Práctica 1.b:
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Agrupamiento con Restricciones.

3º CSI
Problema del Agrupamiento con Restricciones (PAR)

BL Greedy

David Erena Casado 26055001A davidерena@correo.ugr.es
Grupo 1 (Miercoles 17:30 – 19:30)

Indice:

Problema del Agrupamiento por Restricciones – PAR:

El agrupamiento o análisis de clusters o clustering en inglés, persigue la clasificación de objetos de acuerdo a posibles similitudes entre ellos.

El problema del PAR consiste en una generalización del agrupamiento clásico. Permite incorporar al proceso de agrupamiento un nuevo tipo de información: las restricciones. Dado un conjunto de datos X con n instancias, el problema consiste en encontrar una partición $C=\{c_1,\dots,c_{1k}\}$ del mismo que minimice la desviación general y cumpla con las restricciones de instancia existentes en el conjunto R. Dada una pareja de instancias, se establece una restricción de tipo Must-Link (ML) si deben pertenecer al mismo grupo y de tipo Cannot-Link (CL) si no pueden pertenecer al mismo grupo.

A partir de ahora, consideraremos n como el número de instancias/datos y k como el número de clústers que consideramos para el problema.

Aplicación de los algoritmos empleados al problema:

En cuanto a la forma de representar una solución, he decidido hacerlo de dos formas:

S1.- Un vector donde la posición i indica el cluster al que se asigna el dato i.

S2.- Un vector de vectores donde el vector i almacena los índices de los datos que pertenecen al cluster i.

Aunque a la hora de mostrar la solución uso S1.

La razón es que acceder a los datos de un cluster es una tarea que se va a realizar muchas veces a lo largo de la ejecución, y si solo trabajásemos con S1 tendríamos que recorrer el vector entero para poder guardar los datos pertenecientes al cluster, mientras que usando S2 no necesitamos esto ya que lo tenemos actualizado en el momento.
A su vez, S1 es necesaria ya que, aparte de para mostrar la solución, para operaciones lógicas (como ver si dos elementos pertenecen al mismo cluster) se complica con S2 (recorrer S2 cada vez que comprobemos algo), y con S1 es inmediato.

Se pierde un poco en espacio y un poco en velocidad al tener que tener ambos actualizados, pero con lo que ahorramos en las operaciones es una mejora.

_____ Funciones comunes: _____

calcularLambda

Se usa una vez al principio del programa para calcular cuánto penalizan las infracciones cometidas.
Se calcula dividiendo la mayor distancia entre dos datos del conjunto (siendo la distancia entre dos datos el módulo del vector resta de estos) entre el número de restricciones existentes.

calcularCentroide

Calcula el centroide asociado a un cluster, realizando la media aritmética de las características de los datos que pertenezcan a este.

vectorCentroides

Calcula los centroides de todos los clusters, usando la función calcularCentroide.

Ci :

Calcula la distancia intracluster, es decir, la media aritmética de distancias entre cada elemento de un cluster y su centroide (módulo de la resta de estos dos vectores).

C :

Calcula la media aritmética de los Ci's de cada cluster.

violaRestriccion :

Nos dice si una solución viola una restricción de la matriz de restricciones, es decir, si dos datos dados tenían una relación ML no están en el mismo cluster, o tenían una relación CL y comparten clúster.

infeasibility :

Devuelve el número de restricciones infringidas en una solución. Recorre el triángulo superior de la matriz (sin contar la diagonal principal) ya que es simétrica, y aplica violaRestriccion a cada posición i,j (que indicaría la restricción asociada a los datos i y j).

funcionObjetivo :

Calcula la bondad de una solución, dada por la expresión:

funcionObjetivo = C + (infeasibility * LAMBDA)

restaVector :

Resta dos vectores:

$$v-u=(v_1-u_1, v_2-u_2, \ldots, v_n-u_n)$$

moduloVector :

Calcula el módulo de un vector:

$$|v|=\sqrt{(v_1{}^2+v_2{}^2+\ldots+v_n{}^2)}$$

borrarPorValor :

Borra un elemento de un vector, buscando por valor hasta que alguno coincida con el dado. Para ello, recorre el vector desde el inicio y desde el final hasta llegar a la mitad de este, y en caso de encontrar el elemento lo borra y termina.

Búsqueda Local:

Partiendo de una solución aleatoria, exploramos su entorno, "viajando" a la primera solución mejor que encontremos, y volviendo a explorar, repitiendo esto hasta que exploremos el entorno de una solución sin encontrar una solución mejor o evaluamos la función objetivo 100.000 veces.

Necesitamos generar una solución aleatoria para empezar, explorar el entorno de esta y generar los vecinos.

_____ Generar solución aleatoria: _____

Realizaremos un bucle que nos generará una solución aleatoria en cada iteración de esta, generando n números aleatorios entre 0 y k-1, de forma que acabaremos con una solución al problema. Llevaremos una cuenta del número de componentes de cada cluster para saber si esta es factible o no. En caso de no ser factible, realizaremos otra iteración y generaremos otra solución.

_____ Exploración del entorno: _____

Representaremos un posible vecino como un par (i,j) que indicará un cambio del dato i al cluster j.
Para explorar el entorno, crearemos una lista de estos pares para no tener que guardar una gran cantidad de vectores, de los cuales puede que muchos sean inútiles, es decir, que siempre encontremos un vecino mejor mucho antes de que llegue su turno de ser evaluado.
A su vez, solo incluiremos pares que acaben generando vecinos en los que se haga un cambio y además nos lleve a una solución factible.

_____ Generación de vecino: _____

Como ya hemos hecho todas las comprobaciones al generar el entorno de exploración, no tendremos que hacerlas al generar un vecino. Por lo tanto, solo tenemos que realizar una copia de la solución actual cambiando el valor que nos indica el par para S1, y borrar en el vector del cluster e incluirlo en otro para S2.

Por tanto, el pseudocódigo de algoritmo basado en búsqueda local será el siguiente:

— Generamos una solución aleatoria y la evaluamos, guardando el valor como el mínimo y la solución como la mejor obtenida.

— Comenzamos el bucle de búsqueda local:

- Si no hemos generado el entorno virtual de la solución en la que nos encontramos (la mejor hasta el momento), lo generamos (de acuerdo a lo explicado en la exploración de entorno).

- Obtenemos un elemento del entorno (el último de la lista, en mi caso, pero como se baraja al generarse dicha lista, no tiene mucha relevancia).

- Generamos el vecino asociado a dicho elemento, y lo evaluamos.

- Si el valor obtenido es menor que el mínimo, guardamos este valor como el mínimo y la solución del que surge como la mejor solución obtenida) y volveríamos al principio del bucle.

- Si no lo es y ya no nos quedan elementos en el entorno, o hemos realizado 100000 evaluaciones en el bucle, salimos de este.

Algoritmo COPKM:

Antes de detallar el algoritmo, especificaré la función **infeasibilityLocal**, que será necesaria para este. InfeasibilityLocal nos dice el número de restricciones que se incumplirían si incluyésemos el dato i en un clúster j, sin llegar a incluirlo.

— Calculamos máximos y mínimos de cada característica para el conjunto de datos.

— Generamos centroides con números aleatorios para cada característica entre el máximo y mínimo de cada una.

— Bajaramos el conjunto de datos.

— Realizamos el bucle para encontrar la solución:
  — Para cada dato i:
    – Calculamos el n.º de restricciones incumplidas si incluyésemos un dato i en el cluster 0.
    – Para cada cluster j (menos el 0):
      - Calculamos infeasibilityLocal para i y j.
      - Si se incumpliesen menos restricciones con i en j que con el que consideremos el mínimo, guardamos j como el cluster seleccionado.
      - Si se inclumpliesen las mismas que con el mínimo, calculamos la distancia mínima entre i y el centroide de cada clúster a considerar, y seleccionamos el que menos dé.
    – Asignamos el dato i al clúster seleccionado.
  — Tras asignar todos los datos, generamos nuevos centroides según estas asignaciones.
  — Si la solución coincide con la obtenida en la iteración anterior, salimos del bucle. Si no, realizamos otra iteración.

He tenido problemas con este algoritmo debido a ciclos infinitos, en los que oscila entre muchas soluciones. Para solventarlo, opté por dos posibles medidas:

_____ N.º de iteraciones determinado _____

Al igual que hacemos en BL, pero aquí no es una solución ideal ya que puede que en la iteración 100000 acabemos en una solución que no sea muy buena, aunque solo sea ligeramente. Aún así, está implementada.

_____ COPKM_Set _____

Usar un set (por la función de búsqueda con eficiencia logarítmica) para almacenar los infeasibilities que vayan saliendo en cada iteración. Cuando obtengamos un infeasibility que ya hemos obtenido antes, el set tendrá x elementos, realizaremos x iteraciones más, y nos quedaremos con la solución que nos dé menos infeasibility de esas x iteraciones. De esta forma nos quedaremos con una solución indicativa de la calidad del algoritmo COPKM, aunque tal vez en tiempo no nos proporcione una toma realista.

Material considerado y manual de uso:

En cuanto al material usado en la implementación, se han usado las funciones de generación de números aleatorios de PRADO (random.h y random.cpp) y las siguientes librerías de C++ : iostream, fstream, string, vector, set, cmath, numeric, algorithm, limits.h, chrono, functional .

En cuanto al uso del código, este viene dividido en 5 carpetas: bin, obj, include, src, y data, que contienen los ejecutables, archivos objeto, archivos .h, archivos fuente y conjuntos de datos y restricciones, respectivamente.
Las acompaña un *makefile*, con lo cual, para compilar el código, solo necesita usar el comando *make* posicionándose en la terminal sobre la carpeta donde se encuentre.

Una vez compilado, el programa, que se encontrará en la carpeta bin, se ejecuta con los siguientes argumentos:

./bin/practica1 archivoDeDatos archivoDeRestricciones NumeroDeClusters Semilla VersionDeCOPKM (Opcional)

Donde VersionDeCOPKM puede valer nada o 0 para la versión normal, 1 para la versión con iteraciones (100 000 iteraciones) o 2 para usar COPKM_Set

---

Experimentos y análisis de resultados:

He realizado 5 ejecuciones para cada cada porcentaje de restricciones (10% y 20%) y conjunto de datos (iris, ecoli y rand).

Para conjuntos de datos con 10% de restricciones:

	BL											
	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0,669305	0	0,669305	0,0868223	21,3128	105	24,1113	3,56434	0,757315	0	0,757315	0,0720241
Ejecución 2	0,669305	0	0,669305	0,0631348	21,4885	66	23,2475	2,91538	0,757315	0	0,757315	0,0717528
Ejecución 3	0,669305	0	0,669305	0,104398	22,163	73	24,1086	2,45708	0,757315	0	0,757315	0,0697817
Ejecución 4	0,669305	0	0,669305	0,0873164	19,6101	129	23,0482	4,33384	0,757315	0	0,757315	0,0834925
Ejecución 5	0,669305	0	0,669305	0,0766736	21,2146	120	24,4129	3,53218	0,757315	0	0,757315	0,0725153
MEDIA	0,669305	0	0,669305	0,08366902	21,1578	98,6	23,7857	3,360564	0,757315	0	0,757315	0,07391328

HECHO CON COPKM_SET

	COPKM											
	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1,83761	181	2,9719	0,000766904	49,1308	535	63,3898	0,0623241	2,56833	290	4,64532	0,000749738
Ejecución 2	1,93655	64	2,33762	0,000792526	44,852	438	56,5258	0,040928	2,66312	121	3,52972	0,000623597
Ejecución 3	1,80363	105	2,46165	0,000595464	47,9822	468	60,4555	0,082068	2,7648	71	3,2733	0,000932705
Ejecución 4	1,90551	157	2,88939	0,000916775	47,3879	428	58,7952	0,0654642	2,63065	241	4,3567	0,000745016
Ejecución 5	1,88648	152	2,83903	0,000612448	46,8759	556	61,6947	0,0608112	2,76779	111	3,56278	0,000740328
MEDIA	1,873956	131,8	2,699918	0,0007368234	47,24576	485	60,1722	0,0623191	2,678938	166,8	3,873564	0,0007582768

	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	1,873956	131,8	2,699918	0,0007368234	47,24576	485	60,1722	0,0623191	2,678938	166,8	3,873564	0,0007582768
BL	0,669305	0	0,669305	0,08366902	21,1578	98,6	23,7857	3,360564	0,757315	0	0,757315	0,07391328

Semillas:
3,6,7,8,14

Semillas:
1,2,3, 4, 5

Semillas:
2,3,8,9,10

Para los conjuntos de datos con un 20% de restricciones:

	BL											
	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	0,669305	0	0,669305	0,101856	21,9128	176	24,2582	4,18624	0,757315	0	0,757315	0,0735221
Ejecución 2	0,669305	0	0,669305	0,120579	21,5689	246	24,8472	5,37933	0,757315	0	0,757315	0,0982977
Ejecución 3	0,669305	0	0,669305	0,117482	24,7698	162	26,9286	5,08431	0,757315	0	0,757315	0,0754964
Ejecución 4	0,669305	0	0,669305	0,15882	21,7778	169	24,03	4,80957	0,757315	0	0,757315	0,107138
Ejecución 5	0,669305	0	0,669305	0,0993551	21,9065	198	24,5451	3,04065	0,757315	0	0,757315	0,0797743
MEDIA	0,669305	0	0,669305	0,11961842	22,38716	190,2	24,92182	4,50002	0,757315	0	0,757315	0,0868457

HECHO CON COPKM_SET

	COPKM											
	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
Ejecución 1	1,92394	112	2,27472	0,000466023	46,9642	712	56,4525	0,0202916	2,71111	174	3,33393	0,000712264
Ejecución 2	1,93173	101	2,24806	0,000688274	48,3969	522	55,3532	0,178183	2,78079	61	2,99913	0,000887122
Ejecución 3	1,90519	76	2,14322	0,00068974	47,5843	511	54,394	0,222077	2,76827	181	3,41615	0,000948706
Ejecución 4	1,86544	127	2,2632	0,000677153	47,7747	553	55,1441	0,0661433	2,76716	231	3,59401	0,000899807
Ejecución 5	1,94365	382	3,14007	0,000697033	46,5483	423	52,1853	0,0726913	2,79663	126	3,24763	0,000886166
MEDIA	1,91399	159,6	2,413854	0,0006436446	47,45368	544,2	54,70582	0,11187724	2,764792	154,6	3,31817	0,000866813

	IRIS				ECOLI				RAND			
	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T	Tasa_C	Tasa_inf	Agr.	T
COPKM	1,91399	159,6	2,413854	0,0006436446	47,45368	544,2	54,70582	0,11187724	2,764792	154,6	3,31817	0,000866813
BL	0,669305	0	0,669305	0,11961842	22,38716	190,2	24,92182	4,50002	0,757315	0	0,757315	0,0868457

Semillas: 2,3,4,5,11

Semillas: 1,2,3,4,5

Semillas: 1,3,7,10,11

A los tiempos que se han obtenido en ecoli con COPKM no les prestaré mucha atención, independientemente de si tienen sentido o no, ya que los realicé con el método COPKM_Set explicado anteriormente porque me ciclaba con cualquier semilla que probase (Probé con semillas desde 1 hasta 14, y algunas esporádicamente mayor a 14).

En los otros conjuntos de datos, se observa que los tiempos en COPKM son claramente menores a los de BL (Los tiempos de BL son, de media, un 12326% mayor aprox. a los de COPKM, es decir, un 12326% más lento que este).

Sin embargo, en cuanto a resultados, BL obtiene resultados muchísimo mejores que COPKM (en este caso, COPKM obtiene soluciones un 234% peores que BL de media). Incluso en infeasibility, que es el principal objetivo de COPKM, BL consigue mejores resultados.

Estos resultados tienen sentido, ya que, en lo referente al tiempo de ejecución, COPKM trabaja con funciones mucho menos costosas que BL, como son infeasibilityLocal y, en caso de empate, calcular dos distancias a un vector. BL, al trabajar directamente con la función objetivo, en una única iteración va a realizar muchas más operaciones (distancias intracluster e infeasibility total).

Sin embargo, en consecuencia de esto, BL es más probable de obtener una buena solución en comparación con COPKM.
COPKM, al tener un comportamiento greedy, elige la mejor opción en un instante dado para una solución en construcción. Esto, como es usual en algoritmos con esta filosofía, prioriza beneficios a corto plazo e ignora el total de la solución, lo cual nos va a llevar a peores soluciones muy probablemente.
BL trabaja con la solución en sí, es decir, que todos los cambios que se hagan, será porque mejoran la calidad de la solución en su totalidad y no porque es lo inmediatamente mejor en una solución parcial.

Finalmente, en cuanto a la robustez de los algoritmos, es decir, la poca disparidad al producir soluciones, diría que ambos son robustos.
COPKM es bastante dispar para las infeasibilities que obtiene (ecoli con 20% de restricciones, por ejemplo) pero los valores de la función objetivo son bastante constantes (independientemente de la calidad de estas), con lo cual no le doy mucha importancia.
BL tiene un comportamiento similar, pero con mejor calidad en las soluciones. En Iris y Rand consigue soluciones óptimas, pero en Ecoli, donde no lo consigue, tiene infeasibilities algo dispares pero un valor de la función objetivo bastante constante.
