Práctica 3: Algoritmos genéticos para el problema de clustering. 3º A de grado en Informática.

Algoritmos tratados: Algoritmo genético generacional (AGG) y algoritmo genético estacionario (AGE).

Realizado por David Téllez Rodríguez. 75907697-S,roka903@correo.ugr.es. Grupo de prácticas 2 en horario de Miércoles 17:00-19:00.

Contenido

1. Descripción del problema.

La resolución del problema de clustering persigue la agrupación de una serie de datos de entrada en clústeres o grupos según un criterio específico. Es un problema de complejidad alta, NP-Duro (NP-Hard). Para nuestro caso concreto se pretende minimizar la distancia existente entre datos similares; midiendo la similitud de varios datos entre sí como la distancia euclídea que existe entre ellos.

Este es un problema de aprendizaje no supervisado, lo que implica que al algoritmo le ofrecemos sólo los datos de entrada y sus características y el número de clusters en el que se deben agrupar, encargándose el propio algoritmo de seleccionar la mejor agrupación de datos.

Para conseguir este propósito debemos valernos de un elemento importante:

- Centroide: Representa el punto medio de un clúster, obtenido como la media de todos los patrones que se encuentran en el mismo.

De este modo, el problema se reduce a obtener la menor distancia euclídea de cada centroide de cada clúster con los datos que conforman el mismo, esto implicará que los datos están más próximos entre sí, lo que obedece al resultado deseado: Una agrupación de los datos que sean más cercanos (o similares).

Podemos formular esta misma idea ayudándonos de la siguiente expresión:

Minimizar **J**(W, Z, X) = $\sum_{i=1}^{m}\sum_{j=1}^{k} w_{ij} \cdot \left|\left|X^i - Z^j\right|\right|^2$, donde:

X es el conjunto de patrones de tamaño *m* (tendremos m patrones). Un patrón será lo que hemos llamado anteriormente *dato*, X^i. X por tanto será el conjunto de datos a agrupar.

W es la matriz de pertenencia de patrones a clústeres. Un elemento w_{ij} de esta matriz indica que el patrón *i* pertenece al clúster *j*.

Z es el conjunto de centroides de clústeres, de tamaño *k*. Un elemento Z^j será el centroide del clúster *j*.

Para completar la formulación del problema, habremos de respetar las siguientes restricciones:

- $C_i \neq \emptyset$, es decir, no puede haber clusters que no tengan ningún elemento.
- $C_i \cap C_j = \emptyset$ para todo $i \neq j$, es decir, un patrón no puede estar en dos clústeres distintos.
- $\cup_{i=1}^{k} C_i = X$, donde X representa como hemos visto, el conjunto de todos los patrones del problema.

2. Descripción en pseudocódigo del procedimiento de búsqueda para cada algoritmo.

- Representación de la solución.

Para este algoritmo se ha representado la población como un array de permutaciones de longitud igual al número de patrones. Cada permutación permite construir una solución válida que puede ser evaluada, éstas son a su vez un array de longitud igual al número de clústeres, donde para cada posición del mismo tendremos otro array indicando los patrones que pertenecen a un clúster en concreto. Es la representación usada en las prácticas anteriores.

Cada elemento de cada cromosoma contendrá un patrón, de forma que los *k* primeros patrones se tomarán como medoides. Estos medoides se utilizan en la construcción de una solución válida por cromosoma, usando para ello la estrategia del clúster más cercano.

- Pseudocódigo de la función objetivo.

Es la función objetivo usada hasta ahora, ya que cuando evaluamos una población convertimos los cromosomas en fenotipos que se corresponden con los usados hasta ahora:

Para cada clúster *k* **hacer**

Para cada elemento *i* de $solucion_k$ **hacer**

resultado = resultado + distanciaEuclidea($patrones_i$, $centroides_k$)

devolver resultado

Donde distanciaEuclidea devuelve la distancia euclídea entre dos patrones.

- Pseudocódigo de generación de soluciones aleatorias.

Como se especifica en la práctica, la población inicial de cromosomas se genera aleatoriamente para ambos algoritmos genéticos, siguiendo este proceso:

Para *i=0* hasta tamañoPoblacion **hacer** {

Para cada patrón *j* **hacer** {

$poblacion_i$ = j

}

aleatorizar($poblacion_i$)

}

Donde aleatorizar es una función que acepta un array e intercambia de posición sus elementos siguiendo una distribución pseudo-aleatoria. (En la implementación se corresponde con random_shuffle de la librería algorithm en C++).

- Pseudocódigo de los mecanismos de selección para los dos algoritmos.

- Algoritmo genético generacional.

En este algoritmo se ha propuesto un método de selección basado en un torneo binario que se ejecuta para todos los miembros de la población (todos los cromosomas). En cada iteración elegiremos dos padres aleatoriamente y sólo seleccionamos el mejor de ellos:

```
Para i=0 hasta tamañoPoblacion hacer {
    padre1 = enteroAleatorio (0, tamañoPoblacion-1)
    padre2 = enteroAleatorio (0, tamañoPoblacion-1)

    Si valoracion_padre1 < valoracion_padre2 entonces {
        padres_i = padre1
    }
    Sino {
        padres_i = padre2
    }

}
```

Donde padres es un array que rellenaremos con todos los padres seleccionados para luego cruzarlos. El array valoracion contiene los valores heurísticos de todos los cromosomas, calculados en la fase de evaluación anterior. La función enteroAleatorio genera un entero aleatoriamente entre el extremo inferior del intervalo y el extremo superior (incluyendo ambos).

- Algoritmo genético estacionario.

Para este algoritmo se cambia la manera de seleccionar a los padres; esta vez sólo elegimos dos padres, aunque usaremos el mismo torneo binario:

```
Para i=0 hasta 2 hacer {
    padre1 = enteroAleatorio (0, tamañoPoblacion-1)
    padre2 = enteroAleatorio (0, tamañoPoblacion-1)

    Si valoracion_padre1 < valoracion_padre2 entonces {
        padres_i = padre1
    }
```

Sino {

$padres_i$ = padre2

}

}

- Pseudocódigo de operadores genéticos utilizados.

Para ambos algoritmos se han empleado los mismos operadores genéticos: Cruce OX basado en representaciones de orden y mutación usando intercambio aleatorio.

- Operador de cruce.

Primero obtenemos la región de corte que usaremos para copiar la parte del primer padre al hijo. Una vez hecha la copia vamos copiando genes del segundo padre al hijo de modo que sólo asignaremos valores al hijo que todavía no tenga asignado, de este modo nos aseguramos que obtenemos un hijo válido del proceso de cruce.

procedimiento cruzarPadres (poblacion, hijo, padre1, padre2, base, tope) {

numAsignaciones = 0, tamaño = tamaño(hijo)

iteradorPadre1 = obtenerIterador ($poblacion_{padre1}$+base)

iteradorPadre2 = obtenerIterador ($poblacion_{padre2}$)

(1) Mientras numAsignaciones < tope-base **hacer** {

$hijo_{numAsignaciones+base}$ = contenido (iteradorPadre1)

iteradorPadre1 = iteradorPadre+1

numAsignaciones = numAsignaciones+1

}

contador = tope+1

(2) Mientras numAsignaciones < tamaño($poblacion_{padre1}$-1) {

```
        Si (!encontrar(hijo, base, tope, contenido (iteradorPadre2)) entonces {
                Si (iteradorPadre2 != obtenerIterador (poblacion_padre2+tamaño(poblacion_padre2))
                entonces {
                        hijo_contador = contenido (iteradorPadre2)
                }
                contador = contador+1 % tamaño
                numAsignaciones++
        }
        Si (iteradorPadre2 == obtenerIterador (poblacionpadre2+tamaño(poblacionpadre2))
        entonces {
                iteradorPadre2 = obtenerIterador(poblacion_padre2)
        }

        Sino {
                iteradorPadre2 = iteradorPadre2+1
        }

    }
        devolver hijo
}
```

Donde obtenerIterador devuelve un iterador al inicio del array que se le pasa como parámetro (en C++ es el método begin() de la clase vector). Se le pueden sumar enteros para obtener iteradores a otros elementos del array (tal y como ocurre en C++). En este pseudocódigo el hijo se devuelve, pero en el programa se pasa el hijo por punteros y no se devuelve nada (más eficiente). La función encontrar devuelve verdadero si para el array que recibe como primer argumento encuentra el valor que se le pasa como último argumento entre el intervalo que se le pasa como segundo y tercer argumento.

El bloque (1) se encarga de copiar el intervalo definido por base, tope al hijo, mientras que el bloque (2) copia los elementos de padre2 al hijo, según el orden en el que aparecen en padre2 y sólo copia aquellos que no están ya en el hijo. Como

podemos encontrarnos elementos repetidos porque los haya copiado de padre1, sólo debemos preocuparnos del intervalo base, tope.

- Operador de mutación.

Este operador consiste en elegir un número de cromosomas a mutar a priori (se aumenta la eficiencia del programa al no generar tantos números aleatorios), para después elegir un cromosoma y un par de genes aleatoriamente que intercambiaremos en dicho cromosoma.

genesMutar = 0.01*tamañoPoblacion*numPatrones

Para *i=0* hasta genesMutar **hacer** {

cromosomaMutar = enteroAleatorio (0, tamañoPoblacion-1)

genMutar = enteroAleatorio (0, tamañoPoblacion-1)

patronIntercambio = enteroAleatorio (0, tamañoPoblacion-1)

intercambio ($poblacion_{cromosomaMutar, genMutar}$, $poblacion_{cromosomaMutar, patronIntercambio}$)

}

3. Descripción en pseudocódigo del mecanismo de evolución y reemplazamiento.

- Pseudocódigo del mecanismo de evolución.

El ciclo de evolución es a grandes rasgos igual para los dos algoritmos genéticos, sin embargo cambiamos algunos parámetros como probabilidadCruce (0.7 en generacional y 1 en estacionario) y la longitud de algunos arrays (padres, hijos). En el estacionario sólo cruzamos dos padres y el mecanismo de reemplazamiento es distinto: Tendremos que calcular la valoración de los hijos para enfrentarlos con los dos peores cromosomas de la población actual.

```
maxSoluciones = 0
tamañoPoblacion = 50
parejasCombinar = probabilidadCruce*tamañoPoblacion
minimoGlobal = ∞
genesMutar = o.o1*tamañoPoblacion*numPatrones

inicializarPoblacion(poblacion)
evaluarPoblacion (poblacion, centroides, patrones, elite)
maxSoluciones = maxSoluciones+50
mientras maxSoluciones < 20000 hacer{
        seleccionTorneo (valoracion, tamañoPoblacion, padres)
        cruzarPadres (hijos, padres, parejasCombinar)
        mutacion (genesMutar, poblacion)
        reemplazamiento (poblacion, hijos, poblacion)
        elitismo (elite)
        evaluarPoblacion (poblacion, patrones, elite)
        maxSoluciones = maxSoluciones+50
        actualizarMejorSolucion (tamañoPoblacion, minimoGlobal,
mejorSolucion, valoracion)
}

devolver mejorSolucion
```

La función de elitismo consiste en sobrescribir el último cromosoma con el mejor de la población (elite) en el algoritmo generacional, en el estacionario esta función no es necesaria.

- Pseudocódigo del mecanismo de reemplazamiento.

Como comentamos en el apartado anterior, los mecanismos de reemplazamiento son distintos en los algoritmos:

- Algoritmo genético generacional.

En este caso, se ha de reemplazar toda la población actual con los hijos obtenidos del cruce, pero como el número de hijos no es igual al número de cromosomas en la población anterior (una vez realizado el reemplazamiento), rellenamos lo que nos resta con algunos cromosomas de la población anterior. Para facilitar el proceso, lo que hacemos es copiar completamente la población actual en el array de hijos, de modo que sólo modificaremos aquellos cromosomas para los que haya habido un cruce de padres, dejándonos en el resto algunos cromosomas de la población. Una vez realizado el cruce, el reemplazamiento consiste simplemente en copiar el array de hijos al array población actual:

```
Para i=0 hasta tamañoPoblacion hacer{
        hijos_i = poblacion_i
}

cruzarPadres (hijos, padres, parejasCombinar)
mutacion (genesMutar, poblacion)

// Reemplazamiento de la población.
Para i=0 hasta tamañoPoblacion hacer{
        poblacion_i = hijos_i
}
```

- Algoritmo genético estacionario.

En este caso no es necesario repetir todo el proceso anterior, pues sólo reemplazamos los dos peores cromosomas (si los hijos generados son mejores que ellos).

Para *i=0* hasta 2 **hacer** {

peorCromosoma = seleccionarPeor (valoracion, tamañoPoblacion)

valorHijo = calcularValorHeuristico ($hijos_i$, numClusters, numPatrones, dimensionPatrones, patrones)

Si (valorHijo < $valoracion_{peorCromosoma}$) **entonces** {

$poblacion_{peorCromosoma} = hijos_i$

}

}

La función seleccionarPeor devuelve el cromosoma con peor valoración heurística, mientras que la función calcularValorHeuristico devuelve el valor heurístico de un solo cromosoma, en este caso un hijo, siguiendo un proceso parecido al de evaluar una población, pero sólo para un cromosoma.

4. Descripción de KMM.

Este será el algoritmo que utilizaremos para comprar los resultados obtenidos con los otros tres que se han implementado.

En este caso, inicialmente, generamos una solución aleatoria. K-medias funciona seleccionando los patrones más distantes entre sí, para luego iniciar un recorrido secuencial por la estructura de datos en la que tengamos la información sobre los patrones y asignar cada uno de ellos al clúster cuyo centroide es más próximo al mismo. Una vez acabado el proceso, se recalculan los valores de los centroides y se vuelve a iterar el proceso hasta que los centroides se mantengan estables, es decir, no se realizan nuevas reasignaciones a clústeres.

Para la práctica se considera la versión multiarranque de este algoritmo, generando 25 soluciones aleatorias para luego optimizarlas usando el proceso descrito arriba. Al final del proceso se devuelve la mejor solución encontrada.

Es una técnica muy recomendable cuando tenemos una gran cantidad de datos de entrada

Debido a que es un algoritmo basado en heurísticas, no asegura llegar a un óptimo global, aunque generalmente converge a una buena solución, como veremos más tarde a la hora de analizar los tiempos de ejecución de los algoritmos.

5. Procedimiento seguido para desarrollar la práctica.

Para desarrollar esta práctica se ha seguido el mismo proceso que en las demás. Aunque teníamos disponible una implementación de algoritmo genético en la plataforma DECSAI he optado por implementar yo mismo los algoritmos. Para ello me he apoyado en el temario de la asignatura (tema 6 parte 1, disponible en DECSAI), el seminario 3 para clustering también disponible en la plataforma. Adicionalmente también se ha consultado la página http://www.cpluplus.com como referencia de programación al lenguaje utilizado: C++.

6. Presentación y evaluación de los resultados.

- Conjuntos de datos.

Para evaluar la bondad de los algoritmos construidos en la práctica contamos con tres conjuntos de datos, Aggregation, Wdbc y Yeast. El primero se compone de 788 patrones de dimensión 2 y que deberemos agrupar en 7 clústeres. Para Wdbc tendremos 569 patrones de dimensión 31 agrupados en 2 clústeres y finalmente, para Yeast nos encontramos con 1484 patrones de dimensión 8 a agrupar en 10 clústeres. Por supuesto, cada algoritmo necesitará tanto estos conjuntos de datos como la estructura de los mismos (número de patrones y clústeres y dimensión de patrones).

- Parámetros de ejecuciones por algoritmos.

- **AGG:** La población será de tamaño 50, dispondremos de 50 cromosomas en cada población. Tenemos una probabilidad de cruce de 0.7, con la que calculamos el número de parejas a cruzar como 0.7*50 = 35. La probabilidad de mutar un gen será de 0.01, con la que obtenemos también el número de genes a mutar: 0.01*50*(número de patrones). El algoritmo parará devolviendo la mejor solución cuando alcancemos 20000 evaluaciones de la función objetivo.

- **AGE:** Comparte muchos parámetros con el algoritmo genético generacional, como tamaño de población, probabilidad de mutación y número total de evaluaciones. En este caso, la probabilidad de cruce es de 1, con lo que siempre cruzamos los dos padres seleccionados.

Las semillas utilizadas han sido las mismas para todas las ejecuciones de los algoritmos, las recomendadas en el guion de la práctica:

Ejecución	Semilla
1	12345678
2	23456781
3	34567812
4	45678123
5	56781234

AGG	Wdbc		Aggregation		Yeast	
	J	Tiempo	*J*	Tiempo	*J*	Tiempo
Ejecución 1	415131000000000000,00	29,79	9801,67	19,76	45,93	80,34
Ejecución 2	403778000000000000,00	29,84	9799,56	19,19	45,99	81,79
Ejecución 3	425146000000000000,00	29,92	9763,94	20,07	45,75	83,53
Ejecución 4	415229000000000000,00	29,77	9856,95	20,20	45,67	81,53
Ejecución 5	421336000000000000,00	29,78	9839,59	19,49	46,26	81,02
Mejor	425146000000000000,00	29,77	9856,95	19,19	46,26	81,02
Peor	403778000000000000,00	29,92	9763,94	20,20	45,67	83,53
Media	416124000000000000,00	29,82	9812,34	19,74	45,92	81,64
Desv. típica	8105125199526530,00	0,06	36,58	0,41	0,23	1,19

Estos resultados nos ofrecen una visión global del comportamiento del algoritmo genético con reemplazamiento generacional. Podemos observar en la columna de resultados que para el conjunto Wdbc obtenemos unos valores muy buenos, que ningún algoritmo de los vistos hasta ahora ha podido mejorar. La desviación típica para este conjunto de datos es bastante alta, lo que indica que no es demasiado robusto con los datos de entrada que le proporciona, depende mucho de la solución inicial para dar con ciertos valores, los cuales también varían en cuantía debido a la misma razón. Para Aggregation obtenemos una media muy buena, por debajo de casi todos los algoritmos vistos, lo que indica un comportamiento muy bueno; aspecto que se reafirma si vemos su desviación típica (36,58). Para Yeast, como suele ser costumbre al tratarse de un conjunto de datos complicado, los

resultados se asemejan más a otros algoritmos vistos en prácticas anteriores, obteniendo de igual manera buenos resultados en un tiempo que quizá no es tan bueno. La desviación típica para los resultados de Yeast constatan un buen comportamiento en lo que a predictibilidad en resultados se refiere.

AGE	Wdbc		Aggregation		Yeast	
	J	Tiempo	*J*	Tiempo	*J*	Tiempo
Ejecución 1	455314000000000000,00	25,22	9891,16	8,79	46,24	46,12
Ejecución 2	445282000000000000,00	25,19	9885,80	8,81	46,41	45,34
Ejecución 3	443228000000000000,00	25,05	9906,33	8,68	45,61	45,24
Ejecución 4	449634000000000000,00	25,04	9860,63	8,72	46,37	45,57
Ejecución 5	449984000000000000,00	25,15	9906,00	8,71	46,09	45,06
Mejor	455314000000000000,00	25,04	9906,33	8,68	46,41	45,06
Peor	443228000000000000,00	25,22	9860,63	8,81	45,61	46,12
Media	448688400000000000,00	25,13	9889,98	8,74	46,14	45,47
Desv. típica	4687505818663050,00	0,08	18,74	0,06	0,32	0,41

Este otro algoritmo sigue un enfoque de reemplazamiento distinto y vamos a notar la diferencia tanto en tiempo como en calidad de resultados. Si pasamos por la columna Wdbc encontramos unos resultados que no son tan buenos como el anterior algoritmo o como otros que también obtienen buenos valores, como GRASP o ILS. Tampoco destaca demasiado en tiempo de ejecución, obteniendo unos valores parecidos a su predecesor, aunque en lo que sí ha mejorado es en robustez tanto en calidad de resultados como en tiempo de ejecución, sin ser demasiado brillantes. Para el conjunto de datos Aggregation la cosa cambia totalmente: Este algoritmo ha mejorado en todos los aspectos al generacional, obteniendo mejores resultados (aunque por unas pocas decenas) y sobretodo ejecutándose en aproximadamente la mitad de tiempo. Las desviaciones típicas devuelven valores muy buenos, asegurando el buen funcionamiento del algoritmo con este conjunto de datos. Finalmente, para Yeast, obtenemos unos valores muy buenos, un poco peores que el algoritmo anterior pero que bien pueden justificar su pequeño empeoramiento con un tiempo de ejecución que reduce a la mitad el tiempo de ejecución del generacional. En este caso y pese a ser un conjunto de datos difícil para los algoritmos demuestra una robustez y predictibilidad envidiables.

K-medias	Wdbc		Aggregation		Yeast	
	J	Tiempo	*J*	Tiempo	*J*	Tiempo
Ejecución 1	461344550688114000,00	0,97	10996,76	1,41	45,25	22,62
Ejecución 2	461344550688114000,00	0,96	10996,76	1,55	45,25	21,96
Ejecución 3	461344550688114000,00	0,96	10996,76	1,46	45,25	22,67
Ejecución 4	461344550688114000,00	0,97	10996,76	1,45	45,25	22,31
Ejecución 5	461344550688114000,00	0,96	10996,76	1,46	45,25	22,12
Mejor	461344550688114000,00	0,96	10996,76	1,41	45,25	21,96
Peor	461344550688114000,00	0,97	10996,76	1,55	45,25	22,67
Media	461344550688114000,00	0,96	10996,76	1,47	45,25	22,34
Desv. típica	0,00	0,43	0,00	0,05	0,00	0,31

Este algoritmo es con el que debemos comparar los resultados obtenidos por las otras dos implementaciones. Como podemos apreciar, los resultados que devuelve son muy buenos, ya que está pensado originalmente para resolver el problema de clustering, con un método intuitivo y muy extendido. Este es el algoritmo más robusto de todos los estudiados (su desviación típica para todos los conjuntos de datos está muy próxima al 0 absoluto), lo que lo hace muy confiable. Los resultados que nos ofrece, aunque de muy buena calidad, son superados por los dos algoritmos genéticos para Wdbc y Aggregation, aunque como siempre suele ocurrir en este tipo de algoritmos, lo que perdemos en calidad de la solución lo ganamos en tiempo de ejecución, ofreciendo los mejores tiempos para todos los conjuntos de datos. Cabe reseñar que para el conjunto Yeast, se aumenta el tiempo de ejecución hasta llegar a los 22 segundos de media. Este último resultado puede deberse a que los patrones este conjunto de datos sufren pequeñas relocalizaciones un gran número de veces, haciendo que el tiempo de ejecución se alargue rozando la condición de parada.

Resultados globales	Wdbc		Aggregation		Yeast	
	J	Tiempo	*J*	Tiempo	*J*	Tiempo
AGG (med)	416124000000000000,00	29,82	9812,34	19,74	45,92	81,64
AGG (desv)	8105125199526530,00	0,06	36,58	0,41	0,23	1,19
AGE (med)	448688400000000000,00	25,13	9889,98	8,74	46,14	45,47
AGE (desv)	4687505818663050,00	0,08	18,74	0,06	0,32	0,41
KM (med)	461344550688114000,00	0,96	10996,76	1,47	45,25	22,34
KM (desv)	0,00	0,43	0,00	0,05	0,00	0,31

Como comparativa final podemos decir que por supuesto KM nos da una robustez que no podemos encontrar en ninguno de los algoritmos de la práctica, con el mejor tiempo de ejecución disponible. Sin embargo, AG nos devuelve mejores

soluciones en 2 de 3 conjuntos de datos, ofreciendo una mejora considerable en la calidad de soluciones. Si diéramos un límite de ejecución más alto para algoritmos genéticos obtendríamos mejores soluciones sin incrementar en demasía el tiempo de ejecución ya estudiado. En general, elegiría algoritmos genéticos, por su alto grado de adaptación a los problemas, mientras que K-Medias ha sido diseñado específicamente para el problema de clustering. En la elección realizada perderíamos en tiempo de ejecución, pero al no ser tan grande el tiempo que invierten los algoritmos genéticos podemos absorber ese aumento en pos de obtener buenas soluciones para la mayoría de problemas existentes. Si se tratara de elegir un algoritmo de los vistos en esta práctica para clustering, sin duda me decantaría por KM, pues algoritmos genéticos mejoran las soluciones pero no en un grado tan alto como para pasar por alto su pérdida de robustez y el aumento en el tiempo de ejecución que sufren.

7. Bibliografía.

- http://pendientedemigracion.ucm.es/info/socivmyt/paginas/D_departamento/materiales/analisis_datosyMultivariable/21conglk_SPSS.pdf
- http://delendaestcarthago.com/lareinaroja/wp-content/uploads/2010/11/algoritmos-geneticos.jpg