Práctica 5 Metaheurísticas. Algoritmos Meméticos para el problema de la selección de características

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

9 de julio 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 Memético con optimización cada 10 generaciones sobre todos los individuos
2. Algoritmo Memético con optimización cada 10 generaciones sobre el 10 % de los individuos
3. Algoritmo Memético con optimización cada 10 generaciones sobre el mejor 10 % de los individuos

Índice

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 $f : \Omega \longrightarrow C = \{C_1, \dots, C_M\}$, 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 pertenzcan 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

   class ← poll(classes of k_nearest)

   RETURN class = categories_train[sample]
END
```

2.3. Operadores comunes

En esta práctica se ha seguido el mismo procedimiento que en la práctica 3 con relación a los operadores: programar los operadores en distintas funciones y pasarlo al algoritmo memético como argumento. Al considerar algoritmos meméticos basados en el algoritmo generacional de la práctica 3 ahora los operadores comunes son los propios de un algoritmo genético, siendo distintos únicamente los métodos para realizar las optimizaciones locales.

2.3.1. Operador de cruce

En la operación de cruce, como en el esquema generacional de la práctica 3, se hará una selección de tantos padres como individuos tenga la población. Sin embargo, sólo se cruzarán el 70 % de ellos, con lo que el operador cruza únicamente el 70 % primero del vector de padres a cruzar, manteniendo el 30 % restante como estaba. El operador implementado es la modificación sobre el cruce uniforme (HUX) puesto que, como se observa en la práctica 3, se obtienen mejores resultados que con el cruce clásico en dos puntos.

Operador de cruce uniforme Esta modificación del operador *Half Cross Uniform* (HUX) consiste en, dados dos padres, asignar a cada gen del hijo el valor del gen de los padres si éste coincide en ambos; si es distinto, asignamos al gen de cada uno de los hijos los valores `True` o `False` aleatoriamente. Esto significa que, para un gen en el que los padres tienen distintos valores, un hijo (aleatoriamente en cada gen) recibirá `True` y el otro `False`. Así maximizamos la distancia *hamming* entre los hijos, obteniendo mayor diversidad.

```
huxCrossOperator( parent_1, parent_2 ):
BEGIN
   FOR j IN {1,..., num_features} :
      IF parent_1[j] = parent_2[j] THEN
         desc_1[j] ← parent_1[j]
         desc_2[j] ← parent_1[j]
      ELSE
         gen ← random({True,False})
         desc_1[j] ← gen
         desc_2[j] ← not gen
   END

   RETURN desc_1, desc_2
END
```

2.3.2. Operador de mutación

El operador de mutación se encarga de introducir diversidad en la población, favoreciendo la búsqueda en distintas zonas del espacio. Sin embargo, no podemos basar nuestra estrategia de búsqueda de buenas soluciones sólo en la mutación, pues esto nos haría explorar más zonas de búsqueda, sí, pero haciéndolo de forma aleatoria, no intensificando sobre regiones de las que se tenga información de que pueden ser buenas. Debido a esto, la probabilidad de mutación será baja, en concreto, de 0,001 por cada gen. Para ahorrar cálculos repetitivos he descartado la opción de hacer un `random` por cada gen que se somete al operador de mutación, mutando 1 gen seleccionado aleatoriamente por cada 1000 genes que pasen por el método en cuestión. Además, para el resto de dividir el número de genes de la población de hijos entre 1000 se genera un número aleatorio. Si este número es menor que el resto de la división por la probabilidad de realizar una mutación, mutaremos 1 gen más.

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 $1500 \equiv 500 mod(100)$, distinto de 0, luego si se cumple `random()` $< \frac{1500 mod(1000)}{1000} = 0{,}5$, 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, positon):
BEGIN
        neighbour ← copy(solution)
        actual_value ← solution_position
        neighbour_position ← 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_genes_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

Para el sistema de elección necesitamos 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
```

2.3.4. Operador de selección

El operador de selección para el algoritmo memético 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
```

2.3.5. 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 acierto.

2.3.6. Búsqueda local

La aplicación de la búsqueda local se hace a individuos de la población cada 10 generaciones y es de baja intensidad, es decir, se intenta optimizar llevando a cabo una única iteración de la búsqueda local. Devolvemos también el número de iteraciones realizadas por la búsqueda local para, en el algoritmo memético, contabilizar las evaluaciones tanto de la búsqueda local como del algoritmo memético.

```
localSearch1Iteration(data, categories, solution) BEGIN

   first_neig ← random{1,..., num_features}
   neighbours ← {flip(solution,i): i=first_neig,...,first_neig-1}
   found_better ← FALSE
   num_checks

   FOR neig IN neighbours WHILE NOT found_better
      IF neigh is better than solution THEN
         found_better ← TRUE
```

```
            solution ← neig
         num_checks ← num_checks + 1
      END

      RETURN solution, num_checks
END
```

Se han obtenido también los resultados de los tres métodos pero realizando una búsqueda local de alta intensidad, es decir, aplicando la búsqueda local hasta que ésta no encontrase mejora. Esto hace que se aumente la intensificación pero pagando a cambio de realizar menos iteraciones de la parte evolutiva del proceso. El método de búsqueda sería el siguiente:

```
localSearch(data, categories, solution) BEGIN

   max_checks ← 15000
   num_checks ← 0

   REPEAT
      new_sol, checks ← localSearch1Iteration(data,
                        categories, solution)

      found_better ← new_sol is better than solution
      num_checks  ← num_checks + checks

      IF found_better THEN
         solution  ← new_solution

   WHILE( found_better AND  num_checks < max_checks)

   RETURN solution, num_checks
END
```

2.4. Aplicación de la búsqueda local

En este apartado se explica la aplicación de la búsqueda local según el algoritmo pedido en cada caso.

2.4.1. Optimización de individuos de la población

La selección de los individuos a optimizar es la misma tanto para cuando optimizamos localmente con probabilidad 1 (optimizamos todos los individuos) como para cuando lo hacemos con probabilidad 0,1. En lugar de ir por cada individuo y obtener un número aleatorio, optamos por, al igual que con el operador de mutación, seleccionar tantos individuos como esperamos obtener con la selección aleatoria, todos los individuos en el primer caso y un 10 % en el segundo. Por tanto, dada la probabilidad de aplicar la búsqueda local el primer método y el segundo tendrán el mismo operador de optimización.

```
localSearchOperator(current_generation, population, data, categories)
    BEGIN

    num_checks ← 0

    IF current_generation ≡ 0 mod(num_generations) THEN
        agents_to_ls ← random(population,
                              size = length(population) * prob_local_search)

        FOR agent IN agent_to_ls
            agent, ls_checks ← localSearch1Iteration(data, categories,
                                                     agent)

            num_checks ← num_checks + ls_checks
        END

    return num_checks
END
```

2.4.2. Optimización de mejores individuos de la población

Para este método la única diferencia es que debemos ordenar en primer lugar el vector de la población según la puntuación, tomaremos la probabilidad como parámetro para seleccionar y optimizar un cierto número de individuos.

```
localSearchOperator(current_generation, population, data, categories)
    BEGIN

    num_checks ← 0

    IF current_generation ≡ 0 mod(num_generations) THEN
        sort(population, by scores)
        agents_to_ls ← first(size = length(population)*prob_local_search)

        FOR agent IN agent_to_ls
            agent, ls_checks ← localSearch1Iteration(data, categories,
                                                     agent)

            num_checks ← num_checks + ls_checks
        END

    return num_checks
END
```

3. Estructura del método de búsqueda

En esta práctica el esquema general de los algoritmos es idéntico entre los distintos métodos propuestos pero también es igual que el esquema de los algoritmos genéticos de la

práctica 3 a excepción de la inclusión de la búsqueda local, que se incluye como operador. Se incluye en primer lugar la estructura de los algoritmos memético implementados y posteriormente los operadores de búsqueda local específicos de estos algoritmos.

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

   chromosomes ← {{random{T,F}: size = num_features }:size = 10}
   scores ← {score(data[chrom],categories): chrom ∈ chromosomes}

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

   num_checks ← 10
   num_generation ← 0

   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)

      ls_checks ← localSearchOperator(num_generation, population, data,
          categories)
      num_checks ← ls_checks
      num_generation ← num_generation + 1

      sort(population, by scores)
   END
END
```

Esta estructura básica hará más fácil la construcción de algoritmos meméticos, pues sólo tendremos que introducir los operadores de selección, cruce, mutación y reemplazamiento adecuados y modificar el operador de búsqueda local según el método a ejecutar. 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 realiza una optimización local.

VII Se reordena la población.

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

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 ← máx_neighbours getValue(data, categories, ·)

      IF best_value > current_value THEN
         solution ← argmax_neighbours 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 tecla entre `K,S,1,2,3` que se corresponden con KNN, *greedy* SFS, algoritmo memético con optimización local cada diez generaciones en todos los individuos, en un 10 % aleatorio o en el 10 % mejor, 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 Practica5.py base_de_datos [-a algoritmo -seed semilla -write T/F ]
```

Si por ejemplo queremos lanzar la base de datos de `WDBC` con optimización en todos los individuos, semilla 123456 y que los resultados se muestren por pantalla, escribimos

```
python Practica5.py W -a 1 -seed 123456
```

Si optamos por la base de datos `Arrhythmia` con optimización en el 10 % mejor y guardar el resultado en un fichero:

```
python Practica5.py A -a 3 -write True
```

Para mostrar la introducción de parámetros:

```
python Practica5.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
Partición 1-1	95,7895	95,7747	0	0,0407	65,5556	73,8889	0	0,0341	63,9175	60,9375	0	0,1014
Partición 1-2	97,5352	94,0351	0	0,0119	73,8889	76,6667	0	0,0331	63,5417	61,8557	0	0,1094
Partición 2-1	95,4386	97,1831	0	0,0116	74,4444	65,5556	0	0,0334	59,7938	64,0625	0	0,0967
Partición 2-2	95,7747	96,8421	0	0,0114	68,3333	75	0	0,0331	64,0625	63,4021	0	0,1089
Partición 3-1	96,1404	97,1831	0	0,0115	61,6667	76,6667	0	0,0335	63,9175	63,5417	0	0,0967
Partición 3-2	97,5352	96,8421	0	0,0115	72,2222	80	0	0,0333	63,5417	63,4021	0	0,1087
Partición 4-1	95,7895	96,1268	0	0,0115	74,4444	75	0	0,0335	63,4021	62,5	0	0,0967
Partición 4-2	95,7747	97,193	0	0,0114	70	71,1111	0	0,0331	64,5833	64,9485	0	0,1088
Partición 5-1	96,1404	95,0704	0	0,0115	68,8889	71,1111	0	0,0332	64,9485	63,5417	0	0,0968
Partición 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
Partición 1-1	92	89,0845	93,3333	0,2245	71	75,5556	90	1,9763	86	76,5625	97,482	4,6152
Partición 1-2	96	90,1754	93,3333	0,1951	75	62,7778	93,3333	1,3277	73	69,0722	98,2014	3,361
Partición 2-1	95	91,5493	93,3333	0,1946	81	66,6667	88,8889	2,2523	76	68,2292	97,8417	3,9263
Partición 2-2	96	93,6842	90	0,2707	75	74,4444	92,2222	1,5851	80	75,2577	98,2014	3,3409
Partición 3-1	96	94,7183	90	0,2738	72	81,6667	92,2222	1,5394	82	72,3958	96,7626	6,2122
Partición 3-2	96	94,7368	90	0,2702	75	71,6667	92,2222	1,434	83	72,6804	97,1223	5,5603
Partición 4-1	98	95,0704	86,6667	0,3459	73	57,7778	93,3333	1,2212	80	73,9583	98,5611	2,6402
Partición 4-2	95	93,3333	90	0,2696	77	68,8889	90	1,9021	80	68,5567	97,8417	4,0374
Partición 5-1	96	94,0141	86,6667	0,3529	76	67,2222	92,2222	1,4372	75	69,7917	98,5611	2,6352
Partición 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 memético con optimización sobre todos los individuos

	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	97,5439	95,7747	56,6667	82,8147	75	75,5556	51,1111	158,2981	79,3814	65,625	52,5692	654,9527
Particion 1-2	99,2958	95,0877	46,6667	97,4404	78,8889	71,1111	47,7778	165,5432	70,8333	61,3402	50,5929	785,4513
Particion 2-1	98,2456	96,1268	53,3333	88,7401	80,5556	67,7778	55,5556	148,4391	73,7113	67,1875	54,9407	610,674
Particion 2-2	98,2394	94,7368	53,3333	88,9186	75	75	54,4444	150,8433	72,9167	63,4021	56,1265	780,0308
Particion 3-1	98,5965	95,4225	56,6667	86,1501	71,1111	79,4444	54,4444	147,5585	72,1649	63,5417	53,7549	689,6296
Particion 3-2	97,8873	95,7895	46,6667	97,4354	77,2222	80	56,6667	145,2306	75	66,4948	54,5455	829,8546
Particion 4-1	99,2982	96,4789	56,6667	84,4387	80,5556	75,5556	57,7778	141,448	75,2577	63,0208	64,0316	493,0943
Particion 4-2	97,5352	98,2456	46,6667	97,6009	77,7778	71,1111	51,1111	156,8219	74,4792	68,5567	52,9644	809,4174
Particion 5-1	97,5439	94,3662	53,3333	88,7916	80	68,8889	47,7778	167,8981	75,7732	64,5833	56,917	612,984
Particion 5-2	98,9437	97,193	50	94,8636	76,6667	73,3333	54,4444	153,5306	74,4792	65,4639	58,8933	653,5781
Media	98,3129	95,9222	52	90,7194	77,2778	73,7778	53,1111	153,5612	74,3997	64,9216	55,5336	691,9667
Max	99,2982	98,2456	56,6667	97,6009	80,5556	80	57,7778	167,8981	79,3814	68,5567	64,0316	829,8546
Min	97,5352	94,3662	46,6667	82,8147	71,1111	67,7778	47,7778	141,448	70,8333	61,3402	50,5929	493,0943
Desv. Típica	0,6629	1,1035	4	5,3711	2,8377	3,911	3,3259	8,1729	2,1954	2,0586	3,6064	102,249

Como pasa con los demás algoritmos implementados hasta ahora, se observa el incremento esperable con respecto al `KNN` y con respecto a `SFS` (salvo en `Arrhythmia` donde el algoritmo *greedy* sigue siendo más efectivo).

6.2.4. Algoritmo memético con probabilidad de optimización 0.1

	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	97,5439	95,7747	56,6667	84,5276	74,4444	75	54,4444	154,8033	77,3196	64,0625	62,0553	541,446
Particion 1-2	99,6479	94,386	50	95,8134	82,2222	76,1111	63,3333	130,632	71,3542	58,7629	67,5889	641,6245
Particion 2-1	98,2456	96,1268	66,6667	83,8883	79,4444	64,4444	50	165,4983	74,2268	61,4583	61,2648	590,3465
Particion 2-2	97,8873	96,4912	56,6667	81,5165	76,1111	75,5556	55,5556	153,3831	73,9583	65,4639	66,4032	617,3759
Particion 3-1	97,8947	96,4789	56,6667	85,0811	68,3333	77,2222	52,2222	158,3535	74,7423	65,1042	63,6364	564,3825
Particion 3-2	97,5352	95,4386	53,3333	91,1654	79,4444	76,6667	52,2222	166,7048	76,0417	65,4639	65,6126	622,4327
Particion 4-1	98,9474	95,4225	53,3333	90,2284	81,6667	74,4444	54,4444	156,0849	76,2887	69,2708	70,751	471,4687
Particion 4-2	97,8873	94,0351	50	92,1495	80	73,3333	58,8889	143,8847	76,0417	65,4639	67,9842	616,7797
Particion 5-1	97,5439	95,0704	46,6667	95,7547	79,4444	71,1111	51,1111	162,8499	75,2577	62,5	61,2648	608,6627
Particion 5-2	98,2394	94,0351	63,3333	73,1791	77,7778	73,8889	51,1111	161,461	72,3958	62,8866	62,4506	691,9257
Media	98,1373	95,3259	55,3333	87,3304	77,8889	73,7778	54,3333	155,3655	74,7627	64,0437	64,9012	596,6445
Max	99,6479	96,4912	66,6667	95,8134	82,2222	77,2222	63,3333	166,7048	77,3196	69,2708	70,751	691,9257
Min	97,5352	94,0351	46,6667	73,1791	68,3333	64,4444	50	130,632	71,3542	58,7629	61,2648	471,4687
Desv. Típica	0,6489	0,8851	5,8119	6,6939	3,8873	3,5382	3,8952	10,4003	1,7475	2,698	3,0962	57,0462

Con la optimización local sólo en un 10 % de los individuos se observa que los resultados son muy similares a optimizar toda la población. Sin embargo las diferencias en el tiempo de ejecución en la base de datos `Arrhythmia` sí que son significativas.

6.2.5. Algoritmo memético con optimización sobre los mejores individuos

	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	95,4225	60	80,9875	71,1111	75,5556	53,3333	153,603	77,8351	64,5833	59,6838	531,7928
Particion 1-2	99,2958	94,0351	70	77,6759	80	75	61,1111	140,6217	69,7917	61,8557	55,336	757,7418
Particion 2-1	98,2456	96,1268	63,3333	80,2984	79,4444	68,3333	64,4444	123,4238	75,7732	64,0625	60,0791	590,3857
Particion 2-2	98,2394	95,4386	50	91,9506	73,8889	77,2222	41,1111	200,9673	75	67,5258	49,8024	847,1155
Particion 3-1	97,8947	95,0704	70	66,7825	70	75,5556	51,1111	164,7155	71,134	63,5417	54,9407	655,7046
Particion 3-2	97,8873	96,4912	60	82,5852	81,1111	79,4444	55,5556	149,0093	77,0833	68,5567	59,6838	676,1302
Particion 4-1	98,2456	95,0704	53,3333	87,4321	80,5556	70	54,4444	152,3956	68,5567	64,5833	56,5217	668,7804
Particion 4-2	97,5352	94,0351	60	78,1363	80	71,1111	53,3333	154,4069	80,2083	65,9794	55,7312	724,4675
Particion 5-1	97,193	95,4225	60	81,9738	77,7778	73,8889	54,4444	148,5065	76,2887	63,0208	57,3123	607,6495
Particion 5-2	98,5916	94,7368	50	100,4913	77,7778	75,5556	51,1111	174,8954	73,4375	67,5258	49,4071	837,8101
Media	98,1023	95,185	59,6667	82,8314	77,1667	74,1667	54	156,2545	74,5108	65,1235	55,8498	689,7578
Max	99,2958	96,4912	70	100,4913	81,1111	79,4444	64,4444	200,9673	80,2083	68,5567	60,0791	847,1155
Min	97,193	94,0351	50	66,7825	70	68,3333	41,1111	123,4238	68,5567	61,8557	49,4071	531,7928
Desv. Típica	0,5481	0,7511	6,7412	8,5616	3,8446	3,2323	5,8836	19,6937	3,5412	2,0826	3,5925	97,9093

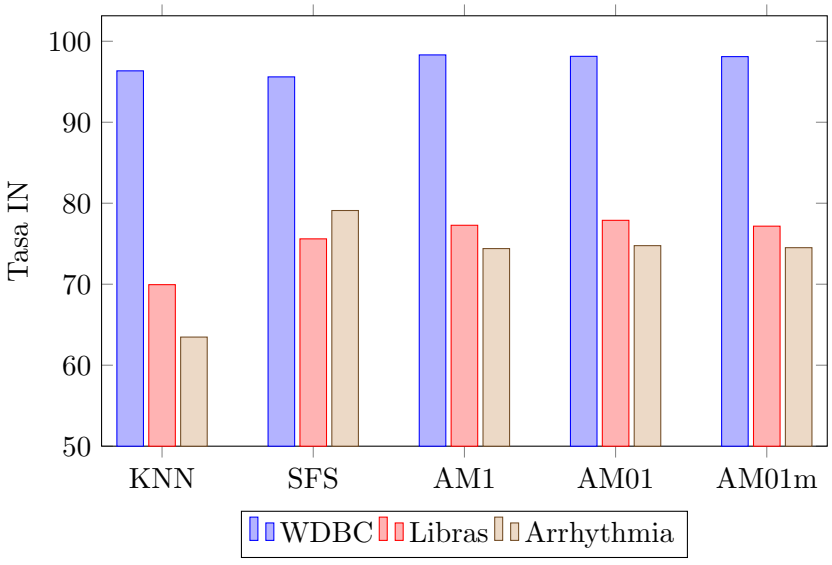
Con la optimización sobre los mejores individuos se obtienen resultados muy similares a los de los otros algoritmos meméticos, observando que en tiempos de ejecución se vuelve a los tiempos de ejecución en `Arrhythmia` de la aplicación de la BL a todos los individuos.

6.2.6. 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
AM1	98.3129	95.9222	52.0000	90.7194	77.2778	73.7778	53.1111	153.5612	74.3997	64.9216	55.5336	691.9667
AM01	98.1373	95.3259	55.3333	87.3304	77.8889	73.7778	54.3333	155.3655	74.7627	64.0437	64.9012	596.6445
AM01m	98.1023	95.1850	59.6667	82.8314	77.1667	74.1667	54.0000	156.2545	74.5108	65.1235	55.8498	689.7578

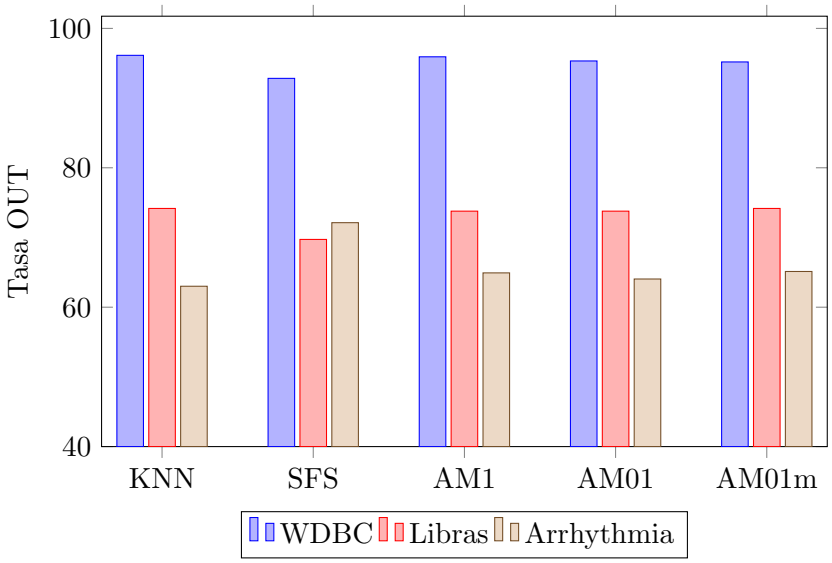
6.3. Análisis de los resultados

6.3.1. Tasa In



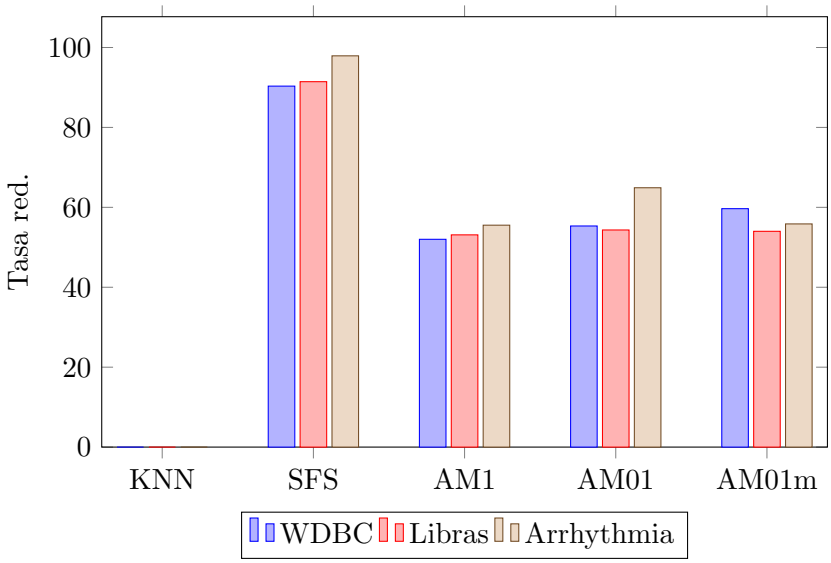
En cuanto a la tasa de acierto dentro de la muestra se observa que los tres algoritmos meméticos se comportan mejor que el algoritmo de control `KNN` y que el `SFS` (con la excepción de la BD `Arrhythmia`). Sin embargo entre ellos las diferencias no son significativas y se limitan a pocas décimas no sólo en `WDBC`, donde ya se ha comentado en otras prácticas que es muy difícil obtener una mejora significativa debido al alto nivel de acierto que se obtiene en esta BD, sino también en `Libras` y `Arrhythmia`.

6.3.2. Tasa Out



Con respecto a la tasa de acierto fuera de la muestra también se obtienen resultados muy similares para los tres algoritmos, mejores que `SFS` en `WDBC` y `Libras` y que `KNN` en `Arrhtyhmia`.

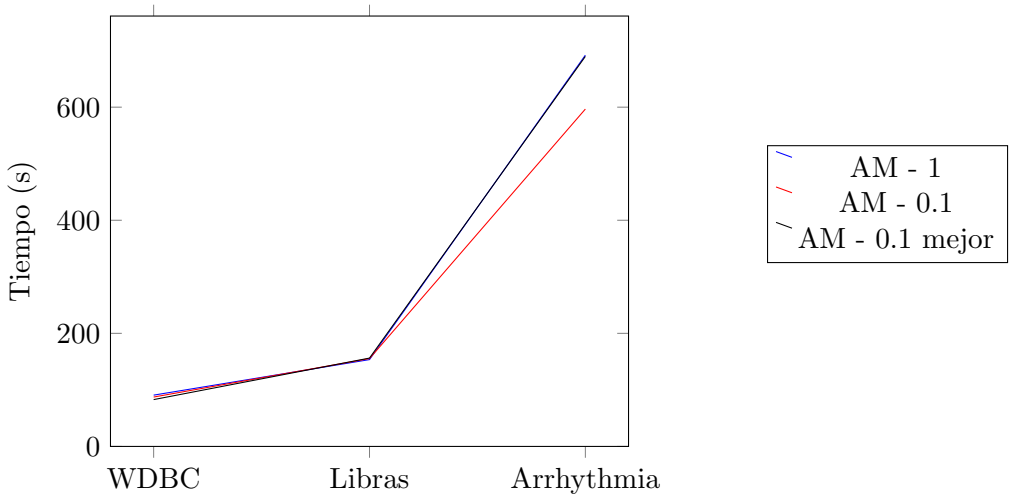
6.3.3. Tasa reducción



Por la implementación de los algoritmos para la decisión de las mejores soluciones, podemos considerar que la reducción del número de características seleccionadas no ha sido un objetivo principal, luego se espera una tasa de reducción no muy elevada y simplemente aquellas que ofrezcan una mayor tasa de acierto. Como vemos en los resultados obtenidos, estas tasas se sitúan en torno a un 55 % de reducción y son similares a las obtenidas con los algoritmos genéticos. La tasa más elevada (a excepción del algoritmo `SFS` que por su naturaleza *greedy* obtiene tasas más altas) se corresponde con la optimización del 10 % de individuos seleccionados aleatoriamente en la base de datos `Arrhythmia`. En esta base de datos se suelen obtener una mayor tasa de acierto cuando la reducción es elevada, y es lo que se aprecia en la tabla de este algoritmo, donde la ejecución con la máxima reducción es la que tiene mayor tasa de acierto. En cambio vemos que en media aunque este algoritmo tenga una mayor tasa de reducción, la tasa de acierto es similar a la de los otros algoritmos meméticos.

6.3.4. Tiempos

DB	0	1	2	3	4
TW	0.0144	0.2664	90.7194	87.3304	82.8314
TL	0.0334	1.6574	153.5612	155.3655	156.2545
TA	0.1033	3.9018	691.9667	596.6445	689.7578



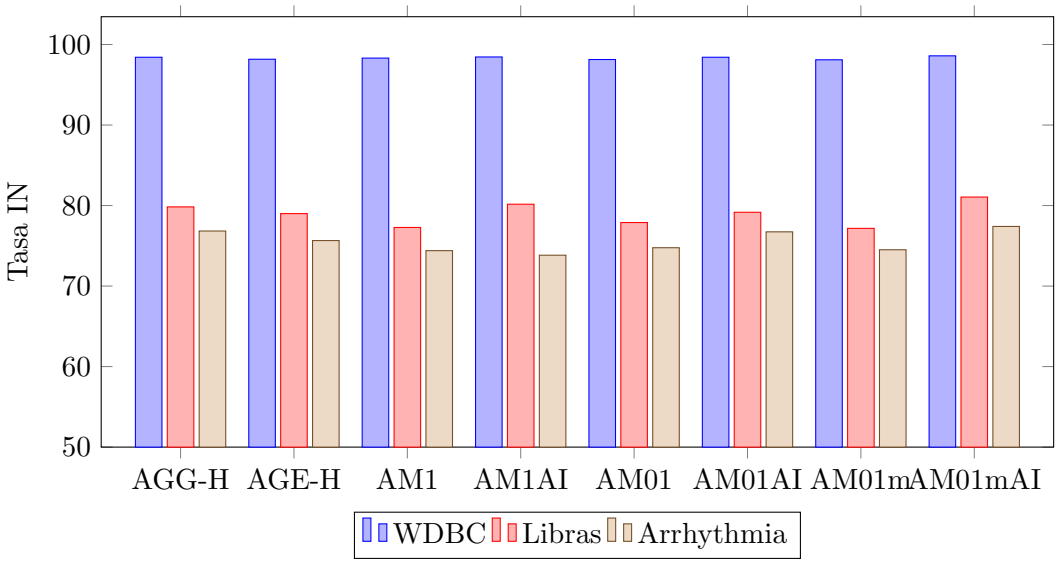
No se han incluido los tiempos para `KNN` ni `SFS` debido a que no son relevantes con respecto a los algoritmos meméticos. Como se ha comentado previamente, en las bases de datos pequeñas los tiempos son muy similares y es en `Arrhythmia` donde la ejecución es más rápida para el algoritmo con optimización en un 10 % aleatorio. Aunque los algoritmos están implementados para que se pueda probar con diferentes parámetros, teniendo en cuenta que la población es de tamaño 10 y que en el tercer algoritmo sólo se optimiza el 10 % mejor, es decir, el mejor individuo, podríamos haber seleccionado el máximo de la población y haber evitado reordenar la población, con lo que hubiese tardado menos.

6.4. Comparación algoritmos poblacionales

En este apartado se comparan todos los algoritmos poblacionales implementados, tanto los genéticos de la práctica 3, como los meméticos que se piden en la práctica y la variante de los meméticos con la búsqueda local de alta intensidad, donde de la búsqueda local no se realiza sólo una iteración sino que se ejecuta hasta se alcanza un óptimo local.

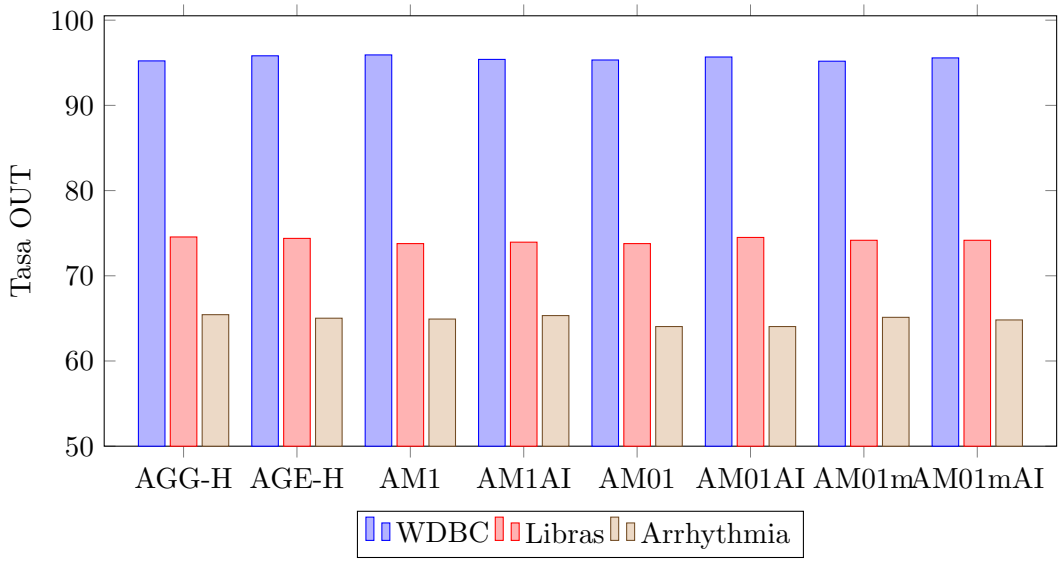
	WDBC				Movement Libras				Arrhythmia			
	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T	%Clas. in	%Clas. out	%red.	T
AGG-H	98.4185	95.2200	48.0000	86.5553	79.8333	74.5556	56.4444	151.5060	76.8385	65.4360	69.2885	535.8592
AGE-H	98.1725	95.8179	34.6667	121.4294	79.0000	74.3889	44.8889	198.5784	75.6508	65.0210	29.6443	950.9298
AM1	98.3129	95.9222	52.0000	90.7194	77.2778	73.7778	53.1111	153.5612	74.3997	64.9216	55.5336	691.9667
AM1AI	98.4538	95.3954	56.6667	89.5388	80.1667	73.9444	54.0000	169.1009	73.8332	65.3302	51.4229	944.9642
AM01	98.1373	95.3259	55.3333	87.3304	77.8889	73.7778	54.3333	155.3655	74.7627	64.0437	64.9012	596.6445
AM01AI	98.4188	95.6767	56.0000	87.1437	79.1667	74.5000	54.2222	154.2332	76.7359	64.0432	57.5494	679.6595
AM01m	98.1023	95.1850	59.6667	82.8314	77.1667	74.1667	54.0000	156.2545	74.5108	65.1235	55.8498	689.7578
AM01mAI	98.5943	95.5709	58.6667	81.0817	81.0556	74.1667	57.0000	153.0862	77.4060	64.8169	53.6759	699.0004

6.4.1. Tasa In



En cuanto a la tasa de acierto dentro de la muestra, vemos que se obtiene de manera general una mejora de los algoritmos meméticos cuando se realiza la búsqueda local con una alta intensidad. En cuanto a la comparación de los algoritmos genéticos con los meméticos, vemos que el algoritmo genético generacional tiene unas tasas ligeramente superiores a las modificaciones con BL de baja intensidad, por lo que podríamos pensar en que es necesaria una mayor intensidad, quizá decrementando el número de generaciones que hay que esperar para que se realice la búsqueda local. El algoritmo con mejores resultados es el que realiza optimización local intensiva sobre los mejores individuos de la población.

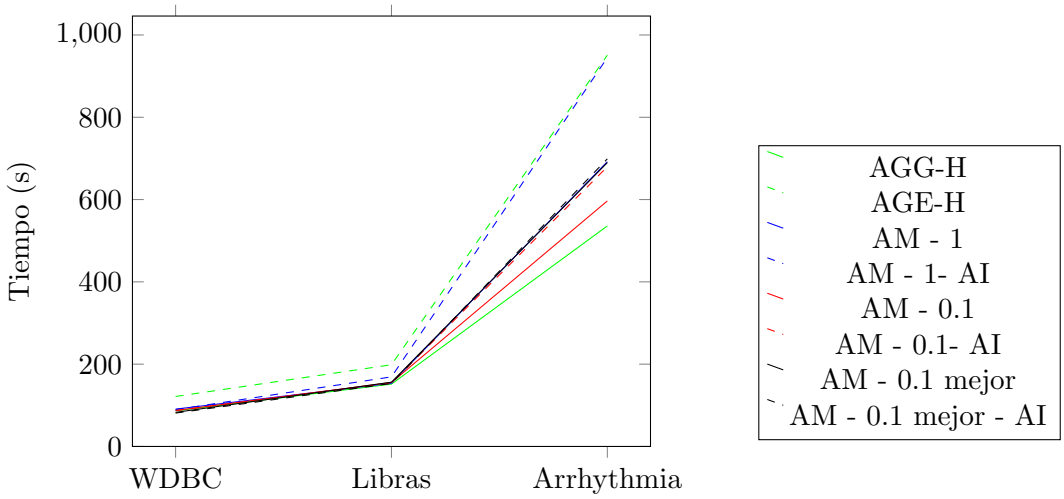
6.4.2. Tasa Out



En cuanto a la tasa de acierto fuera de la muestra se observa que todos los resultados son muy similares, con lo que no podemos realizar un análisis mayor que decir que los algoritmos con mejores resultados dentro de la muestra no se ven resentidos fuera y por tanto no hay un sobreajuste excesivo.

6.4.3. Tiempos

DB	0	1	2	3	4	5	6	7
TW	86.5553	121.4294	90.7194	89.5388	87.3304	87.1437	82.8314	81.0817
TL	151.5060	198.5784	153.5612	169.1009	155.3655	154.2332	156.2545	153.0862
TA	535.8592	950.9298	691.9667	944.9642	596.6445	679.6595	689.7578	699.0004



En las bases de datos de menor tamaño los tiempos son muy similares, a excepción del algoritmo genético estacionario (recordemos que todos los algoritmos restantes tienen un esquema generacional) que es un poco superior. En cuanto a la base de datos `Arrhythmia`, los algoritmos con BL de alta intensidad tardan más, con especial relevancia para el algoritmo con optimización en todos los individuos de la población, que tarda un tiempo semejante al genético estacionario. Donde menos se nota es en el algoritmo memético con optimización del mejor individuo, donde, al estar ya cerca de un óptimo local, esta intensificación es menos costosa en tiempo (tenemos que tener en cuenta que podríamos hacer más rápido este algoritmo para este caso concreto si en lugar de ordenar y coger sólo uno buscásemos directamente el mejor individuo). Vemos también que el algoritmo genético generacional es el que menos tarda y sin embargo el algoritmo da buenos resultados, lo que nos muestra el buen desempeño de los algoritmos genéticos para este problema incluso antes de introducir la búsqueda local.

7. Bibliografía

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