Práctica 2 Metaheurísticas.
Búsquedas multiarranque para el problema de la selección de características

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

12 de mayo 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. Búsqueda Multiarranque Básica
2. GRASP
3. ILS

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

3. Estructura del método de búsqueda **7**
3.1. Búsqueda multiarranque básica 7
3.2. GRASP 7
3.3. ILS 9

4. Algoritmo de comparación **9**

5. Procedimiento para desarrollar la práctica **10**
5.1. Ejecución del programa 11

6. Experimentos y análisis de resultados **12**
6.1. Descripción de los casos 12
6.2. Resultados 12
6.2.1. KNN 12
6.2.2. SFS 13
6.2.3. Búsqueda multiarranque básica 13
6.2.4. GRASP 14
6.2.5. ILS 14
6.2.6. Comparación 15
6.3. Análisis de los resultados 15
6.3.1. Tasa In 15
6.3.2. Tasa Out 16
6.3.3. Tasa reducción 17
6.3.4. Tiempos 17

7. Bibliografía **18**

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. Esta representación sí es adecuada si tenemos restricciones sobre el número de características tomadas ya que no podemos hacerlo con más de m características.

- Representación de orden: Cada solución es una permutación de n elementos, ordenados según la importancia de cada característica. Aquí también se maneja el cumplimiento de restricciones pues una vez encontrada la solución, tomaremos sólo las primeras m características.

Se ha de mencionar que en las dos últimas representaciones el espacio de soluciones es mayor que el espacio de búsqueda, justificado en la representación de orden porque da más información (podríamos tomar soluciones de longitud variable), pero que en la representación entera sólo es razonable asumir si tenemos una restricción de longitud fija. Además, otra ventaja de la representación binaria es la facilidad para aplicarle operadores (de vecindario, en posteriores prácticas de cruce...) manteniendo la consistencia.

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:∞} for i=1,...,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 ← Num. Process
   num_samples ← length(data_train)

   exit if sample > num_samples
   my_features ← data_train[sample]

   k_nearest ← {{item: -1, distance:∞} for i=1,...,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

Entenderemos como vecindario de una solución a los vectores que sólo difieren en una posición. Por tanto, el operador para movernos a una solución vecina consistirá en cambiar una posición determinada:

```
flip(solution, positon):
BEGIN
        neighbour ← copy(solution)
        actual_value ← solution_position
        neighbour_position ← NOT actual_value
        RETURN neighbour
END
```

En esta práctica también es común a todos los métodos el método de búsqueda local. En primer lugar, para favorecer la reducción de características, probé a considerar como mejor solución aquella que con la misma calidad en términos de porcentaje de aciertos, tuviera un menor número de características seleccionadas. Sin embargo, esto hacía incrementar notablemente el tiempo de búsqueda, especialmente en la base de datos `Arrhythmia`, pues al haber tantas características ocupaba gran parte del tiempo de la búsqueda local en reducir el número de seleccionadas en lugar de intentar acceder a posiciones mejores. Sí consideraremos que éstas son mejores soluciones, o mejor dicho, comprobaremos que son mejores soluciones fuera de la búsqueda local.
Como diferencia con respecto a la primera práctica, ahora la solución inicial es un parámetro y no se genera aleatoriamente dentro del método, ya que depende del algoritmo de búsqueda por trayectorias múltiples que usemos:

```
localSearch(data, categories, solution) BEGIN
   REPEAT
      first_neig ← random{1,..., num_features}
      neighbours ← {flip(solution,i): i=first_neig,...,first_neig-1}
      found_better ← FALSE

      FOR neig IN neighbours
         IF neigh is better than solution THEN
            found_better ← TRUE
            solution ← neig
            BREAK
      END
   WHILE( found_better )
```

```
    return solution
END
```

Donde que el vecino sea mejor que la solución actual significa que tiene una mayor tasa de acierto en el conjunto de datos de entrenamiento.

3. Estructura del método de búsqueda

3.1. Búsqueda multiarranque básica

Este método consiste en la realización de diferentes búsquedas locales que parten de soluciones aleatorias. Para ello, simplemente se lanzan, como se indica en el guión de prácticas, 25 búsquedas locales y devolvemos la mejor solución encontrada. El criterio para considerar que una solución es mejor que otra se basa, como se ha dicho previamente, en encontrar el vector de características que ofrezca una mejor tasa de acierto y, de entre ellas, la que tenga menos características seleccionadas. Los resultados esperados serán similares a las mejores ejecuciones del apartado de búsqueda local en la práctica anterior. Se podrá observar también si el hecho de incluir aceptar como mejores soluciones con igual tasa de acierto y menor número de características beneficia o perjudica a la tasa de acierto fuera de los datos de entrenamiento.

```
BMB(data, categories) BEGIN
    num_features ← length ( data[0] )
    best_solution ← {F,...,F: size = num_features}

    FOR i = 1,...,25
        solution ← {random({True,False}), size = num_features}

        solution ← localSearch(data, categories, solution)
        IF solution is better than best_solution THEN
            best_solution ← solution
    END

    return best_solution
END
```

Aquí el criterio para decir que es mejor sí hace referencia al número de características seleccionadas.

3.2. GRASP

El esquema general del algoritmo `GRASP` es similar al de `BMB`, con la única salvedad de que la solución inicial para cada una de las iteraciones no es un vector aleatorio sino que se trata de una solución aportada por un algoritmo *greedy*. Esto repercute en que es necesario un tiempo mayor de ejecución al ser más costoso lógicamente ejecutar el algoritmo *greedy*

que obtener una solución aleatoria. Sin embargo, este tiempo no se pierde en vano, pues resulta lógico pensar *a priori* que si partimos de una buena solución y entonces aplicamos búsqueda local, se llegará a una solución mejor que partiendo de una aleatoria.

```
GRASP(data, categories) BEGIN
   num_features ← length ( data[0] )
   best_solution ← {F,...,F: size = num_features}

   FOR i = 1,...,25
      solution ← getGreedySolution(data, categories)

      solution ← localSearch(data, categories, solution)
      IF solution is better than best_solution THEN
         best_solution ← solution
   END

   return best_solution
END
```

El interés de este método reside, pues, en la obtención de la solución *greedy*. Aquí también hay diferencias con respecto a la primera práctica. Ahora no se trata de ir seleccionando la mejor característica cada vez, sino de ir seleccionando una característica al azar de las que sobrepasen un umbral. Esto añade mayor variabilidad al espacio de búsqueda explorado, con lo que tenemos mayor probabilidad de encontrar una mejor solución.

```
getGreedySolution(data, categories) BEGIN
   solution ← {F,...,F: size = num_features}
   tolerance ← 0.3

   REPEAT
      char ← getCharWithThreshold(data,categories, solution,tolerance)
      exists_benefit ← flip(solution, char) is better than solution
      IF exists_benefit THEN
         solution ← flip(solution, char)

   WHILE exists_benefit

   RETURN solution
END
```

Aún falta por comentar cómo se obtiene la característica con la que seleccionaremos una característica al azar de entre las que cumplan una cierta condición, que consiste en tener un porcentaje de acierto entre el máximo valor del vecindario y éste menos un 30 % de la diferencia entre él y el menor valor del vecindario. El porcentaje de la diferencia aceptada es lógicamente variable, con un valor cercano a 0 % se trataría del *greedy* `SFS`, mientras que con un valor del 100 % se trata de una selección de un vecino aleatorio, y parando el algoritmo cuando seleccionásemos a un vecino peor que la solución actual.

```
getCharWithThreshold(data, categories,solution,tolerance) BEGIN
   values ← vector( size = num_features )
```

```
   FOR i = 1,..., num_features
      neig ← neighbours[i]
      values[i] ← score(data, categories, neig)
   END

   max ← max(values)
   min ← min(values)
   above_threshold ← {i ∈ {1,...,N}: values[i] > max-tolerance(max-min)}
   char ← random({above_threshold})

   RETURN char
END
```

3.3. ILS

Para la búsqueda local iterada, se trata también de realizar 25 iteraciones al igual que en los métodos anteriores. La diferencia ahora está en la solución con la que se parte en cada iteración. En la primera iteración se parte de una solución aleatoria. Para las demás, se parte de una mutación de la mejor solución encontrada hasta el momento. Las mutaciones consisten en cambiar el valor situado en un tanto por ciento de las características, seleccionando aleatoriamente las características a cambiar.

```
ILS(data, categories) BEGIN
   num_features ← length ( data[0] )
   best_solution ← {F,...,F: size = num_features}
   solution ← random({True,False}, size = num_features)

   FOR i = 1,...,25
      IF solution is better than best_solution THEN
         best_solution ← solution

      solution ← mutateSolution(best_solution)
   END

   return best_solution
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 in 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 di 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,B,G,I` que se corresponden con KNN, *greedy* SFS, búsqueda multiarranque básica, *GRASP* e *iterated local search*, 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 Practica2.py base_de_datos [-a algoritmo -seed semilla -write T/F ]
```

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

```
python Practica2.py W -a G -seed 123456
```

Si optamos por la base de datos `Arrhythmia` con la búsqueda multiarranque básica y guardar el resultado en un fichero:

```
python Practica2.py A -a B -write True
```

Para mostrar la introducción de parámetros:

```
python Practica2.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. Reduje el número de características eliminando las columnas que tuvieran el mismo valor para todos los datos. Hice esto cuando tenía en la búsqueda local que se aceptasen soluciones iguales pero con menor número de características. Como he comentado, tardaba mucho así que pensé que en `Arrhythmia`, debido al gran número de datos y columnas, para cada búsqueda se gastaba demasiado tiempo en descartar estas 24 características cuando estuviesen seleccionadas. Sin embargo, debido a que buena parte del resto de columnas tienen valores parecidos para muchos de los elementos, el tiempo seguía siendo excesivo, con lo que descarté esa modificación de la búsqueda local.

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 el de la primera práctica. 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 algoritmo `SFS` también se realizó el análisis en la práctica anterior, lo que nos permite comparar los tiempos entre haber usado el módulo de `scikit` y la paralelización realizada en esta práctica, siendo ahora en media más de 150 veces más rápido, para la base de datos de `Arrhythmia`, en otras BD y algoritmos la diferencia de tiempos puede seguir otra proporción.

6.2.3. Búsqueda multiarranque básica

	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	60	9,0498	75,5556	74,4444	56,6667	50,9651	77,8351	64,0625	50,1976	737,8035
Particion 1-2	99,6479	93,3333	40	8,9142	78,8889	76,1111	52,2222	44,4958	70,3125	62,3711	50,1976	731,1191
Particion 2-1	98,2456	96,831	66,6667	9,4357	80	66,6667	48,8889	56,6316	73,7113	66,1458	52,1739	844,0765
Particion 2-2	97,8873	95,7895	43,3333	8,9942	74,4444	77,7778	53,3333	51,0907	74,4792	66,4948	56,917	880,9571
Particion 3-1	97,8947	94,7183	66,6667	8,577	68,8889	75,5556	41,1111	47,8755	70,6186	65,625	56,917	670,4829
Particion 3-2	98,2394	96,4912	60	7,3264	79,4444	76,6667	47,7778	49,9642	73,9583	65,9794	54,1502	898,8429
Particion 4-1	98,9474	95,0704	60	8,724	78,8889	72,7778	62,2222	45,1795	69,5876	63,5417	56,1265	636,5153
Particion 4-2	97,8873	94,386	60	8,7536	78,8889	71,6667	58,8889	49,5447	75	67,0103	47,4308	929,0148
Particion 5-1	97,5439	94,7183	56,6667	9,0287	80	70	57,7778	54,1461	76,2887	65,1042	50,9881	713,8755
Particion 5-2	98,9437	95,0877	50	8,8832	77,2222	75	50	51,6029	72,9167	66,4948	58,498	805,7788
Media	98,3132	95,2905	56,3333	8,7687	77,2222	73,6667	52,8889	50,1496	73,4708	65,283	53,3597	784,8466
Max	99,6479	96,831	66,6667	9,4357	80	77,7778	62,2222	56,6316	77,8351	67,0103	58,498	929,0148
Min	97,5439	93,3333	40	7,3264	68,8889	66,6667	41,1111	44,4958	69,5876	62,3711	47,4308	636,5153
Desv. Típica	0,6242	1,0425	8,6217	0,5288	3,2961	3,2413	5,9255	3,5198	2,5325	1,4248	3,4953	96,011

En la búsqueda multiarranque básica vemos cómo llegamos a valores de la tasa de acierto (en especial dentro de la muestra) a las que no llegamos en la práctica anterior. Si observamos los resultados de la búsqueda local de la práctica anterior se comprueba que las mayores tasas de clasificación de entrenamiento de entre las diferentes particiones son

similares a la media de las tasas obtenidas ahora. Y es que (aún siendo distintas particiones) si en la práctica anterior se realizaron 10 búsquedas locales por BD, ahora estamos haciendo 25 por partición.

6.2.4. GRASP

	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	95,0704	46,6667	10,6061	76,6667	70,5556	81,1111	35,2843	85,0515	73,9583	92,8854	191,9178
Particion 1-2	99,2958	93,3333	43,3333	10,976	81,6667	75,5556	74,4444	47,3416	78,6458	72,1649	87,3518	199,0155
Particion 2-1	97,5439	96,1268	43,3333	11,9668	83,8889	70,5556	58,8889	57,6303	80,4124	71,3542	90,9091	193,9727
Particion 2-2	97,5352	95,0877	40	12,3856	75,5556	75,5556	64,4444	52,4627	80,2083	72,1649	84,1897	252,4235
Particion 3-1	97,5439	95,7747	26,6667	11,8944	70	76,1111	58,8889	54,8985	80,9278	70,8333	78,2609	236,8755
Particion 3-2	97,8873	96,1404	16,6667	11,098	80,5556	80,5556	57,7778	54,1712	80,2083	71,134	78,2609	327,0603
Particion 4-1	98,5965	95,0704	30	12,4167	81,6667	77,7778	50	59,3037	78,866	73,4375	76,2846	307,5072
Particion 4-2	97,8873	96,1404	36,6667	12,4423	84,4444	69,4444	54,4444	66,364	82,8125	69,0722	70,751	585,1382
Particion 5-1	97,5439	95,7747	30	12,0175	79,4444	73,8889	47,7778	67,0545	80,4124	68,2292	68,7747	466,6614
Particion 5-2	98,9437	96,1404	26,6667	12,0377	78,3333	76,1111	48,8889	58,8591	79,1667	66,4948	67,9842	629,5999
Media	98,0672	95,4659	34	11,7841	79,2222	74,6111	59,6667	55,337	80,6712	70,8843	79,5652	339,0172
Max	99,2958	96,1404	46,6667	12,4423	84,4444	80,5556	81,1111	67,0545	85,0515	73,9583	92,8854	629,5999
Min	97,5352	93,3333	16,6667	10,6061	70	69,4444	47,7778	35,2843	78,6458	66,4948	67,9842	191,9178
Desv. Típica	0,6129	0,8364	9,0431	0,6222	4,1066	3,343	10,4119	8,7644	1,8446	2,2166	8,5471	155,7725

De los resultados de GRASP, teniendo en cuenta que parte de una solución *greedy*, llama la atención la poca reducción de las características en `WDBC`, aunque si vemos que la tasa de acierto con el `KNN` es tan alta, ha podido ir tomando características de una en una de aquellas mejores que un umbral obteniendo siempre mejores resultados. En cambio para `Arrhythmia`, la reducción sí es mayor, fruto de que haya tantas características y no todas tengan realmente influencia.

6.2.5. ILS

	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	98,2456	95,0704	43,3333	7,8924	76,1111	76,6667	58,8889	51,314	80,9278	61,9792	50,9881	742,3719
Particion 1-2	99,2958	94,386	43,3333	7,2976	80	75	50	46,141	72,9167	62,3711	53,3597	675,2898
Particion 2-1	97,8947	97,1831	56,6667	6,6382	80,5556	66,1111	52,2222	45,359	75,2577	66,1458	56,1265	678,0366
Particion 2-2	98,5916	95,0877	53,3333	7,4998	78,3333	75,5556	55,5556	52,9768	76,5625	67,0103	51,7787	689,171
Particion 3-1	98,5965	95,7747	73,3333	5,263	73,8889	74,4444	62,2222	47,8783	75,2577	64,5833	58,1028	595,518
Particion 3-2	98,2394	96,8421	53,3333	7,2592	80,5556	78,8889	52,2222	42,0196	75	67,5258	53,3597	718,6257
Particion 4-1	98,9474	94,3662	43,3333	7,4867	82,2222	70	60	48,8344	72,6804	64,0625	55,336	639,662
Particion 4-2	97,8873	95,0877	56,6667	9,0106	82,2222	71,6667	62,2222	45,5346	79,1667	63,9175	55,7312	671,2428
Particion 5-1	97,5439	95,0704	60	7,572	80,5556	68,8889	63,3333	43,4099	78,3505	65,1042	57,7075	630,6454
Particion 5-2	98,9437	96,1404	53,3333	6,8579	81,1111	76,1111	63,3333	44,6671	76,5625	67,5258	54,5455	724,5674
Media	98,4186	95,5009	53,6667	7,2777	79,5556	73,3333	58	46,8135	76,2683	65,0226	54,7036	676,5131
Max	99,2958	97,1831	73,3333	9,0106	82,2222	78,8889	63,3333	52,9768	80,9278	67,5258	58,1028	742,3719
Min	97,5439	94,3662	43,3333	5,263	73,8889	66,1111	50	42,0196	72,6804	61,9792	50,9881	595,518
Desv. Típica	0,5263	0,9141	8,7496	0,9058	2,5556	3,8006	4,8381	3,2686	2,4976	1,9073	2,2373	43,167

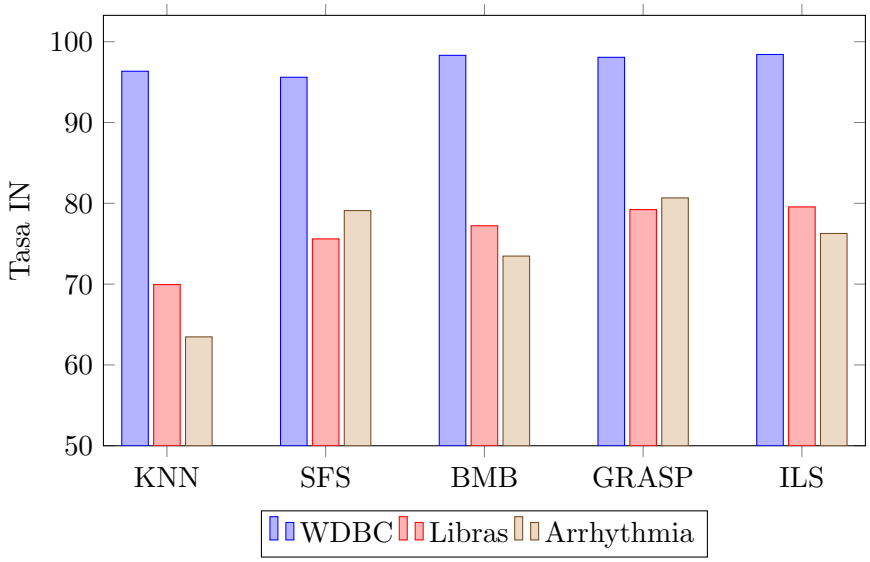
En `ILS` vemos que se obtienen resultados similares a los de `BMB`, sólo un poco mejores. Esto se traduce en que pese a disponer un algoritmo simple para explorar vecindarios mayores a los considerados hasta ahora, el resultado está mejor dirigido que generar soluciones completamente aleatorias y aplicar búsqueda local.

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
BMB	98.3132	95.2905	56.3333	8.7687	77.2222	73.6667	52.8889	50.1496	73.4708	65.2830	53.3597	784.8466
GRASP	98.0672	95.4659	34.0000	11.7841	79.2222	74.6111	59.6667	55.3370	80.6712	70.8843	79.5652	339.0172
ILS	98.4186	95.5009	53.6667	7.2777	79.5556	73.3333	58.0000	46.8135	76.2683	65.0226	54.7036	676.5131

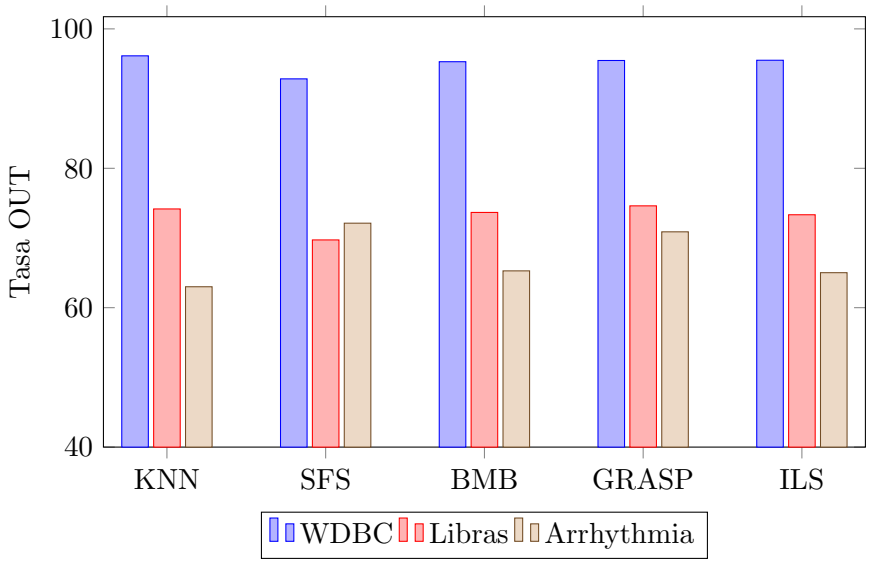
6.3. Análisis de los resultados

6.3.1. Tasa In



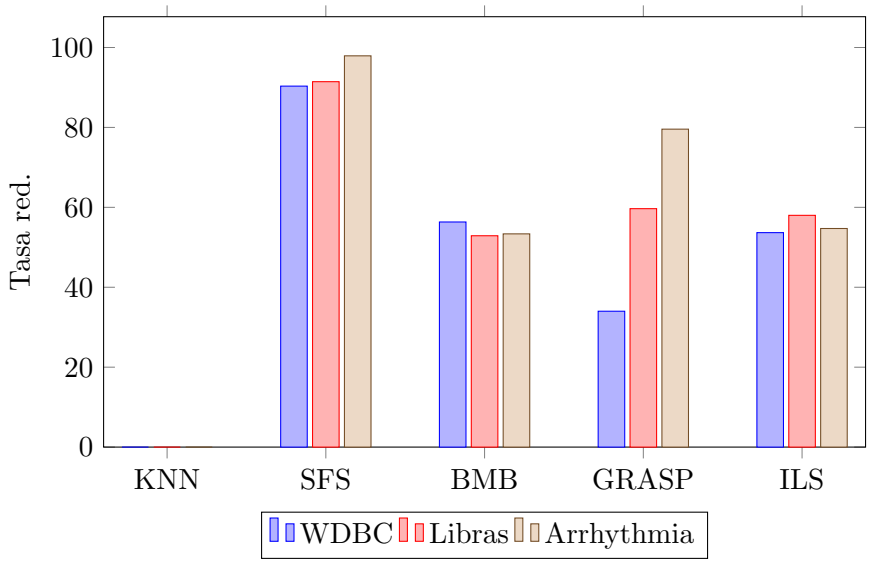
En la tasa de acierto en la muestra de entrenamiento se aprecia una mejora en general de los algoritmos de esta práctica con respecto a los algoritmos de comparación (sólo en `Arrhythmia` `SFS` consigue una mejora con respecto a `BMB` e `ILS`). La mejor tasa de acierto se obtiene en `WDBC` y `Libras` con `ILS`, sin embargo en término medio el mejor es `GRASP`, pues para `Arrhytmia` obtiene unos resultados mejores sí con una cierta distancia y sin embargo, en las otras dos BD los resultados son sólo un poco peores. El mejor desempeño en esta base de datos se puede deber a que el número de características es mayor, con lo que resulta efectivo partir de la solución sin características seleccionadas y encontrar máximos locales, que serán soluciones donde sólo cuenten las características más relevantes.
`BMB` queda entonces como el algoritmo menos efectivo, pues, al encontrar óptimos locales en cada una de las veinticinco iteraciones desestima seguir buscando por el entorno de ese óptimo local y pega un salto. Sería interesante combinar `BMB` e `ILS`, realizando por ejemplo 5 búsquedas locales realizadas a partir de soluciones aleatorias, y por cada trayectoria, lanzar 5 búsquedas mutando la mejor solución encontrada en cada trayectoria.

6.3.2. Tasa Out



En la tasa de acierto en los datos de test resulta que considerar todas las características tiene un buen valor para `WDBC` y `Libras`. Sin embargo no consideraríamos esta opción en una situación real pues no estaríamos reduciendo el número de características, y como se observa en `Arrhythmia`, una base de datos con mayor número de clases y características, los resultados son bastante peores. De entre los demás algoritmos, es de nuevo `GRASP` el que obtiene los mejores resultados en término medio para las tres bases de datos, destacando especialmente en `Arrhythmia`.

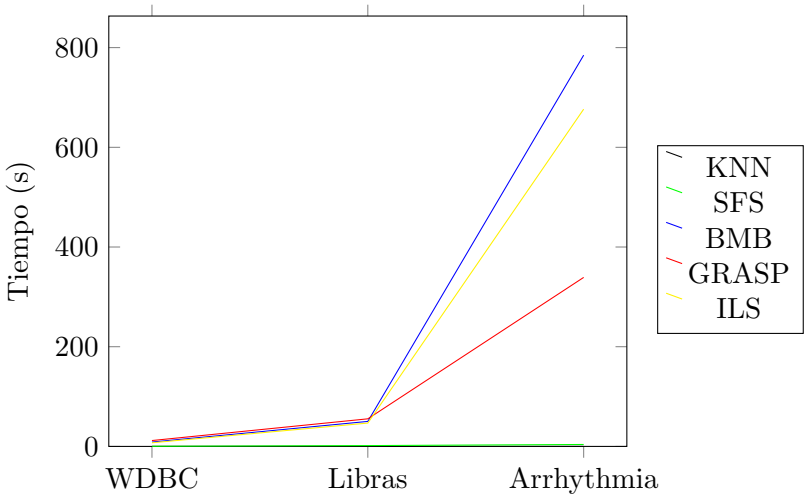
6.3.3. Tasa reducción



Con la tasa de reducción se puede entender mejor la tasa de acierto. Se observa que en `Arrhythmia` se obtiene una mayor tasa de acierto en los datos de test a mayor tasa de reducción y en cambio para `WDBC` a menor tasa de reducción mayor acierto fuera de la muestra. Es relevante la tasa de reducción en `Arrhythmia` en `GRASP` debido a que los otros algoritmos empezaron con una solución aleatoria y se han quedado en óptimos locales que tenían en torno a un 50 % de reducción y en cambio `GRASP`, empezando sin ninguna característica, ha parado antes obteniendo mejores resultados. Para las otras bases de datos este algoritmo también ha llegado a valores cercanos a los de los otros algoritmos, con lo que se deduce que es una buena estrategia de búsqueda.

6.3.4. Tiempos

DB	0	1	2	3	4
TW	0.0144	0.2664	8.7687	11.7841	7.2777
TL	0.0334	1.6574	50.1496	55.3370	46.8135
TA	0.1033	3.9018	784.8466	339.0172	676.5131



Observamos en la gráfica de tiempos que los tiempos son similares en los tres algoritmos de esta práctica para las dos primeras bases de datos, con `GRASP` un poco más lento. En cambio, `GRASP` es con diferencia es más rápido en `Arrhythmia` al quedarse con una buena solución muy pronto (alta tasa de reducción de características). También hay que destacar que, teniendo mejores resultados, `ILS` es más rápido que `BMB`, pues con cada mutación nos quedamos más cerca del óptimo que generando una solución aleatoria.

7. Bibliografía

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