

Práctica 4 Metaheurísticas.
Optimización basada en colonias de hormigas
para el problema de la selección de características

Jacinto Carrasco Castillo
N.I.F. 32056356-Z
jacintocc@correo.ugr.es

3 de junio de 2016

Curso 2015-2016
Problema de Selección de Características.
Grupo de prácticas: Viernes 17:30-19:30
Quinto curso del Doble Grado en Ingeniería Informática y Matemáticas.

Algoritmos considerados:

1. Algoritmo de Sistema de Colonias de Hormigas
2. Algoritmo Sistema de Hormigas Max-Mín

Índice

1. Descripción del problema	3
2. Descripción de la aplicación de los algoritmos	4
2.1. Representación de soluciones	4
2.2. Función objetivo	4
2.3. Operadores comunes	6
2.3.1. Operador de cruce	6
2.3.2. Operador de mutación	7
2.3.3. Torneo	8
3. Estructura del método de búsqueda	9
3.1. Algoritmo Genético Generacional	10
3.1.1. Operador de selección	10
3.1.2. Operador de reemplazamiento	10
3.2. Algoritmo Genético Estacionario	11
3.2.1. Operador de selección	11
3.2.2. Operador de reemplazamiento	11
4. Algoritmo de comparación	11
5. Procedimiento para desarrollar la práctica	12
5.1. Ejecución del programa	13
6. Experimentos y análisis de resultados	14
6.1. Descripción de los casos	14
6.2. Resultados	14
6.2.1. KNN	14
6.2.2. SFS	15
6.2.3. Algoritmo basado en colonia de hormigas	15
6.2.4. Algoritmo basado en hormigas Max-Mín	16
6.2.5. Comparación	16
6.3. Análisis de los resultados	17
6.3.1. Tasa In	17
6.3.2. Tasa Out	18
6.3.3. Tasa reducción	19
6.3.4. Tiempos	19
7. Bibliografía	20

1. Descripción del problema

El problema que nos ocupa es un problema de clasificación. Partimos de una muestra de los objetos que queremos clasificar y su etiqueta, es decir, la clase a la que pertenece y pretendemos, en base a esta muestra, poder clasificar nuevas instancias que nos lleguen. La clasificación se realizará en base a una serie de características, que nos permitan determinar si un individuo pertenece a un grupo u otro. Por tanto, tendremos individuos de una población Ω representados como un vector de características: $\omega \in \Omega; \omega = (x_1(\omega), \dots, x_n(\omega))$, donde ω es un individuo de la población y $x_i, i = 1, \dots, n$ son las n características sobre las que se tiene información. Buscamos $f : \Omega \rightarrow C = \{C_1, \dots, C_M\}$, donde $C = \{C_1, \dots, C_M\}$ es el conjunto de clases a las que podemos asignar los objetos.

El problema de clasificación está relacionado con la separabilidad de las clases en el sentido de que existirá la función f anteriormente mencionada siempre que las clases sean separables, es decir, siempre que un individuo con unas mismas características pertenezcan a una misma clase. Sin embargo, si se diese que dos individuos $\omega_1, \omega_2 \in \Omega$, $(x_1(\omega_1), \dots, x_n(\omega_1)) = (x_1(\omega_2), \dots, x_n(\omega_2))$ y sin embargo $f(\omega_1) \neq f(\omega_2)$, no podrá existir f . En todo caso, querríamos obtener la mayor tasa de acierto posible.

Por tanto, tratamos, en base a unos datos, hallar la mejor f posible. De esto trata el aprendizaje supervisado: Se conocen instancias de los datos y las clases a las que pertenecen. Usaremos como técnica de aprendizaje supervisado la técnica estadística conocida como k vecinos más cercanos. Se trata de buscar los k vecinos más cercanos y asignar al objeto la clase que predomine de entre los vecinos. En caso de empate, se seleccionará la clase con más votos más cercana.

Pero no nos quedamos en el problema de clasificación, sino que buscamos reducir el número de características. Con esto pretendemos seleccionar las características que nos den un mejor resultado (por ser las más influyentes a la hora de decidir la categoría). Usaremos los datos de entrenamiento haciendo pruebas mediante diferentes metaheurísticas hasta obtener la mejor selección que seamos capaces de encontrar.

El interés en realizar la selección de características reside en que se aumentará la eficiencia, al requerir menos tiempo para construir el clasificador, y que se mejoran los resultados al descartar las características menos influyentes y que sólo aportan ruido. Esto hace también que se reduzcan los costes de mantenimiento y se aumente la interpretabilidad de los datos.

Las funciones de evaluación pueden estar basadas en la consistencia, en la Teoría de la Información, en la distancia o en el rendimiento de clasificadores. Nosotros usaremos el rendimiento promedio de un clasificador 3 – NN.

2. Descripción de la aplicación de los algoritmos

2.1. Representación de soluciones

Para este problema tenemos varias formas posibles de representar las soluciones:

- Representación binaria: Cada solución está representada por un vector binario de longitud igual al número de características, donde las posiciones seleccionadas tendrán un 1 o **True** y las no seleccionadas un 0 o **False**. Esta opción, que será la que tomaremos, sólo es recomendable si no tenemos restricciones sobre el número de características seleccionadas.
- Representación entera: Cada solución es un vector de tamaño fijo $m \leq n$ con las características seleccionadas.
- Representación de orden: Cada solución es una permutación de n elementos, ordenados según la importancia de cada característica.

En las dos últimas representaciones el espacio de soluciones es mayor que el espacio de búsqueda. Además, la representación binaria nos facilita la aplicación de los operadores de cruce o mutación, manteniendo la consistencia.

Para esta práctica, aunque la representación de las soluciones también venga dada por la representación binaria, tendremos en todo momento una población en forma de **array** estructurado. Esto es, un **array** donde cada elemento está formado a su vez por dos atributos: el cromosoma, es decir, el vector de valores *booleanos* que determinan la selección o no de una característica, y el valor o *score* del mismo. De esta manera podemos saber la tasa de acierto de un vector solución sin tener que volver a llamar a la función de evaluación; sólo llamaremos a esta función cuando surja el nuevo individuo de la población.

2.2. Función objetivo

La función objetivo será el porcentaje de acierto en el conjunto de test para el clasificador 3-NN obtenido usando las distancias de los individuos ω en las dimensiones representadas por las características seleccionadas en el vector solución para el conjunto de entrenamiento. El objetivo será maximizar esta función. A la hora de buscar esta solución sólo estaremos manejando los datos de entrenamiento, luego aquí la función objetivo será la media de tasa de acierto para cada uno de los datos de entrenamiento con respecto a todos los demás, por lo que tenemos que usar la técnica de *Leave-One-Out*. Esta técnica consiste en quitar del conjunto de datos cada uno de los elementos, comprobar el acierto o no para este dato en concreto, y devolverlo al conjunto de datos. Así evitamos que los resultados estén sesgados a favor de la clase o etiqueta original, al contar siempre con un voto la clase verdadera.

La implementación de la función objetivo (obtener el *score* para Test) la he realizado basándome en el código paralelizado realizado para CUDA por Alejandro García Montoro para la función de *Leave One Out*. El pseudocódigo incluido se trata del esquema seguido para cada proceso, esto es, cada elemento en el conjunto de datos de entrenamiento, puesto que el método en Python para pasarle a la GPU los datos de entrenamiento, test, categorías y un puntero con la solución no tiene mayor interés.

```
targetFunction(data_train, categories_train,
               data_test, categories_test):
BEGIN
    test ← Get Process Number
    num_test ← length(data_test)

    EXIT IF test > num_test
    my_features ← data_test[test]

    k_nearest ← {{item: -1, distance: ∞}, size = k}

    FOR item IN data_train
        distance ← computeDistance(my_features, item)
        k_nearest ← update(item, distance)
    END

    class ← poll(classes of k_nearest)

    RETURN class = categories_test[test]
END
```

Esto en CUDA lo que hace es guardarnos, para cada proceso, si se ha acertado o no. Posteriormente se pasa el vector con cada resultado (cada ejecución de este código) de nuevo a Python y se calcula el porcentaje de aciertos. Nótese que no se realiza la proyección por las características seleccionadas, esto lo hacemos al pasar los datos.

Para la función de evaluación de las posibles soluciones que se van generando durante la búsqueda utilizo el método realizado por Alejandro García Montoro para usar CUDA. El algoritmo es similar al anterior, pero incluye *Leave One Out*:

```

targetFunctionLeaveOneOut(data_train, categories_train):
BEGIN
    sample ← Get Process Number
    num_samples ← length(data_train)

    EXIT IF sample > num_samples
    my_features ← data_train[sample]

    k_nearest ← {{item: -1, distance:∞}, size=k}

    FOR item IN data_train
        IF item ≠ sample THEN
            distance ← computeDistance(my_features, item)
            k_nearest ← update(item, distance)
        END
    END

    class ← poll(classes of k_nearest)

    RETURN class = categories_train[sample]
END

```

2.3. Operadores comunes

Para los dos algoritmos se comparte el operador de mutación y también, en cierta medida, el operador de cruce:

2.3.1. Operador de cruce

El operador de cruce de cada algoritmo se diferencia en que en el esquema estacionario únicamente se reproducen dos de los padres, mientras que en el esquema generacional, se seleccionan tantos padres como individuos tenga la población (se dará el caso con frecuencia de que un individuo se reproduzca más veces que otro). Sin embargo, sólo se cruzarán el 70 % de ellos, con lo que el operador de cruce para el caso generacional cruza únicamente el 70 % primero del vector de padres a cruzar, manteniendo el 30 % restante como estaba.

Operador de cruce de dos puntos El operador de cruce que se indica en la práctica para hacer es el cruce clásico en dos puntos. Para este cruce, dados dos padres tomamos dos posiciones aleatorias en dos vectores e intercambiamos la parte central, obteniendo a los hijos:

```

twoPointsCrossOperator( parent_1, parent_2 ):
BEGIN
    a,b ← {random{1,...,num_features-1}, size = 2,replace=False}

    desc_1 ← concatenate(parent_10,...,parent_1a-1,
                        parent_2a,...,parent_2b-1,

```


Ejemplo Sea una población de 30 individuos con 50 características. Esto hace un total de 1500 genes que pasarán por el proceso de mutación. $\lfloor \frac{1500}{1000} \rfloor = 1$, luego mutaremos un gen. Pero $1500 \equiv 500 \bmod(1000)$, distinto de 0, luego si se cumple $\text{random}() < \frac{1500 \bmod(1000)}{1000} = 0,5$, modificamos un gen adicional.

La operación para mutar un bit de un determinado individuo es la ya conocida operación **flip** que nos hacía obtener los vecinos en las prácticas anteriores:

```
flip(solution, position):
BEGIN
    neighbour ← copy(solution)
    actual_value ← solutionposition
    neighbourposition ← NOT actual_value
    RETURN neighbour
END
```

Por tanto, el operador de mutación nos queda de la siguiente manera:

```
mutate(descendants, mutation_prob):
BEGIN
    total_genes ← length(descendants)*num_features

    num_gens_to_mutate ← floor(total_genes*mutation_prob)

    IF random() < total_genes*mutation_prob-num_gens_to_mutate
        num_gens_to_mutate ← num_gens_to_mutate +1

    individ ← {random(descendants), size = num_gens_to_mutate}
    genes ← {random({0,...,num_features-1}), size = num_gens_to_mutate}
    FOR (individ, gen) IN (individ, genes):
        individ[chromosome] ← flip(individ[chromosome], gen)
        individ[score] ← 0
    END

    RETURN descendants
END
```

Ajustando el valor del individuo mutado nos aseguramos que si el individuo mutado pertenecía a la población (estaba en el 30 % de la selección de padres que no se ha cruzado) se vuelve a evaluar y nos ahorramos hacerlo si el individuo no hubiera mutado.

2.3.3. Torneo

Hay además una operación que se necesita para los dos esquemas y su sistema de elección, y es el torneo. Esta función se encarga de determinar cuál de dos individuos de la población es mejor y, por tanto cuál de ellos se reproducirá. En nuestro caso, entendemos que una solución es mejor que otra si tiene una mejor tasa de acierto o, en caso de tener la misma tasa de acierto, tiene seleccionadas menos características. Si se volviese a producir un empate, se devolvería uno de los dos candidatos al azar.


```

tournament(p_1, p_2):
BEGIN
  IF p_1 is better than p_2 THEN
    RETURN p_1
  ELSE IF p_2 is better than p_1 THEN
    RETURN p_2
  ELSE
    RETURN random({p_1, p_2})
END

```

3. Estructura del método de búsqueda

En esta práctica no sólo hay diversos operadores comunes, sino que el esquema general de los algoritmos es el mismo y las diferencias están, precisamente, en operadores concretos diferentes para cada algoritmo. Presentaré en primer lugar la estructura de los algoritmos genéticos implementados y posteriormente los operadores específicos de estos algoritmos.

```

geneticAlgorithm(data, categories, operators):
BEGIN
  MAX_CHECKS ← 1500
  MUTATION_PROBABILITY ← 0.001

  chromosomes ← {{random{T,F}: size = num_features }:size = 30}
  scores ← {score(data[chrom],categories): chrom ∈ chromosomes}

  population ← concatenate( (chromosomes, scores), by columns)
  sort(population, by scores)

  num_checks ← 30

  WHILE num_checks < MAX_CHECKS DO
    selected_parents ← selectionOperator(population)

    descendants ← crossOperator(selected_parents)
    descendants ← mutationOperator(descendants, MUTATION_PROBABILITY)

    FOR desc IN descendants which are not evaluated:
      score(data[desc],categories)
      num_checks ← num_checks + 1

    replaceOperator(population, descendants)
    sort(population, by scores)
  END
END

```

Esta estructura básica hará más fácil la construcción de algoritmos evolutivos, pues sólo tendremos que modificar los operadores de selección, cruce, mutación y reemplazamiento. El funcionamiento será el visto en clase: en primer lugar se genera una población aleatoria,

evaluamos y ordenamos con respecto a esta puntuación obtenida. Entonces, mientras el número de evaluaciones realizados sea menor que el máximo establecido (15000 en nuestro caso), se realiza el siguiente bucle:

- I Se seleccionan unos padres de la población con `selectionOperator`.
- II Estos padres se reproducen mediante `crossOperator`
- III Los cromosomas obtenidos se someten a una mutación aleatoria usando `mutationOperator`
- IV Se evalúan los hijos.
- V Se produce un reemplazo en la generación.
- VI Se reordena la población.

Finalmente, se devuelve el mejor individuo de la población existente al final de las evaluaciones.

3.1. Algoritmo Genético Generacional

3.1.1. Operador de selección

El operador de selección para el esquema generacional consiste en seleccionar aleatoriamente elementos de la población hasta obtener tantas parejas como elementos de la población y realizar el torneo entre estas parejas, quedándose con los vencedores.

```
selectionOperator_generational(population):  
BEGIN  
    tourn_cand ← {random(population): size = 2 length(population)}  
    parents ← {tournament(tourn_cand[i], tourn_cand[i+1]) :  
                i = 0, 2, ..., 2*(length(population)-1)}  
  
    RETURN parents  
END
```

3.1.2. Operador de reemplazamiento

En el operador de reemplazamiento, simplemente se intercambia la población actual por la de los descendientes, aunque como se da el elitismo, se reemplaza al peor individuo de la población de descendientes por el mejor de la población anterior si el mejor individuo de los descendientes no tuviese una mejor puntuación:

```

replaceOperator_generational(population, descendants):
BEGIN
    best ← first(population)
    IF best is better than best(descendants) THEN
        last(descendants) = best

    RETURN descendants
END

```

Para simplificar la lectura en el código y minimizar el número de operaciones, sobre todo usando los vectores de `numpy`, el orden es de menor a mayor, luego el mejor individuo se encuentra en la última posición. En el pseudocódigo se supone que los vectores están ordenados de mayor a menor tasa de aprendizaje.

3.2. Algoritmo Genético Estacionario

3.2.1. Operador de selección

El operador de selección en el esquema estacionario es el mismo que para el generacional, con la salvedad de que los padres seleccionados son únicamente 2, con lo que necesitamos 4 elementos que participen en sendos torneos.

3.2.2. Operador de reemplazamiento

El operador de reemplazamiento sí es diferente al del esquema generacional. Al tener únicamente dos descendientes en cada iteración, no podríamos sustituir enteramente la población por la siguiente generación, sino que seleccionamos los dos individuos de la población, los comparamos con los dos descendientes generados y escogemos los dos mejores de estos cuatro elementos.

```

replaceOperator_stationary(population, descendants):
BEGIN
    replacement ← concatenate(last(population, size=2), descendants)

    sort(replacement, by 'score')
    last(population, size = 2) ← first(replacement, size = 2)

    RETURN population
END

```

4. Algoritmo de comparación

Como algoritmo de comparación tenemos el algoritmo *greedy* SFS. Partiendo de un vector con ninguna característica seleccionada, exploramos por el entorno y nos quedamos con el vecino que genera una mejor tasa de acierto. Repetimos este proceso hasta que ningún vecino aporta una mejora a la solución obtenida.

```

greedySFS(data, categories):
BEGIN
    solution ← {F,...,F: size = num_features}
    current_value ← getValue(data, categories, solution)

    REPEAT
        neighbours ← {flip(solution,i): i ∈ characteristics}

        best_value ← maxneighbours getValue(data, categories, ·)

        IF best_value > current_value THEN
            solution ← argmaxneighbours getValue(data, categories, ·)

    WHILE (best_value > current_value)

    RETURN solution
END

```

5. Procedimiento para desarrollar la práctica

El código de la práctica está realizado en Python 3.5.1 y en CUDA. Como se ha comentado anteriormente, el código para el KNN está paralelizado usando el código de Alejandro García Montoro para usarlo con *leave-one-out* y añadiéndole un método para usarlo como función de evaluación de la solución obtenida para el conjunto de test. Esto ha permitido reducir los tiempos considerablemente.

Los paquetes utilizados son:

1. `scipy` para leer de una manera sencilla la base de datos.
2. `numpy` para el manejo de vectores y matrices y tratar que sea algo más eficiente en lugar de las listas de Python.
3. `ctype` para importar el generador de números aleatorios en C disponible en la página web de la asignatura.
4. `csv` para la lectura y escritura de ficheros `.csv` con los que manejar más cómodamente los datos.
5. `pycuda` y `jinja2` para la paralelización en CUDA.

La semilla con la que he realizado las ejecuciones es 3141592, insertada tanto en el generador en C como en el generador de números de `numpy` y en el propio de Python. He usado los dos porque pretendía usar el primero, que es con el que se realizan las particiones, pero al llegar a los métodos que usan los generadores de números pseudoaleatorios en su funcionamiento me dí cuenta de que tendría que repetir el código de importación del módulo en C para cada método, por lo que opté por usar en los métodos el `random` de `numpy`.

5.1. Ejecución del programa

La salida de cada ejecución (10 iteraciones de un algoritmo con una base de datos) se puede elegir entre mostrar por pantalla o redirigir a un archivo `.csv` para manejarlo posteriormente, por ejemplo para incluir la tabla en \LaTeX .

Los parámetros que acepta el programa son:

- Base de datos: Será una letra `W,L,A` que representa cada una de las bases de datos a utilizar. Este parámetro es el único obligatorio.
- Algoritmo utilizado: Por defecto es el KNN. Para introducir uno distinto, se usa `-a` seguido de una letra entre `K,S,G,E,GH,EH` que se corresponden con KNN, *greedy* SFS, algoritmo genético generacional y algoritmo genético estacionario, generacional con cruce HUX y estacionario con cruce HUX, respectivamente.
- Semilla. Para incluir una semilla, se añade `-seed` seguido del número que usaremos como semilla. Por defecto es 3141592.
- Salida por pantalla o a fichero. Se utiliza con el parámetro opcional `-write` para escribir en un fichero en una carpeta llamada **Resultados**. El nombre del fichero será la primera letra de la base de datos utilizada seguida por las iniciales del algoritmo. Incluye también la media, el mínimo, el máximo y la desviación típica para cada columna.
- `-h` o `--help` para mostrar la ayuda y cómo se introducen los parámetros.

Por tanto, la ejecución del programa se hará de la siguiente manera:

```
python Practica4.py base_de_datos [-a algoritmo -seed semilla -write T/F ]
```

Si por ejemplo queremos lanzar la base de datos de WDBC con el AGE, semilla 123456 y que los resultados se muestren por pantalla, escribimos

```
python Practica4.py W -a E -seed 123456
```

Si optamos por la base de datos *Arrhythmia* con el AGG con el cruce HUX y guardar el resultado en un fichero:

```
python Practica4.py A -a GH -write True
```

Para mostrar la introducción de parámetros:

```
python Practica4.py --help
```

6. Experimentos y análisis de resultados

6.1. Descripción de los casos

Los casos del problema planteados son tres, cada uno de ellos asociado a una base de datos:

- **WDBC**: Base de datos con los atributos estimados a partir de una imagen de una aspiración de una masa en la mama. Tiene 569 ejemplos, 30 atributos y debemos clasificar cada individuo en dos valores.
- **Movement Libras**: Base de datos con la representación de los movimientos de la mano en el lenguaje de signos LIBRAS. Tiene 360 ejemplos y consta de 91 atributos.
- **Arrhythmia**: Contiene datos de pacientes durante la presencia y ausencia de arritmia cardíaca. Tiene 386 ejemplos y 254 atributos para categorizar en 5 clases. He reducido el número de características eliminando las columnas que tuvieran el mismo valor para todos los datos. Está explicado en la práctica 2 que tardaba excesivamente en la búsqueda local si se intentaba bajar el número de características deseleccionando aquellas que no influyesen en la tasa de acierto.

6.2. Resultados

6.2.1. KNN

	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	red	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
Particion 1-1	95,7895	95,7747	0	0,0407	65,5556	73,8889	0	0,0341	63,9175	60,9375	0	0,1014
Particion 1-2	97,5352	94,0351	0	0,0119	73,8889	76,6667	0	0,0331	63,5417	61,8557	0	0,1094
Particion 2-1	95,4386	97,1831	0	0,0116	74,4444	65,5556	0	0,0334	59,7938	64,0625	0	0,0967
Particion 2-2	95,7747	96,8421	0	0,0114	68,3333	75	0	0,0331	64,0625	63,4021	0	0,1089
Particion 3-1	96,1404	97,1831	0	0,0115	61,6667	76,6667	0	0,0335	63,9175	63,5417	0	0,0967
Particion 3-2	97,5352	96,8421	0	0,0115	72,2222	80	0	0,0333	63,5417	63,4021	0	0,1087
Particion 4-1	95,7895	96,1268	0	0,0115	74,4444	75	0	0,0335	63,4021	62,5	0	0,0967
Particion 4-2	95,7747	97,193	0	0,0114	70	71,1111	0	0,0331	64,5833	64,9485	0	0,1088
Particion 5-1	96,1404	95,0704	0	0,0115	68,8889	71,1111	0	0,0332	64,9485	63,5417	0	0,0968
Particion 5-2	97,5352	95,0877	0	0,0114	70	76,6667	0	0,0333	63,0208	61,8557	0	0,1087
Media	96,3453	96,1338	0	0,0144	69,9444	74,1667	0	0,0334	63,4729	63,0047	0	0,1033
Max	97,5352	97,193	0	0,0407	74,4444	80	0	0,0341	64,9485	64,9485	0	0,1094
Min	95,7747	94,0351	0	0,0114	61,6667	71,1111	0	0,0331	59,7938	60,9375	0	0,0967
Desv. Típica	0,8014	1,0529	0	$8,7 \cdot 10^{-3}$	3,9083	3,8349	0	$3 \cdot 10^{-4}$	1,3382	1,1383	0	$5,8 \cdot 10^{-3}$

En este caso el análisis es el mismo que en las prácticas anteriores. Con el KNN se ve cómo es la BD en general y qué tasas de acierto se obtiene seleccionando todas las categorías. Las diferentes iteraciones y sus resultados no son más que para particiones distintas, pues la solución es la misma para todas las ejecuciones del algoritmo. Se ve que en **WDBC** y **Arrhythmia** se obtienen porcentajes similares para la clasificación dentro de la muestra y fuera, como cabría esperar, pues las particiones se hacen equilibradamente, y sin embargo en **Libras** el porcentaje de acierto fuera de la muestra es superior. Esto se puede deber a que hay un gran número de clases y no tantos representantes de esas clases como en las dos primeras bases de datos.

6.2.2. SFS

	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	red	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
Particion 1-1	92	89,0845	93,3333	0,2245	71	75,5556	90	1,9763	86	76,5625	97,482	4,6152
Particion 1-2	96	90,1754	93,3333	0,1951	75	62,7778	93,3333	1,3277	73	69,0722	98,2014	3,361
Particion 2-1	95	91,5493	93,3333	0,1946	81	66,6667	88,8889	2,2523	76	68,2292	97,8417	3,9263
Particion 2-2	96	93,6842	90	0,2707	75	74,4444	92,2222	1,5851	80	75,2577	98,2014	3,3409
Particion 3-1	96	94,7183	90	0,2738	72	81,6667	92,2222	1,5394	82	72,3958	96,7626	6,2122
Particion 3-2	96	94,7368	90	0,2702	75	71,6667	92,2222	1,434	83	72,6804	97,1223	5,5603
Particion 4-1	98	95,0704	86,6667	0,3459	73	57,7778	93,3333	1,2212	80	73,9583	98,5611	2,6402
Particion 4-2	95	93,3333	90	0,2696	77	68,8889	90	1,9021	80	68,5567	97,8417	4,0374
Particion 5-1	96	94,0141	86,6667	0,3529	76	67,2222	92,2222	1,4372	75	69,7917	98,5611	2,6352
Particion 5-2	96	91,9298	90	0,2669	81	70,5556	90	1,8991	76	74,7423	98,5611	2,6895
Media	95,6	92,8296	90,3333	0,2664	75,6	69,7222	91,4444	1,6574	79,1	72,1247	97,9137	3,9018
Max	98	95,0704	93,3333	0,3529	81	81,6667	93,3333	2,2523	86	76,5625	98,5611	6,2122
Min	92	89,0845	86,6667	0,1946	71	57,7778	88,8889	1,2212	73	68,2292	96,7626	2,6352
Desv. Típica	1,4283	1,9527	2,3333	0,0509	3,2	6,4082	1,4948	0,3147	3,8328	2,877	0,5976	1,1785

Para el SFS se obtienen buenos resultados comparándolos con el KNN, especialmente en **Arrhythmia**, debido a que al haber muchas características y relativamente pocas clases, seleccionando bien unas pocas características se obtienen muy buenos resultados, introduciendo ruido con otras.

6.2.3. Algoritmo basado en colonia de hormigas

	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	red	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
Particion 1-1	98,5965	97,1831	56,6667	76,2891	75,5556	75	50	171,6746	82,4742	64,0625	34,7826	846,7426
Particion 1-2	99,2958	95,0877	46,6667	87,2603	80,5556	76,6667	50	172,6995	70,8333	62,8866	28,4585	1,038,1132
Particion 2-1	98,2456	96,1268	33,3333	102,834	80,5556	65,5556	55,5556	154,6355	73,1959	66,1458	24,1107	911,6371
Particion 2-2	97,8873	96,4912	46,6667	87,0733	75,5556	73,3333	53,3333	166,1735	74,4792	68,5567	34,3874	990,7866
Particion 3-1	97,8947	94,7183	50	87,5283	70,5556	77,7778	51,1111	172,3702	74,2268	64,5833	23,7154	952,948
Particion 3-2	98,2394	95,0877	53,3333	76,7275	77,2222	77,2222	51,1111	169,1397	78,125	65,9794	35,1779	959,9031
Particion 4-1	98,9474	96,1268	70	66,1774	80,5556	71,1111	54,4444	157,8225	70,1031	63,0208	23,3202	938,3802
Particion 4-2	97,8873	94,386	53,3333	71,9413	77,2222	71,1111	46,6667	181,3022	81,25	65,9794	32,8063	974,0427
Particion 5-1	97,5439	95,7747	43,3333	96,6748	79,4444	72,2222	54,4444	155,7864	75,2577	62,5	28,0632	896,1235
Particion 5-2	98,5916	95,4386	46,6667	92,3847	79,4444	75	53,3333	160,9737	76,5625	66,4948	31,6206	1,000,6204
Media	98,313	95,6421	50	84,4891	77,6667	73,5	52	166,2578	75,6508	65,021	29,6443	950,9298
Max	99,2958	97,1831	70	102,834	80,5556	77,7778	55,5556	181,3022	82,4742	68,5567	35,1779	1,038,1132
Min	97,5439	94,386	33,3333	66,1774	70,5556	65,5556	46,6667	154,6355	70,1031	62,5	23,3202	846,7426
Desv. Típica	0,5162	0,8167	9,0676	10,9142	3,0205	3,5053	2,5724	8,2864	3,8527	1,8384	4,5066	52,7613

Los mejores resultados con respecto a los algoritmos de comparación (KNN y SFS) se obtienen en las bases de datos **WDBC** y **Libras**, en cambio en **Arrhythmia** los mejores resultados siguen siendo los del algoritmo **SFS** debido a la importancia de la reducción de características comentada en el apartado anterior. Si comparamos los resultados con los obtenidos en la práctica 2, tampoco sale bien parado pues los resultados son inferiores, especialmente en las bases de datos de mayor tamaño. Sin embargo, el tiempo en **Arrhythmia** es menor que **BMB** e **ILS** obteniendo los tres algoritmos resultados similares, con lo que cabe pensar que los resultados podrían mejorar si una parte de las evaluaciones se dedicaran a hacer búsqueda local.

6.2.4. Algoritmo basado en hormigas Max-Mín

	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	red	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
Particion 1-1	97,8947	96,4789	43,3333	103,2269	77,7778	75,5556	64,4444	136,6319	78,3505	64,0625	23,3202	900,374
Particion 1-2	99,2958	93,3333	53,3333	73,3013	83,3333	75	63,3333	139,921	70,3125	60,3093	23,3202	1,045,0403
Particion 2-1	98,2456	95,7747	53,3333	73,2874	82,7778	68,3333	57,7778	139,3208	75,2577	64,0625	19,3676	893,0498
Particion 2-2	98,2394	96,1404	40	87,6747	77,7778	72,2222	58,8889	145,0618	71,875	65,9794	22,5296	1,046,6335
Particion 3-1	98,5965	94,3662	60	74,526	72,2222	80,5556	62,2222	146,3931	71,6495	65,1042	25,2964	866,6091
Particion 3-2	97,8873	95,4386	26,6667	102,5508	81,1111	80,5556	54,4444	156,2113	72,3958	62,8866	23,3202	1,045,6902
Particion 4-1	98,9474	95,4225	66,6667	63,3089	81,1111	74,4444	45,5556	165,5814	71,134	62,5	20,5534	947,923
Particion 4-2	97,8873	95,0877	40	98,1373	80,5556	71,6667	55,5556	154,0673	77,6042	63,4021	31,2253	964,6313
Particion 5-1	97,8947	95,0704	46,6667	96,1828	81,1111	70	54,4444	158,4285	78,866	60,4167	25,6917	901,1278
Particion 5-2	99,2958	95,0877	50	93,3565	80,5556	77,2222	47,7778	173,4431	76,5625	65,9794	25,6917	994,6445
Media	98,4185	95,22	48	86,5553	79,8333	74,5556	56,4444	151,506	74,4008	63,4703	24,0316	960,5723
Max	99,2958	96,4789	66,6667	103,2269	83,3333	80,5556	64,4444	173,4431	78,866	65,9794	31,2253	1,046,6335
Min	97,8873	93,3333	26,6667	63,3089	72,2222	68,3333	45,5556	136,6319	70,3125	60,3093	19,3676	866,6091
Desv. Típica	0,55	0,847	10,7703	13,5829	3,0535	3,9189	5,9587	11,522	3,1074	1,9122	3,1062	66,083

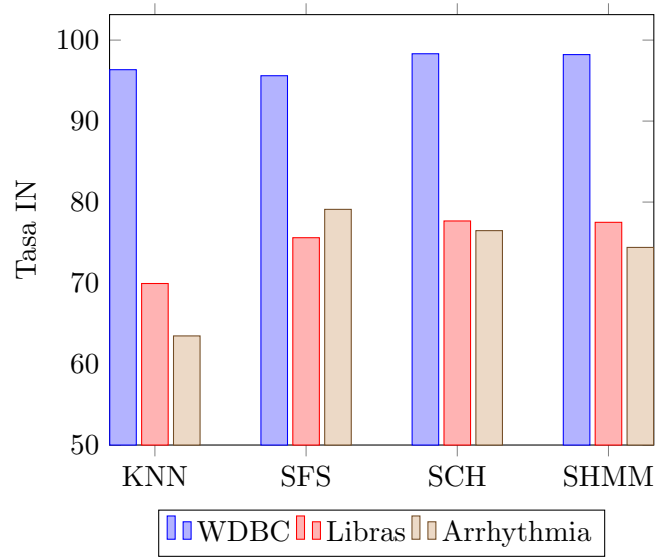
Si comparamos con los resultados del algoritmo genético con el esquema generacional los resultados son similares, aunque no es así en el tiempo, siendo superior en el esquema estacionario. Esto se puede deber a que se hacen muchas reordenaciones cuando sólo se han introducido dos elementos en cada generación, por lo que podríamos pensar en cambiar el algoritmo de ordenación y también en aumentar la presión para que se introduzcan más elementos en cada generación.

6.2.5. Comparación

	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
KNN	96.3453	96.1338	0.0000	0.0144	69.9444	74.1667	0.0000	0.0334	63.4729	63.0047	0.0000	0.1033
SFS	95.6000	92.8296	90.3333	0.2664	75.6000	69.7222	91.4444	1.6574	79.1000	72.1247	97.9137	3.9018
SCH	98.3130	95.6421	50.0000	84.4891	77.6667	73.5000	52.0000	166.2578	76.4728	64.3482	68.5771	544.5233
SHMM	98.2081	95.3606	33.3333	122.0403	77.5000	74.4444	47.3333	182.7097	74.4008	63.4703	24.0316	960.5723

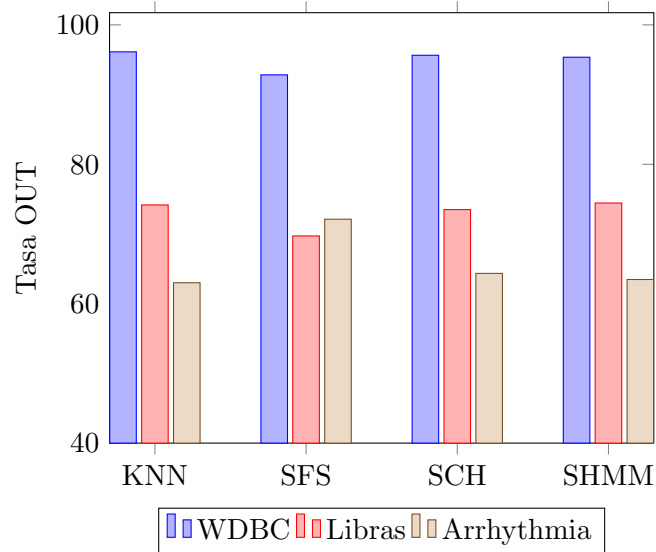
6.3. Análisis de los resultados

6.3.1. Tasa In



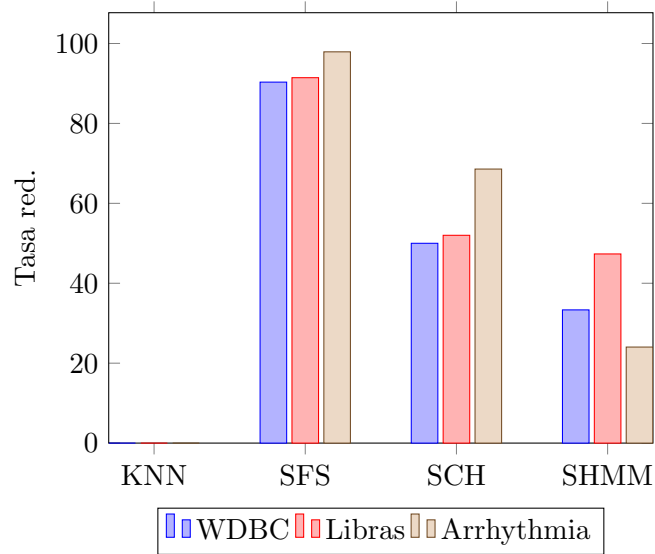
Se aprecia una mejora de los algoritmos evolutivos sobre los algoritmos de comparación, de nuevo como en la práctica 2, salvo para la base de datos **Arrhythmia**, con la salvedad de que ahora el no hay ningún algoritmo, como estaba el **GRASP** en la práctica anterior que se le acerque. De entre los algoritmos genéticos, se observan mejores resultados para el operador de cruce uniforme, en especial en las BD mayores, donde el margen de mejora era más amplio. De los dos esquemas implementados, el generacional se muestra más efectivo, posiblemente por las razones comentadas previamente: al ir introduciendo hijos poco a poco, con una probabilidad uniforme de la selección de los padres a participar en el torneo, no estamos cambiando tanto la población en cada generación como para que el nivel general aumente.

6.3.2. Tasa Out



Los resultados para la tasa fuera de la población son para los algoritmos genéticos muy parejos y las diferencias no son determinantes como para decir que un algoritmo es mejor que otro, pero sí que usando el operador de cruce uniforme, los resultados vuelven a ser un poco superiores. También para esta variable el ganador en *Arrhythmia* es el algoritmo *SFS* el que obtiene mejores resultados. En cambio para *WDBC* sí que hay diferencias a favor de los algoritmos genéticos en detrimento de *SFS*. En esto tiene mucho peso la distribución de los valores de la BD y el número de características, pues los algoritmos genéticos, que no tienen ningún mecanismo para disminuir el número de características, se ven perjudicados cuando este número es elevado y sólo hay unas pocas relevantes.

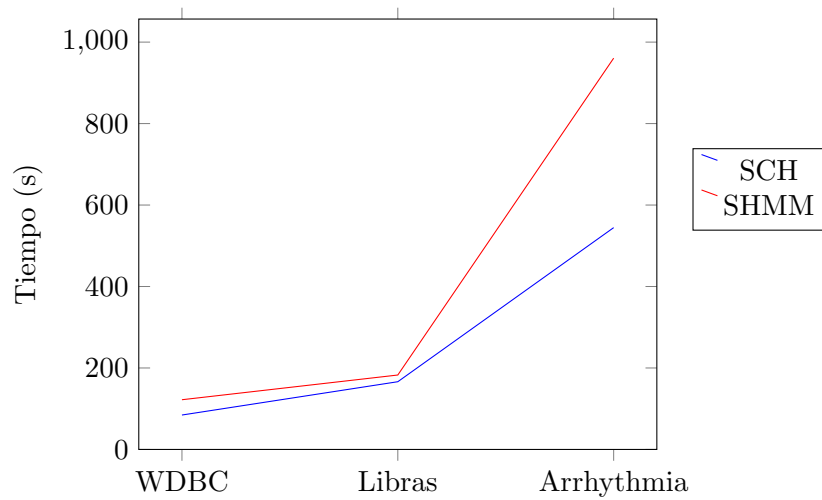
6.3.3. Tasa reducción



Con la tasa de reducción se observa que las hipótesis realizadas anteriormente son coherentes con los resultados. Los mejores resultados se obtienen con el esquema generacional y, en concreto, con el operador de cruce uniforme. Las tasas de reducción son claramente mayores en *SFS* y esto es efectivo en *Arrhythmia*, aunque no tanto en las otras bases de datos.

6.3.4. Tiempos

DB	0	1	2	3
TW	0.0144	0.2664	84.4891	122.0403
TL	0.0334	1.6574	166.2578	182.7097
TA	0.1033	3.9018	544.5233	960.5723



No se han incluido los tiempos para KNN ni SFS debido a que no son relevantes con respecto a los algoritmos genéticos. Tampoco existen diferencias significativas usando distintos operadores de cruce. Lo que sí se observa es un mayor tiempo para el algoritmo genético estacionario, especialmente en *Arrhythmia*. De nuevo debemos pensar en que estamos ordenando en cada iteración del bucle general una ordenación, que es efectiva cuando la nueva generación ha sufrido cambios (de un 70 % en el esquema generacional), pero que en el estacionario sólo introducimos dos individuos nuevos, que pueden incluso no entrar en la nueva población si fuesen peores. Además, la ordenación en este caso, debido al operador de selección, es poco efectiva en cuanto a que sólo nos sirve para tener localizados a los dos peores individuos de la población; en cambio si influyese la posición en la probabilidad de ser seleccionado para el torneo y posterior cruce, quizá los resultados mejorasen con un tiempo similar.

7. Bibliografía

- Módulo en `scikit` para KNN
- Para realizar las tablas en \LaTeX : Manual PGFPLOTSTABLE