l] += datos[j][l]
                num_total++

        for j ∈ {0...d}:
            nuevo_centroide[j] = nuevo_centroide[j]/num_total

        centroides[i] = nuevo_centroide
```

3 – Descripción de los algoritmos

ALGORITMO GREEDY

La solución greedy para este problema estará basado en el algoritmo ***k-medias***, donde además se tendrá en cuenta el cumplimiento de las ***restricciones*** débiles.

K-medias es un método de agrupamiento en el cual se divide un conjunto de n objetos en k grupos, donde cada objeto pertenece al gurpo cuyo valor medio es más cercano.

Por ello, inicialmente barajaremos los índices para recorrer el conjunto de datos de forma aleatoria sin repetición y asignaremos cada instancia al clúster más cercano de entre los que produzcan un menor aumento en el valor de infeasibility.

Guardaremos la ***infeasibility*** de asignar a cada instancia los clústeres, para que de haber varios con infeasibility minima, podamos elegir aquel que tiene menor distancia dato-centroide.

Una vez hemos obtenido la nueva partición, se deberá ***re-calcular los centroides.***

Iteraremos el algoritmo mientras no haya cambio en la partición.

```
def GREEDY(datos, restricciones, centroides, n, d):
    random.shuffle(rsi) # rsi será un vector cuyos valores van de 0 a n

    mientras que ( hay_cambio ) && ( num_iteraciones < MAX_ITERACIONES=10000):
        num_iteraciones++
        anterior_solucion ← v_solucion

        for i ∈ {0...rsi}:
            for j i ∈ {0...k}:
                v_solucion[i] ← j #Asignamos a cada instancia un cluster "j"
                infeasibility ← caculamos infeasibility(v_solucion)

                #Guardo en un vector los valores de infeasibility de asignar a una
                #instancia Xi al cluster j
                v_infeasibility[j] ← infeasibility

            #De los valores de v_infeasibility, guardaremos en un vector auxiliar
            #(v_minimos) las posiciones de los valores mínimos.
            for c ∈ {0...k}:
                si v_infeasibility[c] < minimo:
                    minimo ←  valor_infeasibility
                    v_minimos ←  posicion
```

```
                    #Tras obtener las posiciones cuyo valor del infeasibility es mínimo.
                    # Nos quedaremos con el centroide que tenga la menor distancia
                    for c ∈ {0...len(v_minimos)}:
                            distancia ← distancia(datos,centroides)
                            v_distancias[c] ← distancia

                    for c ∈ {0...len(v_distancias)}:
                            si v_distancias[c] < minimo:
                                    minimo ←  v_distancias[c]
                                    centroide_cercano ←  v_minimos[c]

                    #Una vez obtenemos el centroide más cercano con menor valor de
                    #infeasibility, lo asignamos al vector solución

                    v_solucion[i]  ← centroide_cercano

            #Recalculamos centroides
            recalcular_centrodies(datos, v_solucion, centroides, d)

            #Comprobamos si ha habido un cambio en la partición
            cambio ← cambio(v_solucion, anterior_solucion)

    return v_solucion
```

BÚSQUEDA LOCAL

La ***búsqueda local*** es un proceso iterativo que empieza en una solución inicial la cual mejoramos mediante modificaciones locales buscando en su vecindad.

Si se encuentra una mejor solución, reemplaza la solución actual por la nueva y continúa (***primero el mejor***).

Nuestra solución inicial se representará mediante un vector de n posiciones cuyos valores irán hasta un valor “k”, representando el número de clusters. Esta solución será aquella que deberemos modificar en busca de una mejor.

Para ello, generaremos un ***vecindario***, recorriendo todos los elementos de la solución, y cada elemento lo añadiremos en todos los cluster exceptuando en el que ya está y teniendo en cuenta de que ningún cluster deberá quedarse vacío.

Una vez generado el vecindario, los recorremos de manera aleatoria; calculando para cada vecino su función objetivo y comparándola con el valor de la función objetivo de la solución actual. Nuestro objetivo será ***minimizar la función*** objetivo hasta que no se encuentre mejora en el entorno o se llegue a las 10000 iteraciones.

```
def BusquedaLocal(datos, restricciones, n, d):

        #Genereamos un vector n aleatoriamentes con valores hasta “k”
        v_solucion  ← generar_solucion_aleatoria(k, n)
        #Calcularemos unos centroides según nuestra solucion inicial
        centroides ← recalcular_centroides(datos, v_solucion, v_centroides, d)
        #Calculamos el valor de lambda para nuestra funcion objetivo
        lambda ← calcular_lambda(datos,restricciones,n)

        mientras que (hay_cambio) and (num_iteraciones < MAX_ITERACIONES):
                num_iteraciones ++

                for i ∈ {0...n}:
                        for j i ∈ {0...k}:
                                Si j ! = v_solucion[i]: #Si no es el cluster en el que ya está
                                        vecinos  ← v_solucion

                                        #Generamos el vecindario asignando a cada elemento un cluster
                                        #nuevo
                                        vecinos.back()[i] = j

                                        #A no ser que el cluster se quede vacío, donde generaríamos
                                        #una solución infactible y por tanto, no  tendremos en cuenta
                                        #este vecino borrándolo
                                        si cluster_vacio():
                                                borrar vecino

                        #Para no coger siempre los primeros vecinos:
                        shuffle(vecinos)

                        #Re-calculamos centroides para cada vecino
                        for vec ∈ {0...n}:
                                recalcular_centroides(datos, vec, centroides[vec], d

                        #Comparamos el valor de la funcion objetivo para la solución actual con el de
                        #cada vecino hasta encontrar uno que sea mejor.
                        #Como nos basamos en una busqueda local de EL PRIMERO EL MEJOR, en
                        #cuanto encontremos una solución que mejore la actual, nos quedaremos con
                        #ella
                        for v ∈ vecinos:
                                sol1  ←  funcion_objetivo(vecino)
                                sol2  ←  funcion_objetivo(v_solucion)

                                si sol1 < sol2
                                        v_solucion ←  v
                                        hay_cambio ← TRUE
                                        break

        return v_solucion
```

4 – Manual de usuario

Para ejecutar la práctica solo debe indicarse qué conjunto de datos se desea ejecutar.

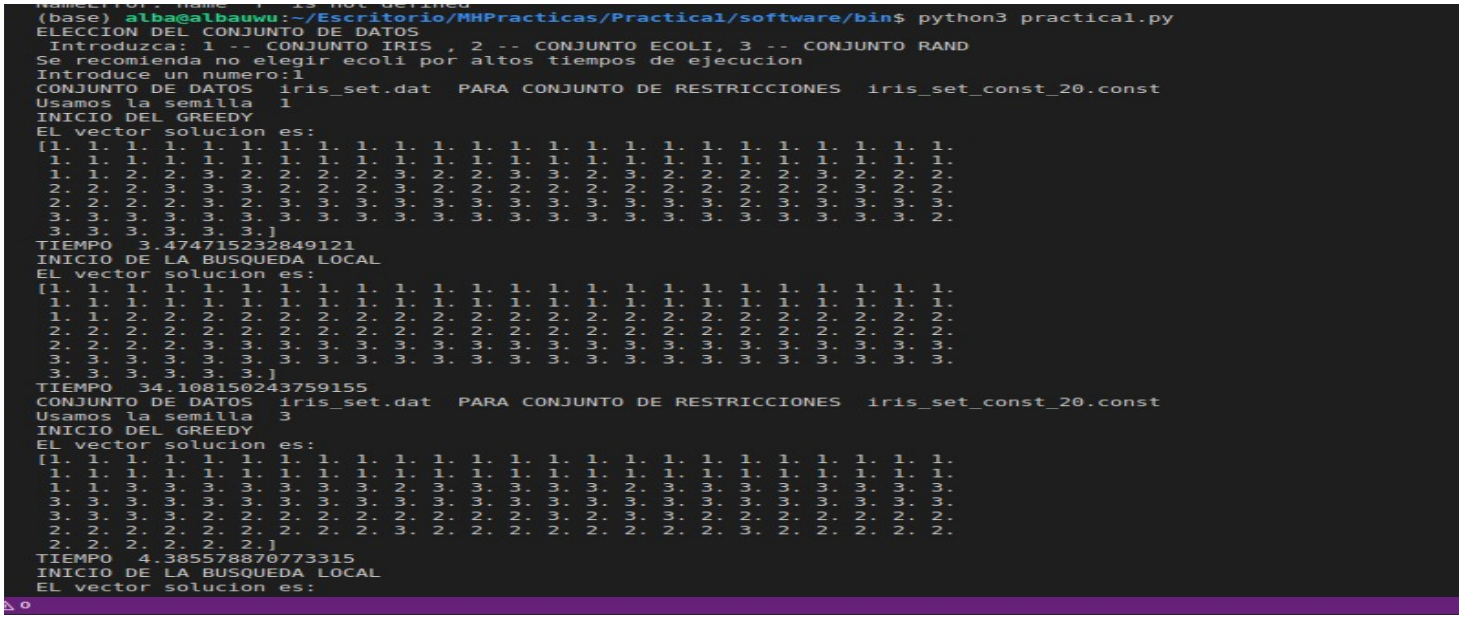
No recomiendo ejecutar el “ecoli” debido a los altos tiempos de ejecución.

Tras indicar el conjunto de datos, se ejecutarán ambos conjuntos de restricciones 5 veces.

NO HACE FALTA INDICAR LAS SEMILLAS.

Las semillas utilizadas: (1,3,2,7, 5) han sido guardadas en un vector, de manera que por cada ejecución se seleccionará una semilla diferente.

Ejemplo de ejecución:



** Cabe destacar que en el programa ejecutable se ha comentado el código con el que se realizan de manera automática las tablas y gráficas proporcionadas para el análisis de resultados.

5 – Tablas de resultados

5.1 – Greedy

Resultados obtenidos en el PAR GREEDY 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	0,108	7	1,045	2,958	7,799	472	678,827	1602,398	0,134	9	1,78	4,043
Ejecución 2	0,117	102	13,765	5,995	7,385	682	976,965	1600,871	0,125	0	0,125	3,233
Ejecución 3	0,107	0	0,107	5,075	8,174	386	556,938	1821,637	0,227	113	20,893	2,945
Ejecución 4	0,217	103	13,999	4,839	8,889	249	362,885	1680,457	0,199	107	19,768	3,907
Ejecución 5	0,148	65	8,846	3,089	8,398	453	652,415	1559,319	0,125	0	0,125	3,907
Media	0,14	55,40	7,55	4,39	8,13	448,40	645,61	1652,94	0,16	45,80	8,54	3,61

Resultados obtenidos en el PAR GREEDY 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	0,123	60	4,266	3,232	10,778	306	231,492	1558,121	0,195050216	137	13,124	3,504
Ejecución 2	0,106	18	1,349	3,103	10,882	275	209,236	1651,675	0,137303561	28	2,779	3,442
Ejecución 3	0,127	323	22,43	3,323	8,34	1021	744,776	1767,392	0,153182552	77	7,419	3,412
Ejecución 4	0,122	40	2,883	3,281	9,667	163	127,237	1638,876	0,125278691	0	0,125	3,265
Ejecución 5	0,14	126	8,84	3,498	10,188	167	130,643	1648,364	0,230758575	288	27,409	3,336
Media	0,12	113,40	7,95	3,29	9,97	386,40	288,68	1652,89	0,17	106,00	10,17	3,39

5.2 – Búsqueda local

Resultados obtenidos en el PAR BÚSQUEDA LOCAL 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	0,106	0	0,106	36,134	8,41	695	996,471	2476,345	0,125	0	0,125	40,88
Ejecución 2	0,106	0	0,106	42,198	7,383	721	1032,407	2436,402	0,125	0	0,125	46,217
Ejecución 3	0,106	0	0,106	26,425	8,363	823	1178,397	2247,221	0,125	0	0,125	29,712
Ejecución 4	0,106	0	0,106	43,814	7,288	609	873,085	2359,086	0,125	0	0,125	29,371
Ejecuión 5	0,107	0	0,107	32,084	6,987	698	999,313	2421,338	0,125	0	0,125	27,428
Media	0,11	0,00	0,11	36,13	7,69	709,20	1015,93	2388,08	0,13	0,00	0,13	34,72

Resultados obtenidos en el PAR BÚSQUEDA LOCAL 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	0,106	0	0,106	42,16	7,392	1452	1054,703	2471,63	0,125	0	0,125	31,971
Ejecución 2	0,107	0	0,107	35,403	8,178	1269	923,493	2303,145	0,125	0	0,125	28,083
Ejecución 3	0,107	0	0,107	26,365	8,729	1160	845,424	2164,985	0,125	0	0,125	27,644
Ejecución 4	0,106	0	0,106	38,699	7,191	1554	1128,074	2131,458	0,125	0	0,125	30,654
Ejecución 5	0,106	0	0,106	36,777	7,16	1483	1076,831	2181,651	0,125	0	0,125	28,737
Media	0,11	0,00	0,11	35,88	7,73	1383,60	1005,71	2250,57	0,13	0,00	0,13	29,42

5.3 – Tabla global comparativa

Resultados globales 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
COPKM	0,14	55,40	7,55	4,39	8,13	448,40	645,61	1652,94	0,16	45,80	8,54	3,64
BL	0,11	0,00	0,11	36,13	7,69	709,20	1015,93	2388,08	0,13	0,00	0,13	34,72

Resultados globales 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
COPKM	0,12	113,40	7,95	3,29	9,97	386,40	288,68	1652,89	0,17	106,00	10,17	3,39
BL	0,11	0,00	0,11	35,88	7,73	1383,60	1005,71	2250,57	0,13	0,00	0,13	29,42

6 – Análisis global de resultados

Compararemos ambos algoritmos basándonos en sus capacidades para obtener soluciones de calidad, el número de restricciones no satisfechas, y rapidez de estos. Estas comparaciones se realizarán sobre tres conjuntos de datos diferentes, sometiéndolos a distintos porcentajes (10% y 20%) de restricciones.

Comenzamos observando los resultados del greedy; podemos observar que durante las 5 ejecuciones se obtienen valores de *"Tasa_C"* (desviación general de la partición) similares; sin embargo, algo que llama la atención en los resultados, es la significativa oscilación que hay en el valor de *"infeasibility"* entre una ejecución y otra, siendo 0 en algunas ejecuciones y superando el centenar de

restricciones incumplidas en otra (afectando de esta manera al valor final de la función objetivo y a la rapidez del algoritmo, cuyos aumentos de sus valores son directamente proporcionales a los del *"infeasibility"*); estas oscilaciones podrían deberse al calculo inicial de los centroides, ya que al ser puramente aleatoria, en algunas ejecuciones podrían haberse situado más alejados del conjunto de datos.

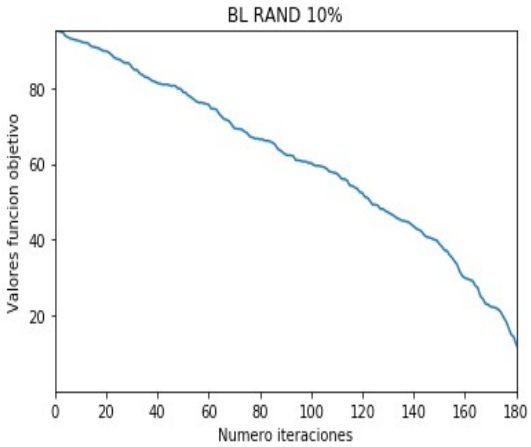
También podemos observar que al aumentar el porcentaje de restricciones (de 10% a 20%), y a pesar de que la desviación general no sufre ningún cambio significativo, lo valores del *"infeasibility"* aumentan, es decir, la calidad del greedy empeora al incumplirse un mayor números de restricciones; al contrario de lo que pasará en la búsqueda local.

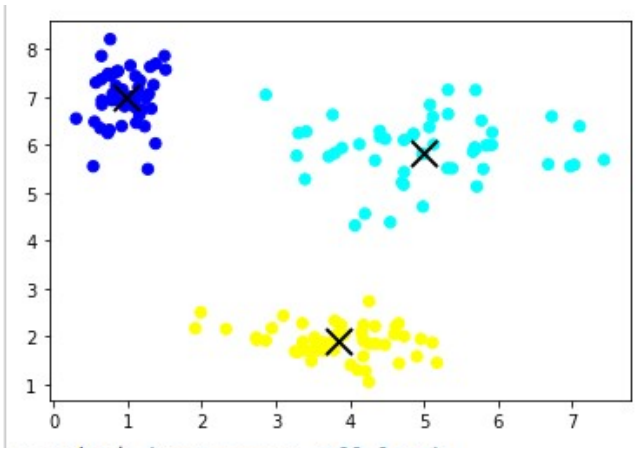
A continuación, viendo los datos de las distintas ejecuciones en la búsqueda local, podemos concluir que, sin importar el porcentaje de restricciones , se obtienen fácilmente soluciones de calidad y en un tiempo de ejecución prácticamente constante, debido a que obtenemos un valor de *"Tasa_C"* baja y con *"infeasibility"* 0, es decir , nuestras particiones resultantes no incumplen ninguna restricción.

Con ello podemos asegurar de que cuando la búsqueda local se detenga, habremos encontrado un mínimo (local o global).

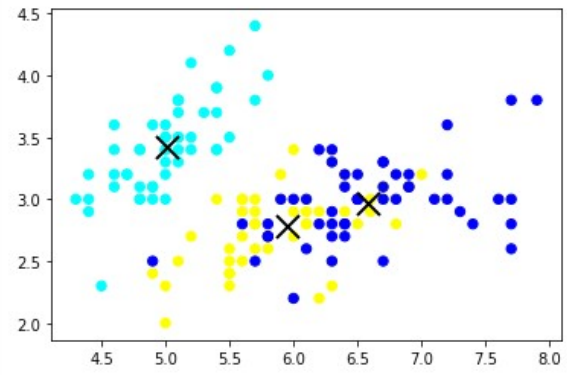
Adjunto gráficas de convergencia y representación gráfica de las soluciones para:

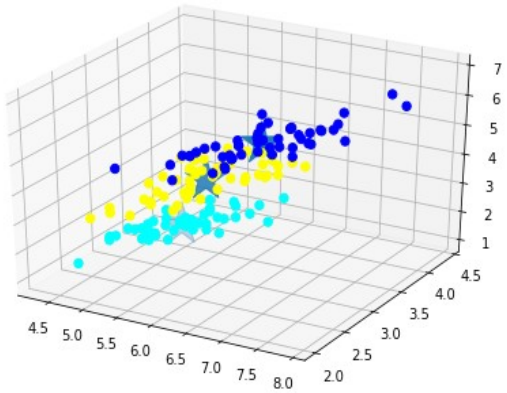
- BÚSQUEDA LOCAL – RAND 10%

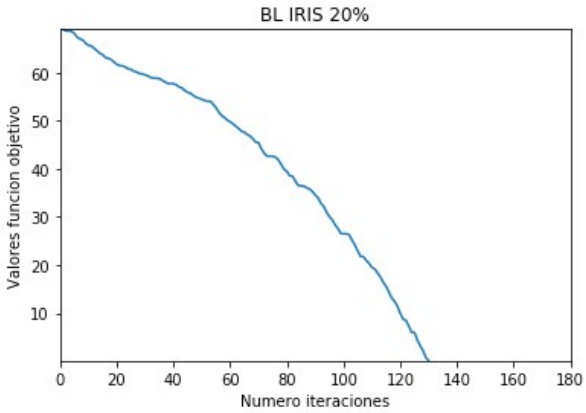




- BÚSQUEDA LOCAL – IRIS 20%







Cómo podemos observar, la búsqueda local converge disminuyendo el valor de la función objetivo a medida que incrementa el número de iteraciones; hasta el punto en el que llega a un óptimo, donde el algoritmo deja de converger y termina su ejecución.

Contrastando los resultados del greedy con nuestro algoritmo principal: la búsqueda local, observamos que los valores de la desviación general de la partición son muy similares en ambos algoritmos, lo que quiere decir que ambos pueden conseguir una partición final con agrupamientos bastantes precisos y de calidad.

Sin embargo, la mayor diferencia se encontrará entonces en las restricciones incumplidas en ambos algoritmos, obteniendo claramente agrupamiento más factibles en la búsqueda local.

Para una solución de calidad, la búsqueda local tardará más tiempo de ejecución en conseguirlo. Esto se debe a que, a pesar de modificar la solución cuando se produce alguna mejora inmediata (puesto que usamos la filosofía de “primero el mejor”) encontrando el mínimo más próximo a la solución iniciada aleatoriamente , deberá generar y explorar continuamente el vecindario en busca de soluciones más óptimas; al contrario del greedy que se centra en maximizar el criterio de mejora quedándonos con el *“infeasibility”* más pequeño posible .

7 – Bibliografía

- Entendimiento del problema

- Tema 2: Modelos de Búsuqeda: Entornos y Trayectorias vs Poblaciones
- Seminario 2: Problemas de optimización con técnicas basadas en búsqueda local
-https://en.wikipedia.org/wiki/Cluster_analysis#Definition
-https://ccc.inaoep.mx/~emorales/Cursos/Busqueda/node58.html

-Programación del problema

- https://stackoverflow.com/questions/4319236/remove-the-newline-character-in-a-list-read-from-a-file
- https://docs.python.org/3/library/

-Análisis de resultados y muestra de resultados

- https://www.aprendemachinelearning.com/k-means-en-python-paso-a-paso/
- https://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html
-https://datatofish.com/export-dataframe-to-excel/