Prácticas de Metaheurísticas

Doble Grado en Ingeniería Informática y Matemáticas, UGR

Práctica 1.b

Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento con Restricciones

Fernández Gutiérrez, Miguel Ángel

mianfg@correo.ugr.es

Índice

I | Descripción del problema

El **problema de agrupamiento de restricciones** es una generalización del problema de agrupamiento convencional, al que le añadimos una serie de restricciones. Se formula como sigue:

> **Problema** (PAR, restricciones débiles). Dado un conjunto de datos $X = \{\vec{x_1}, ..., \vec{x_n}\}$, encontrar una partición $C = \{c_1, ..., c_k\}$, de modo que se minimicen la desviación general, $\bar{C}$, y la *infeasibility*(C) (número de restricciones de R, el conjunto de restricciones, incumplidas).

Definiremos a continuación ciertos parámetros y elementos de el problema:[1]

- **Conjunto de datos**, X. Matriz de $n \times d$ valores reales, siendo n el número de instancias (o datos) y d la dimensión de cada instancia (todas las instancias tienen la misma dimensión). Lo denotamos como:

$$X = \{\vec{x_1}, ..., \vec{x_n}\},\ \vec{x_i} = (x_{[i,1]}, ..., x_{[i,d]}),\ x_{[i,j]} \in \mathbb{R}\ \ \forall i \in \{1, ..., n\}, j \in \{1, ..., d\}$$

- **Partición**, C. Es una asignación de cada $\vec{x_i}$ a un cluster:

$$C = \{c_1, ..., c_k\}, c_i \subset \{1, ..., n\}$$

  Para facilitar su representación usamos un vector de clusters, con una correspondencia uno a uno con los datos. De este modo, $\texttt{clusters}[i] = j \iff j \in c_i$.

- **Centroide**, μ_i. Es el vector promedio de las instancias de X que componen al cluster:

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

- **Desviación general de la partición**, $\bar{C}$. Es la media de las **desviaciones intra-cluster** para cada cluster, $\bar{c_i}$. Definimos ambas a continuación:

$$\bar{c_i} = \frac{1}{|\,c_i\,|} \sum_{j \in c_i} \|\vec{x_j} - \vec{\mu_i}\|_2,\ \ \bar{C} = \frac{1}{k} \sum_{c_i \in C} \bar{c_i}$$

- La validez de una partición C está sujeta al cumplimiento de las siguientes restricciones:
  - $c_i \neq \emptyset\ \forall i \in \{1, ..., k\}$ (no hay clusters vacíos)
  - Propiedades de una partición: $c_i \cap c_j = \emptyset\ \forall i, j \in \{1, ..., k\}, i \neq j, \cup_{c_i \in C} c_i = \{1, ..., n\}$

- Para cada partición, calculamos su *infeasibility* como el número de restricciones incumplidas:

$$\textit{infeasibility}(C) = \sum_{i=0}^{|ML|} \mathbb{1}(h_c(\vec{ML_{[i,1]}} \neq h_c(\vec{ML_{[i,2]}})) + \sum_{i=0}^{|CL|} \mathbb{1}(h_c(\vec{CL_{[i,1]}} = h_c(\vec{CL_{[i,2]}}))$$

[1] No aparecen todos los elementos, sino los más imprescindibles. Seguimos la notación del seminario y del guion de la práctica.

II | Algoritmos implementados

1. Elementos comunes

A continuación, explicaremos brevemente los elementos comunes de todos los algoritmos. La implementación de esta práctica, hecha en Python, ha sido diseñada para reutilizar una clase `PAR` para cada algoritmo mediante el mecanismo de herencia. Por tanto, explicaremos en primer lugar las utilidades de `PAR`.

Atributos comunes

En primer lugar, tenemos los diversos *input* del problema:

- `datos`: es la matriz de datos, X.
- `clusters`: es una lista que mantiene una correspondencia uno a uno entre los datos y los clusters a los que son asignados (ver "partición" en la descripción del problema).
- `restr_mat`: restricciones en forma de matriz (en el formato de los ficheros).
- `restr_list`: restricciones en forma de lista de 3-uplas de forma $(i, j, r(i, j))$, donde $r(i, j) = 1$ si $\exists \vec{ML}_{[i,j]}$ y $r(i, j) = -1$ si $\exists \vec{CL}_{[i,j]}$.
- `centroides`: lista de centroides, uno para cada cluster correspondiente a su índice.

Cuando usemos en los diversos métodos variables que contengan alguno de los nombres anteriores, tendrán el mismo formato que los justo descritos. Por ejemplo, `new_centroides` es una lista de centroides que contiene en cada posición el centroide del cluster correspondiente a su índice.

Métodos comunes

En la implementación de `PAR` aparecen diversos algoritmos sencillos para obtener estructuras de datos útiles (como una lista de los índices de datos para cada cluster, por ejemplo), así como para cargar los ficheros de datos, etc. Dado que no tienen especial interés para la asignatura, no los describiremos en profundidad. En caso de que sea necesario aclarar qué hace cada uno, se expondrá para cada algoritmo en las siguientes páginas.

2. Algoritmo greedy COPKMN

Este algoritmo ha sido implementado en la clase `Greedy`. La idea principal es ir asignando cada dato a un cluster, de modo que dicha asignación sea la que tenga menor infeasibility posible y que se encuentre más cercana a su centroide. Lo describimos a continuación:

Algoritmo (Greedy COPKMN).

```
01  function greedy:
02      centroides ← generarCentroides()
03      rsi ← shuffle({1, ..., n})
04      hay_cambio ← True
05      while hay_cambio:
06          clusters_prev ← clusters
07          for i in rsi:
08              clusters[i] ← índice del cluster cuyo centroide está más cercano a x_i,
                              de entre los que producen el menor incremento en infeasibility
09          actualizarCentroides()
10          hay_cambio ← clusters_prev ≠ clusters
11      return clusters
```

Algunos comentarios para ciertas líneas del algoritmo:

02. La generación de la solución inicial se hace de manera aleatoria. Tras implementar la generación de centroides aleatoria en el espacio de los datos, lo modifiqué por tomar datos como centroides, es decir, aleatoriamente tomar como centroide uno de los datos, sucesivamente para cada cluster. De este modo tenemos particiones válidas en todo momento.

10. La forma de detectar el cambio es que la asignación de clusters haya cambiado.

3. Algoritmo de búsqueda local

Este algoritmo ha sido implementado en la clase `BusquedaLocal`. En resumidas cuentas, hacemos una búsqueda primero el mejor, con la siguiente función objetivo:

$$f = \bar{C} + \lambda \cdot infeasibility, \ \lambda = \frac{n}{|R|}$$

Algoritmo (Búsqueda local).

```
01  function busqueda_local(max_iters):
02      explorar_vecinos ← generarExplorarVecinos()
03      random_indices ← shuffle({0, ..., len(explorar_vecinos) – 1})
04      stop, restablecer ← False
04      f_prev ← f()
04      it ← 0
05      while not stop and it < max_iters:
06          stop ← True
07          for i in {0, ..., len(explorar_vecinos) – 1}:
08              v ← explorar_vecinos[i]
09              prev_clusters ← clusters
10              actualizarClusters(v)
11              if esClustersValido():
12                  it ← it + 1
13                  actualizarCentroides()
14                  f_vecino ← f()
15                  if f_vecino < f_prev:
16                      f_prev ← f_vecino
17                      explorar_vecinos ← generarExplorarVecinos()
18                      shuffle(random_indices)
19                      stop ← False
20                      break
21                  else:
22                      restablecer ← True
23              else:
24                  restablecer ← True
25              if restablecer:
26                  clusters ← prev_clusters
27                  actualizarCentroides()
28                  restablecer ← False
29  return clusters
```

Para la implementación de este algoritmo, hemos seguido las indicaciones del seminario. El vector S corresponde con el vector `clusters` (pues son las asignaciones de cada dato a un cluster) en cada iteración. Describimos el algoritmo a continuación.

Algunas consideraciones:

- La función de **generación de vecinos**, *generarExplorarVecinos*(), crea una lista de tuplas de la forma (i, j) donde i es el índice de un dato y j es el cluster al que se asignará ese dato.
- **Recorremos** los vecinos de forma aleatoria mediante el uso de *shuffle*(). Cada vez que vemos si el vecino minimiza f, debemos comprobar primero que el vecino es válido, para lo cual usamos la función *esClustersValido*() (que comprueba si el vector `clusters` representa una partición válida).
- La función f está implementada para calcularse como un método de la clase por motivos de eficiencia; es por eso por lo que para calcular la f del vecino modificamos la clase (habiendo almacenado previamente `clusters` para restablecerlo en caso de que el vecino minimice f). Análogamente ocurre con las funciones *actualizarClusters*(), *esClusterValido*() y *actualizarCentroides*().
- La sentencia de llamada de *actualizarCentroides*() en el código implementado difere un poco del pseudocódigo, ya que por motivos de eficiencia sólo actualizamos los centroides involucrados en el operador vecino, pues son los únicos que pueden cambiar.
- El booleano `stop` sirve para terminar prematuramente la ejecución:si al terminar un ciclo del *while* no se ha modificado a *False*, quiere decir que se han explorado todos los vecinos posibles y no hay mejoras. En este momento, la ejecución para.
- El booleano `restablecer` vale para reasignar `clusters` en caso de que no haya mejora. Para ello lo almacenamos, antes de hacer comprobaciones sobre un vecino, en `prev_clusters`.

III | Implementación

Para la implementación de esta práctica me he decantado por el uso del lenguaje Python. A pesar de que sea un poco más lento que lenguajes de más bajo nivel, su simplicidad y su funcionalidad me han hecho decidirme por éste. Además, el código es mucho más legible y funcional.

Para ejecutar la práctica es necesaria tener instalada la librería `numpy`. Para instalarla, puede usar `pip`, haciendo uso del `requirements.txt` proporcionado. También puede hacer uso de `virtualenv` para crear un entorno virtual para la ejecución de la práctica, o usar `Pipenv`. Algunos paquetes como Spyder ya tienen integrados estos paquetes.

Ejecución

Para ejecutar la práctica, introduzca el siguiente comando:

```
python main.py nombre_set porc_restricciones
          algoritmo [num_repeticiones] [semilla]
```

Inserte directamente `python main.py` para una explicación más detallada de los argumentos.

IV | Análisis de resultados

Con la semilla por defecto (la que aparece como `semilla_fija` en `main.py`), obtenemos los siguientes resultados de ejecución (los logs completos están en la carpeta `logs`).

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
Repetición 1		13	0.1891	0.3818		8	0.0241	1.3975		36	0.0041	14.4020
Repetición 2		13	0.1891	0.4088		8	0.0241	1.3953		36	0.0041	14.4132
Repetición 3		13	0.1891	0.3894		8	0.0241	1.4037		36	0.0041	14.5148
Repetición 4		13	0.1891	0.3887		8	0.0241	1.4122		36	0.0041	15.1210
Repetición 5		13	0.1891	0.4120		8	0.0241	1.4043		36	0.0041	15.1745
Media		13	0.1891	0.3961		8	0.0241	1.4026		36	0.0041	14.7251

Tabla 1. Greedy COPKMN con 10 % de restricciones

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
Repetición 1		0	0	0.3234		0	0.0086	1.2252		1	0.0187	9.0529
Repetición 2		0	0	0.3340		0	0.0086	1.2310		1	0.0187	9.0507
Repetición 3		0	0	0.3231		0	0.0086	1.2237		1	0.0187	9.1771
Repetición 4		0	0	0.3277		0	0.0086	1.2267		1	0.0187	9.2201
Repetición 5		0	0	0.3251		0	0.0086	1.2298		1	0.0187	9.2024
Media		0	0	0.3267		0	0.0086	1.2273		1	0.0187	9.1406

Tabla 2. Greedy COPKMN con 20 % de restricciones

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
Repetición 1	0.6096	24	0.3445	3.9671	0.2200	71	0.1770	17.5574	0.2462	576	0.0764	40.4818
Repetición 2	0.6096	24	0.3445	4.1536	0.2200	71	0.1770	17.8893		36	0.0764	41.6119
Repetición 3	0.6096	24	0.3445	4.2314	0.2200	71	0.1770	18.7569		36	0.0764	42.2173
Repetición 4	0.6096	24	0.3445	4.2675	0.2200	71	0.1770	17.9423		36	0.0764	41.8520
Repetición 5	0.6096	24	0.3445	4.4307	0.2200	71	0.1770	17.7403		36	0.0764	41.5649
Media	0.6096	24	0.3445	4.2101	0.2200	71	0.1770	17.9772		36	0.0764	41.5456

Tabla 3. Búsqueda local con 10 % de restricciones

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
Repetición 1	0.6429	52	0.3188	3.4448	0.2326	191	0.1783	20.2457	0.2146	930	0.0917	53.4925
Repetición 2	0.6429	52	0.3188	3.6081	0.2326	191	0.1783	20.1962	0.2146	930	0.0917	54.7689
Repetición 3	0.6429	52	0.3188	3.6237	0.2326	191	0.1783	19.7059	0.2146	930	0.0917	52.1164
Repetición 4	0.6429	52	0.3188	3.5326	0.2326	191	0.1783	20.6254	0.2146	930	0.0917	53.2953
Repetición 5	0.6429	52	0.3188	3.4264	0.2326	191	0.1783	20.5561	0.2146	930	0.0917	53.2005
Media	0.6429	52	0.3188	3.5271	0.2326	191	0.1783	20.2659	0.2146	930	0.0917	53.3747

Tabla 4. Búsqueda local con 20 % de restricciones

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
COPKM		13	0.1891	0.3961		8	0.0241	1.4026		36	0.0041	14.7251
BL	0.6096	24	0.3445	4.2101	0.2200	71	0.1770	17.9772		36	0.0764	41.5456

Tabla 5. Resultados globales con 10 % de restricciones

	zoo				glass				bupa			
	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t	A	T_{inf}	E_{dist}	t
COPKM		0	0	0.3267		0	0.0086	1.2273		1	0.0187	9.1406
BL	0.6429	52	0.3188	3.5271	0.2326	191	0.1783	20.2659	0.2146	930	0.0917	53.3747

Tabla 6. Resultados globales con 20 % de restricciones

Podemos ver que el algoritmo greedy desempeña sorprendentemente bien, con una convergencia más rápida que en el caso de la búsqueda local, y llegando a soluciones con muy poco error. Notemos que, por término general, los valores de *infeasibility* son menores para COPKM. Esto es claro teniendo en cuenta que estamos, por diseño, escogiendo los vecinos con menor *infeasibility*. Por otra parte, vemos cómo también el algoritmo de búsqueda local va reduciendo el valor de la función objetivo. Un experimento interesante es mostrar por pantalla estos valores, e iremos viendo cómo la función se va acercando a un f mínimo.

Además, vemos cómo conforme aumenta la complejidad del dataset el algoritmo tarda más en encontrar soluciones, y llegará en general a soluciones con un mayor error. Sin embargo, en el caso del algoritmo de búsqueda local esto puede resultar más evidente, dado que conforme vamos minimizando la función f más complicado será encontrar un vecino que consiga obtener un valor aún menor de la función objetivo, resultando en recorridos más largos en el vecindario, con el consiguiente gasto computacional de recalcular las desviaciones en cada paso.

Además, es interesante ver cómo la convergencia va modificándose en función del parámetro λ. Consideremos la ejecución del set *glass*, con un 10 % de restricciones (y la semilla por defecto), para los valores de $\lambda = \delta \frac{n}{|R|}$, con $\delta \in \{0.25, 0.5, 0.75, 0.1\}$:

	A	T_{inf}	E_{dist}	t
$\delta = 1$	0.2200	71	0.1872	17.4143
$\delta = 0.75$	0.2087	63	0.1869	20.9972
$\delta = 0.5$	0.2004	161	0.1633	19.9512
$\delta = 0.25$	0.1773	237	0.1499	20.4341

Tabla 6. Resultados en *glass* con 10 % de restricciones, variación de λ