Prácticas - Metaheurísticas

Selección de características.

José Carlos Entrena Jiménez

Índice

1. Descripción del problema

La selección de características es un problema generalmente tratado en el ámbito del *data mining*, que intenta resolver la relevancia de cómo ciertos atributos de un conjunto de datos a la hora de dividirlos en distintas clases. El objetivo es hacer que los atributos considerados irrelevantes a la hora de clasificar los datos (esto es, que no influyan, o incluso influyan negativamente en la clasificación) sean eliminados del conjunto de datos, suponiendo tres ventajas fundamentales:

- Ahorro en espacio, al poder eliminar los valores de estas características de nuestro conjunto de datos.
- Ahorro computacional, pues la clasificación tendrá que considerar menos características. reduciendo lo tiempo de ejecución.
- Ahorro en la toma de medidas, ya que atributos sin relevancia no tendrán que ser analizados en nuevas recopilaciones de datos.

Hecha esta diferenciación, consideraremos los atributos relevantes para la clasificación a la hora de realizar predicciones futuras.

Realizando un enfoque más particularizado al problema que nos atañe, disponemos de tres conjuntos de datos: Winconsin Database Breast Cancer (en lo sucesivo, WDBC), conjunto de datos que clasifican un tumor conforme a 30 características, Movement_Libras (en lo sucesivo, Libras), que clasifica los movimientos realizados con la mano en la Lengua de Signos Brasileña conforme a 90 características, y Arrhythmia, que distingue la ausencia o presencia de 4 tipos de arritmia cardiaca conforme a 278 características.

El objetivo es obtener un conjunto reducido de características relevantes para cada problema, mediante la aplicación de distintas metaheurísticas de trabajo, teniendo siempre en cuenta que la correcta clasificación de los datos es primordial.

2. Aplicación de los algoritmos

2.1. Representación de la solución

Notaremos las soluciones de forma binaria,con un array de ceros y unos. donde el uno represante que la característica es considerada, y el cero que es despreciada.

2.2. Operador de vecino

Como operador de vecino cambiaremos la relevancia de una característica: si no está considerada pasamos a considerarla, y viceversa.

2.3. Generación de vecinos

Para considerar el entorno completo, generaremos un array de enteros, que tomará todas las características disponibles salvo la clase, que no es relevante a la hora de clasificar pues no se considerará. Para saber la posición de la clase, pasaremos un parámetro al método clasificador, que le servirá para distinguir si está en la primera posición (como ocurre con WDBC) o en la última (como ocurre en los dos otros conjuntos).

Una vez generados los posibles cambios, los barajaremos para considerar el entorno. Hacemos esto para evitar considerarlos por orden y dar más prioridad a las características en una posición más cercana a la primera. Cabe destacar que este barajado se realiza con una semilla aleatoria que es siempre la misma para cada heurística, por lo que nos aseguramos que el entorno es considerado siempre en el mismo orden (y en el caso de la búsqueda tabú, que tomamos los 30 primeros, es siempre igual).

2.4. Función objetivo

Hemos dividido cada conjunto de datos en 5 particiones con dos elementos, cada uno con el 50 % de los datos, que actuarán como conjunto de aprendizaje y conjunto de test alternativamente. Nuestra función objetivo será la tasa de aciertos de la clasificación del conjunto test utilizando el vector binario obtenido como solución del proceso de aprendizaje, y buscaremos maximizar este dato.

3. Pseudocódigo de los algoritmos

3.1. Práctica 1

3.1.1. KNN

El algoritmo k-NN aplicado sobre un conjunto de datos y con un elemento de este conjunto como entrada, nos devuelve una predicción de la clase de dicho elemento, correspondiente a la clase mayoritaria de los k elementos más cercanos.

En nuestro caso particular, iremos calculando las distancias con todos los otros elementos del *dataset*, si bien hemos efectuado una ligera modificación: Vamos almacenando las distancias a los elementos más cercanos al nuestro, y si en el calculo de una nueva distancia (que se realiza característica a característica, de forma iterativa) se supera la k-ésima mayor, abortamos y no consideramos el elemento correspondiente.

Una vez obtenidos los elementos cercanos (que nos aseguramos que son un mínimo de k), tomamos la clase mayoritaria de los k primeros. En caso de empate, tomaremos aquella correspondiente al elemento con menor distancia al nuestro.

3.1.2. Clasificador (función objetivo)

Tomaremos todos los elementos del conjunto, y uno a uno, calcularemos su clase mediante el 3NN. Si coincide con clase del elemento, sumamos uno al número de clasificados correctamente. Devolvemos el porcentaje de aciertos.

Algorithm 1 Clasificador

```
Correctos ← 0
for all element in dataset do
    class ← 3NN(elemento)
    if class = trueClass then
        Correctos ← Correctos + 1
    end if
end for
Devolver Correctos/Número de elementos
```

3.1.3. SFS

Partiremos de una solución inicial con todas las características a 0, e iremos añadiendo aquella que más mejore el porcentaje de clasificación. Una vez que ninguna característica mejore, pararemos y devolveremos la solución que tengamos en ese momento.

Pseudocódigo del algoritmo:

Algorithm 2 Greedy

```
solucion ← [0, ..., 0]
parar ← false
while !parar do
    mejor_índice ← −1
    for all cambionoañadido do
        Añadir cambio
        Clasificar con la nueva solución
        if La nueva clasificación mejora la actual then
            Actualizar mejor índice y mejor clasificación
        end if
        Quitar cambio
    end for
    if Alguna solución mejora then
        Actualizar mejor solución
    else
        parar ← true
    end if
end while
Devolver mejor solución
```

3.1.4. Búsqueda Local

Crearemos una solución inicial aleatoria, e iremos aplicando todos los posibles cambios (usando el operador de vecino) hasta que encontremos uno que mejore la solución. Si hemos explorado todo el vecindario sin mejora o hemos alcanzado el máximo de iteraciones, paramos y devolvemos la solución que tengamos en ese momento.

Pseudocódigo del método de exploración del entorno:

Algorithm 3 Búsqueda local

```
continuar ← true
while continuar do
    continuar ← false
    Barajamos el array de cambios
    for all cambio ∈ arraydecambios do
        iteraciones ← iteraciones + 1
        Solucionlocal = Mejorsolucion
        Cambiamos el valor de la solución local que indica cambio
        nuevovalor = Clasificar(Solucionlocal)
        if nuevovalor > mejorvalor then
            Actualizamos mejor solución
            if iteraciones < limite then
                continuar ← true
            end if
            Salir del for
        end if
    end for
end while
Devolver mejor solución
```

3.1.5. Enfriamiento Simulado

Al igual que en búsqueda local, tendremos generado todo el vecindario, sobre el que iremos considerando elementos para efectuar cambios en la solución local dentro de cada iteración (enfriamiento). Generaremos vecinos hasta que encontremos uno que mejore o cumpla la condición Metrópolis, en cuyo caso actualizaremos la solución local y la mejor solución en caso de ser necesario.

Una vez generado un número de vecinos prefijado o alcanzado un número de aciertos, enfriaremos la temperatura, pasando a una nueva iteración del bucle.

Pseudocódigo del cálculo de los parámetros iniciales:

Algorithm 4 Parámetros iniciales

$MaxVecinos \leftarrow 5 * nCaracteristicas$
$MaxAciertos \leftarrow 0{,}1 * MaxVecinos$
$Enfriamientos \leftarrow MaxIteraciones/MaxVecinos$
$TemperaturaFinal \leftarrow 0{,}001$
$Temperatura \leftarrow TemperaturaFinal/(0{,}9^{Enfriamientos})$

El esquema de enfriamiento seguido es un esquema geométrico de parámetro 0.9, ya que el proporcionado en el guión de prácticas hacía que la temperatura tomase un valor muy cercano a la temperatura final en las primeras iteraciones, lo que reduce el número de activaciones del criterio Metrópolis.

Pseudocódigo de la obtención de vecinos:

Algorithm 5 Obtención de vecinos

Creamos un array con los posibles cambios (todas las características salvo la clase).
$stop \leftarrow false$
while $temperatura > temperaturaFinal$ and $!stop$ **do**
  $aciertos \leftarrow 0$
  $generados \leftarrow 0$
  $index \leftarrow 0$
  while $generados < MaxVecinos$ and $aciertos < MaxAciertos$ **do**
    $solucionLocal \leftarrow solucionActual$
    Cambiamos $solucionLocal[cambios[index]]$
    $generados \leftarrow generados + 1$
    $nuevovalor = Clasificar(Solucionlocal)$
    if $nuevovalor > mejorvalor$ or $condicionMetropolis$ **then**
      $aciertos \leftarrow aciertos + 1$
      Actualizamos solución actual, y si es necesario, la mejor solución.
      $index \leftarrow 0$
      Barajamos array de cambios
    end if
    $index \leftarrow (index + 1) mod (numCambios)$
  end while
  if $aciertos = 0$ **then**
    $stop \leftarrow true$
  end if
  $temperatura \leftarrow temperatura * 0{,}9$
end while
Devolver mejor solución

3.1.6. Búsqueda Tabú

Para la búsqueda tabú hemos reducido el entorno a 30 vecinos, que generaremos de forma aleatoria en cada iteración. De estos 30 vecinos tomaremos el mejor de ellos siguiendo el esquema de la búsqueda tabú: solo elegimos un movimiento que es tabú conforme al criterio de aspiración; mejorar la mejor solución hasta el momento.

Una vez tenemos el mejor vecino, aplicamos el cambio y actualizamos la lista tabú, con dos posibles casos: Si el movimiento era tabú, lo ponemos al final de la cola de movimientos tabú. Si no lo era, sacamos un movimiento e insertamos el que acabamos de hacer.

Pseudocódigo del interior del bucle, con manejo de la lista tabú.

Algorithm 6 Búsqueda tabú

```
mejorIndice ← −1
valorActual ← −1
movTabu ← false
entorno ← barajar(Cambios).primeros(30)
for all elemento ∈ entorno do
    solucionLocal ← solucionActual
    Cambiamos solucionLocal[elemento]
    nuevoValor ← Clasificar(Solucionlocal)
    if nuevoValor > valorActual then
        if elemento no es tabú then
            movTabu ← false
            Actualizamos mejor índice
        else if nuevoValor > mejorValor then
            movTabu ← true
            Actualizamos mejor índice
        end if
    end if
end for
if mejorIndice ≠ −1 then
    Cambiamos el valor que indica la posición mejorIndice
end if
if valorActual > mejorValor then
    Actualizamos mejor solución
end if
if movTabu then
    Borramos mejorIndice de la lista tabú
else
    Sacamos el elemento más antiguo de la lista tabú
end if
ListaTabu.push(mejorIndice)
```

3.2. Práctica 2

3.2.1. Generación de soluciones aleatorias

Para generar soluciones aleatorias usaremos una función con una semilla aleatoria, que genera números aleatorios entre 0 y 1 y asigna 0 a una característica si el número es mayor que 0.5, y 1 si es menor, hasta completar el tamaño del vector de características.

Pseudocódigo:

Algorithm 7 Generación de soluciones aleatorias

```
solucion ← [0, . . . , 0]
for all Característica do
    if random > 0,5 then
        Característica ← 1
    end if
end for
Devolver solución
```

3.2.2. Búsqueda Local Multiarranque Básica

Generaremos n soluciones aleatorias y aplicaremos sobre ella la búsqueda local de la práctica 1.

Pseudocódigo:

Algorithm 8 Búsqueda local multiarranque

```
loop
    Generar nueva solución aleatoria, aplicar BL y clasificar
    if Valor > mejor_valor then
        Actualizar mejor solución
    end if
end loop
Devolver mejor solución
```

3.2.3. GRASP

GRASP generará n soluciones *greedy* aleatorizadas, y sobre ellas aplicará la búsqueda local, quedándose con la mejor solución de las encontradas. A continuación podemos ver el pseudocódigo del algoritmo y de la generación de soluciones *greedy* aleatorizadas:

Pseudocódigo del algoritmo:

Algorithm 9 GRASP

```
loop
    Generar nueva solución greedy aleatorizada, aplicar BL y clasificar
    if Valor > mejor_valor then
        Actualizar mejor solución
    end if
end loop
Devolver mejor solución
```

Pseudocódigo de la generación de soluciones greedy aleatorizadas:

Algorithm 10 Generación de solución inicial

```
solucion ← [0, ..., 0]
parar ← false
while !parar do
    candidatos ← []
    μ ← max − α ∗ (max − min)
    for all cambio no añadido do
        Cambiar característica
        Clasificar con la nueva solución
        if valor >= μ then
            Añadir cambio a candidatos
        end if
        Restablecer característica
    end for
    if Candidatos es no vacío then
        Seleccionar un candidato de forma aleatoria
        if El candidato no mejora then
            parar ← true
        else
            Actualizar mejor valor
        end if
    end if
end while
Devolver solución
```

3.2.4. ILS

ILS toma una solución inicial aleatoria y aplica el algoritmo de búsqueda local sobre ella. Una vez optimizada al solución, genera una mutación de esta, sobre la que vuelve a aplicar búsqueda local, y compara la solución obtenida con la mejor solución hasta el momento. Si mejora, actualiza la mejor solución, volviendo a generar una mutación de esta mejor solución y repitiendo el proceso. Se repetirá tantas veces como iteraciones, quedándonos con la mejor solución final como valor de salida.

Se detalla el pseudocódigo del algoritmo y de la mutación de soluciones.

Algoritmo:

Algorithm 11 ILS

```
Generar solución aleatoria y aplicar búsqueda local
loop
    Mutar mejor solución y aplicar búsqueda local
    Clasificar nueva solución
    if Valor > Mejor_valor then
        Actualizar mejor solución
    end if
end loop
Devolver mejor solución
```

Mutación de solución:

Algorithm 12 Mutación

```
Tomamos n_mutaciones cambios aleatorios distintos
for all Cambio do
    Cambiar solución[cambio]
end for
Devolver solución modificada
```

3.3. Práctica 3

Nota: La generación de soluciones aleatorias ya se ha tratado en la práctica 2.

3.3.1. Operador de cruce

El operador de cruce es común para las dos variantes del algoritmo genético. En él, tomaremos dos soluciones padres y dos puntos de corte, que dividirán cada padre en tres trozos. Para generar cada hijo, tomaremos los extremos de un padre y la parte central del otro, alternando los papeles de cada padre para generar cada hijo.

Pseudocódigo:

Algorithm 13 Operador de cruce

```
Obtener par aleatorio de puntos de corte
Hijo1 ← Trozo1Padre1 + Trozo2Padre2 + Trozo3Padre1
Hijo2 ← Trozo1Padre2 + Trozo2Padre1 + Trozo3Padre2
```

3.3.2. Operador de mutación

El operador de mutación consiste en, seleccionados una solución y un gen (característica), cambiar su valor de 0 a 1 o viceversa. Si bien el cambio se realiza de la misma forma en las dos variantes, no se aplica el mismo criterio para decidir si la mutación se realiza o no.

En el caso de la versión generacional, la probabilidad de 0.1 % sobre el producto de características y número de elementos en la población produce un número cercano o mayor a 1, por lo que calcularemos el número de mutaciones, redondearemos al alza y aplicaremos siempre dicho número de iteraciones en lugar de calcular una probabilidad para cada gen, que sería costoso computacionalmente.

Para la versión estacionaria, la probabilidad de mutar uno de los dos hijos va a ser siempre menor o igual a uno en los tres conjuntos de datos, luego generaremos un número aleatorio entre 0 y 1, y si es menor que la probabilidad de mutación elegimos un gen en una solución al azar para mutarlo.

El pseudocódigo de la mutación es simple y es el siguiente:

Algorithm 14 Mutación

Obtener par de enteros aleatorio [solución. gen]
Cambiar $poblacion[solucion][gen]$

3.3.3. Mecanismo de selección

El operador de selección de soluciones utilizado es el torneo binario, que enfrenta a dos candidatos y se queda con el que tiene mejor valor de clasificación. Dicho comportamiento favorece la consideración como padres de las mejores soluciones, haciendo menos probable que las soluciones de peor calor se reproduzcan.

Este operador se ha utilizado únicamente en la versión generacional, siendo la selección completamente aleatoria en la elitista, es decir, sin torneo alguno. El cambio de selección es sencillo: en lugar de generar un índice aleatorio, se generan dos y nos quedamos con el elemento de la población que indique el índice que tenga mejor valor de clasificación. Esta elección se ha hecho para considerar dos casos de selección de padres e intentar sacar conclusiones sobre el uso o no de dicho torneo.

En nuestro algoritmo, la población estará siempre ordenada por valor de clasificación de mayor a menor. Por ser así, el torneo binario consiste únicamente en generar un par de enteros aleatorios entre 0 y el tamaño de la población, y quedarnos con el menor de ellos, que será el que referencie a la solución de mayor valor de clasificación.

3.3.4. Algoritmo genético generacional

En el caso generacional, en cada iteración generaremos una nueva población que sustituirá a la actual. Primero seleccionaremos a los posibles padres mediante torneo, y cruzaremos tantas parejas como el porcentaje de cruce nos indique. Los hijos reemplazarán a sus padres, y aquellos padres que no hayan sido utilizados irán directamente a la siguiente generación. Como además tenemos un esquema elitista, si la mejor solución de la anterior generación no se mantiene, eliminaremos la peor solución de la nueva y añadiremos esta.

Vemos el pseudocódigo del algoritmo a continuación:

Algorithm 15 Versión generacional

```
Generar población inicial aleatoria y clasificar
Ordenar población
while iteraciones <máximo do
    Crear nueva generación por torneo
    Realizar los cruces de padres y obtener hijos que los reemplazan
    Mutar las soluciones
    Clasificar la población y sumar las iteraciones pertinentes
    if No se mantiene la mejor solución anterior then
        Añadir la mejor solución anterior, eliminar la peor
    end if
    Ordenar población
end while
Devolver mejor solución de la población
```

3.3.5. Algoritmo genético estacionario

Para el caso estacionario, en cada iteración seleccionaremos únicamente dos padres, que se cruzarán y generarán dos hijos. Si cada uno de ellos es mejor que el peor de la población, pasará a reemplazarlo.

Aunque inicialmente la selección de cada padre debería hacerse mediante torneo binario (eligiendo dos, tomando el mejor), he decidido omitir dicho torneo para intentar comparar resultados con la otra versión con torneo. De esta forma, cada padre se toma aleatoriamente de la población, por lo que todos los elementos tienen la misma probabilidad de ser elegidos como padres.

Pseudocódigo del algoritmo:

Algorithm 16 Versión estacionaria

```
Generar población inicial aleatoria y clasificar
Ordenar población
while iteraciones <máximo do
    Elegir dos padres aleatorios
    Realizar los cruces de padres y obtener hijos
    if random < porcentaje_mutacion then
        Mutar un gen aleatorio de un hijo aleatorio
    end if
    Clasificar los hijos y sumar dos iteraciones
    if Un hijo es mejor que el peor de la población then
        Añadir el hijo, eliminar la peor solución
        Ordenar población
    end if
end while
Devolver mejor solución de la población
```

3.4. Práctica 5

El algoritmo genético generacional tratado en la práctica 3 no ha sido modificado, por lo que se mantiene su estructura. Al final de cada generación de una nueva población, comprobaremos si han pasado 10 iteraciones desde la última vez que aplicamos búsqueda local, para saber si tenemos que volver a hacerlo. Esto lo hacemos con una variable entera, introducida en el algoritmo.

El caso de interés es el esquema de búsqueda local. Dependiendo de qué versión de hibridación queramos utilizar, tenemos a nuestra disposición 3 métodos privados, que se encargarán de cada tipo de hibridación. Para cambiar entre una y otra solo hemos de cambiar el método al que llamamos. Todos estos métodos reciben los mismos parámetros, siendo el más importante la población actual del algoritmo genético.

A continuación detallamos el pseudocódigo de los tres métodos de hibridación:

3.4.1. Versión 1: hibridación total

Algorithm 17 Hibridación total

```
Crear nueva población vacía
for all elemento ∈ poblacion do
    Aplicar un iteración de BL a elemento
    Clasificar elemento, sumar una iteración
    Añadir [valor, elemento] a la nueva población
end for
Devolver la nueva población
```

3.4.2. Versión 2: hibridación aleatoria

Algorithm 18 Hibridación aleatoria

Copiar la población en una nueva población
Elegir una posición aleatoria
Aplicar un iteración de BL al elemento dado por la posición
Clasificar el elemento modificado, sumar una iteración
Actualizar $[valor, elemento]$ en la nueva población
Devolver la nueva población

3.4.3. Versión 3: hibridación de los mejores

Algorithm 19 Hibridación aleatoria

Copiar la población en una nueva población
Aplicar un iteración de BL al mejor elemento
Clasificar el nuevo elemento, sumar una iteración
Actualizar $[valor, elemento]$ en la nueva población
Devolver la nueva población

4. Algoritmo de comparación

Para comparar, tendremos en cuenta los resultados obtenidos por la función objetivo, cuyo valor queremos maximizar. Además, tendremos en cuenta otros datos, como el porcentaje de reducción que nos da la solución y el tiempo que tardamos en obtenerla. Lo ideal sería tener una solución que de altos valores de clasificación, con una reducción alta y un tiempo de ejecución bajo. Valoraremos que las soluciones cumplan estos tres requisitos, teniendo siempre en cuenta que lo principal es que la clasificación sea realizada correctamente.

5. Procedimiento de implementación

Esta práctica está implementada en Ruby, más concretamente en JRuby 9.0.5.0., una implementación en Java de Ruby que permite incorporar funcionalidad de Java al código. Esto nos servirá para utilizar el paquete *Weka*, que nos permite trabajar con el formato *arff*, y en particular en esta práctica leer los archivos de datos para después procesarlos internamente.

Salvo el uso del paquete *Weka*, el resto del código es una implementación propia. El lector de datos y creador de las particiones 5x2 se encuentra en el archivo *PartitionSelector.rb*, mientras que el programa principal lo hallamos en el archivo *selection.rb*, y la implementación de cada heurística en el archivo de Ruby que lleva su nombre. Adicionalmente. tenemos un archivo *Random.rb*, que permite generar soluciones iniciales pseudo aleatorias, a utilizar en las heurísticas que lo requieran.

Como guía para la implementación de las metaheurísticas he utilizado las transparencias de la clase de teoría, basándome en el pseudocódigo que se encuentra en ellas.

5.1. Requisitos de réplica / Guía

Será necesario tener instalado JRuby 9.0.5.0 para la correcta ejecución del programa. Para ello, se recomienda el uso de un gestor de versiones de ruby, ya sea rvm (https://www.rvm.io/) o rbenv (https://github.com/rbenv/rbenv), que nos permita cambiar la versión de Ruby en uso a JRuby 9.0.5.0.

Una vez estemos usando JRuby, es necesario instalar la gema de weka para que el programa pueda usarla. Para esto, simplemente hemos de ejecutar el comando *gem install weka* en la terminal, y la

tendremos instalada. Es importante recalcar que debemos estar usando JRuby 9.0.5.0 para que esta instalación se haga de forma correcta.

Hechas las instalaciones pertinentes, ya podemos ejecutar el programa. Como se ha mencinado anteriormente, el programa principal se encuentra en el archivo *selection.rb*, y se ejecuta desde terminal con el comando *ruby selection.rb*. Dicho programa principal lee los datos arff con un método llamado *readerMethod*, y seguidamente pasa a ejecutar todas las heurísticas sobre los conjuntos de datos: para 3NN, llama a un método con nombre *kNNSolver*, mientras que para el resto de metaheurísticas llama al método *cSolver*, que recibe como parámetro el nombre de la clase (heurística) que va a utilizar. Para replicar únicamente una o varias metaheurísticas basta comentar las líneas de código en el *main* que correspondan a aquellas que no queremos ejecutar. De igual forma. para cambiar el número máximo de iteraciones en una heurística, basta modificarlo en su archivo correspondiente.

6. Experimentos y análisis

6.1. Práctica 1

6.1.1. Casos empleados

Para aquellos algoritmos que necesiten una semilla aleatoria, tanto en la generación aleatoria de soluciones iniciales como en el barajado del entorno, consideramos una semilla por defecto que es la utilizada en todos los experimentos. Esta semilla puede ser modificada pasándose como parámetro a los métodos de resolución.

Para el 3NN he considerado un vector binario con todas las características activadas, y he clasificado cada conjunto usando ese vector. En este caso no dependemos de ningún parámetro adicional.

Para SFS no tenemos consideraciones iniciales, pues partimos de un vector binario puesto a 0 y añadimos la característica que más mejore.

En el resto de heurísticas, el número de iteraciones máximas ha variado dependiendo del conjunto y de la metaheurística empleada. Podemos ver los valores a continuación:

- Búsqueda local: El número máximo de iteraciones ha sido de 15000 para los tres conjuntos.
- Enfriamiento simulado: Para WDBC y Libras se ha considerado un máximo de 3000 iteraciones, mientras que para Arrhythmia, un máximo de 1000.
- Búsqueda tabú: Para WDBC y Libras tenemos un máximo de 6000 iteraciones, siendo 1200 en el conjunto Arrhythmia.

6.1.2. Resultados obtenidos

6.1.2.1. 3NN

	Wdbc			Libras			Arrhythmia		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.9614035	0	0.802	0.6777778	0	0.430	0.6134021	0	1.711
Par. 1-2	0.9718310	0	0.368	0.7777778	0	0.379	0.6197917	0	1.626
Par. 2-1	0.9614035	0	0.356	0.7222222	0	0.42	0.6030928	0	1.756
Par. 2-2	0.9612676	0	0.347	0.6611111	0	0.391	0.6041667	0	1.589
Par. 3-1	0.9508772	0	0.355	0.7444444	0	0.391	0.6340206	0	1.606
Par. 3-2	0.9612676	0	0.368	0.6277778	0	0.392	0.6302083	0	1.645
Par. 4-1	0.9473684	0	0.362	0.7055556	0	0.426	0.5876289	0	1.706
Par. 4-2	0.9859155	0	0.384	0.7	0	0.425	0.6041667	0	1.556
Par. 5-1	0.9508772	0	0.363	0.6833333	0	0.408	0.6391753	0	1.68
Par. 5-2	0.9577465	0	0.369	0.6944444	0	0.406	0.65625	0	1.565
Media	0.9600996	0	0.407	0.6994444	0	0.407	0.6191903	0	1.644

6.1.2.2. SFS

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.943662	0.838709	29.991	0.761111	0.890108	215.496	0.734375	0.956989	2621.949
Par. 1-2	0.901754	0.838709	41.463	0.683333	0.890108	152.257	0.639175	0.971326	1291.011
Par. 2-1	0.919014	0.806451	34.598	0.583333	0.934066	122.113	0.697916	0.985663	1299.541
Par. 2-2	0.961403	0.774193	49.552	0.772222	0.890109	233.983	0.747422	0.9713262	2072.618
Par. 3-1	0.936619	0.838709	39.195	0.638888	0.868131	217.941	0.692708	0.9749104	1404.079
Par. 3-2	0.912280	0.903225	23.379	0.711111	0.890109	167.262	0.659793	0.9713262	2488.381
Par. 4-1	0.961267	0.838709	47.081	0.688888	0.912088	146.734	0.744791	0.9569892	2377.977
Par. 4-2	0.936842	0.903225	31.663	0.661111	0.890109	186.596	0.556701	0.9892473	801.873
Par. 5-1	0.968309	0.870967	43.12	0.683333	0.890109	179.581	0.713541	0.9820789	1298.934
Par. 5-2	0.912280	0.838709	33.521	0.583333	0.934066	104.27	0.134020	0.9892473	1140.715
Media	0.935343	0.845161	37.356	0.676666	0.898900	172.623	0.632044	0.9749104	1679.708

6.1.2.3. Búsqueda Local

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.9542253	0.4516129	13.008	0.7611111	0.4725274	34.912	0.6197917	0.4874552	874.091
Par. 1-2	0.9157894	0.5161290	11.861	0.6444444	0.4615384	56.423	0.5927835	0.4910394	491.409
Par. 2-1	0.9260563	0.5161290	10.776	0.6666667	0.4945054	79.033	0.640625	0.4695340	749.113
Par. 2-2	0.9614035	0.4516129	27.969	0.6944444	0.4615384	79.919	0.5670103	0.4982078	337.893
Par. 3-1	0.9612676	0.3870967	18.975	0.6388889	0.4505494	60.407	0.6302083	0.5053763	592.8
Par. 3-2	0.9403508	0.4516129	15.382	0.7277778	0.4615384	58.231	0.6134020	0.4910394	813.183
Par. 4-1	0.9577464	0.4516129	13.576	0.6777778	0.4285714	58.873	0.5833333	0.5125448	409.606
Par. 4-2	0.9508771	0.4193548	15.308	0.6833333	0.4395604	31.762	0.5670103	0.5017921	552.739
Par. 5-1	0.9647887	0.2580645	22.805	0.6888889	0.4395604	44.945	0.65625	0.4982078	468.693
Par. 5-2	0.9333333	0.4838709	18.512	0.6944444	0.4395604	45.79	0.6391752	0.4874551	377.804
Media	0.9465838	0.4387096	16.817	0.6877778	0.454945	55.029	0.6109589	0.4942651	566.733

6.1.2.4. Enfriamiento Simulado

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.9577464	0.5806451	400.648	0.8	0.4725274	770.888	0.6458333	0.5017921	1612.66
Par. 1-2	0.9298245	0.5161290	555.836	0.6388889	0.5054945	995.236	0.6134020	0.5053763	1461.166
Par. 2-1	0.9436619	0.3548387	544.998	0.7166667	0.5384615	974.951	0.6770833	0.5017921	1615.839
Par. 2-2	0.9649122	0.4193548	571.41	0.7166667	0.4835164	465.301	0.5876288	0.4910394	1294.374

	Wdbc			Libras			Arrh		
Par. 3-1	0.9577464	0.4193548	539.304	0.6611111	0.4835164	1026.545	0.6354167	0.4946236	1558.966
Par. 3-2	0.9684210	0.4838709	496.143	0.7	0.4835164	1115.37	0.6185567	0.4982078	1621.783
Par. 4-1	0.9718309	0.6451612	520.008	0.6555556	0.4285714	1014.223	0.59375	0.4982079	1640.048
Par. 4-2	0.9438596	0.5161290	617.329	0.7055555	0.5164835	735.126	0.6340206	0.5268817	1537.679
Par. 5-1	0.9753521	0.4516129	664.145	0.7166667	0.5824175	728.246	0.625	0.5232974	1552.126
Par. 5-2	0.9333333	0.4838709	439.958	0.6777778	0.4065934	812.13	0.6340206	0.5017921	1301.461
Media	0.9546688	0.4870967	534.977	0.6988889	0.4901098	863.801	0.6264712	0.5043010	1519.610

6.1.2.5. Búsqueda Tabú

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.9612676	0.5806451	1637.461	0.7833333	0.6153846	1607.897	0.625	0.4982078	1147.343
Par. 1-2	0.9298245	0.5483870	1667.399	0.6833333	0.5824175	1600.208	0.5824742	0.4874551	1274.201
Par. 2-1	0.9471830	0.4516129	1615.276	0.6888889	0.5824175	1663.569	0.6041667	0.4946236	1182.883
Par. 2-2	0.9649122	0.4193548	1661.019	0.6777778	0.5164835	1758.166	0.5773195	0.4838709	1120.115
Par. 3-1	0.9612676	0.4516129	1647.04	0.6333333	0.4835164	1701.034	0.640625	0.5017921	1065.649
Par. 3-2	0.9649122	0.6129032	1536.409	0.7166667	0.5714285	1624.208	0.6391752	0.5125448	1048.134
Par. 4-1	0.9683098	0.6451612	1603.39	0.6833333	0.6263736	1717.307	0.6197917	0.4695341	1284.405
Par. 4-2	0.9473684	0.4193548	1701.557	0.6944444	0.5494505	1613.467	0.5979381	0.4910394	1063.355
Par. 5-1	0.9683098	0.6774193	1621.372	0.7	0.6043956	1616.455	0.6770833	0.5161290	1098.326
Par. 5-2	0.9403508	0.6451612	1429.072	0.6722222	0.5604395	1563.391	0.6288659	0.4910394	1124.43
Media	0.9553705	0.5451612	1611.999	0.6933333	0.5692307	1646.570	0.6192439	0.4946236	1140.884

6.1.3. Análisis de resultados

6.1.3.1. WDBC

Para el conjunto WDBC, en un orden de peor a mejor tenemos los algoritmos SFS, búsqueda local, enfriamiento simulado, búsqueda tabú y 3NN, si bien las diferencias no son muy significativas. Observando otros datos, las heurísticas de enfriamiento simulado y búsqueda tabú son demasiado costosas computacionalmente como para ser tenidas en verdadera consideración, llegando a alcanzar los 10 minutos de ejecución por partición en enfriamiento simulado, y casi 30 minutos en búsqueda tabú, sin llegar a obtener una diferencia de peso en la clasificación.

Por otra parte, el algoritmo 3NN no nos da reducción en el conjunto de características, lo que sería deseable. Parándonos con más detalle en los datos de SFS y búsqueda local, vemos que la reducción obtenida es considerable: un 43 % en búsqueda local, y casi un 85 % en SFS. Si bien los tiempos

de la solución *greedy* no son muy bajos, pues llegan a unos 40 segundo por partición, la búsqueda local, con 16 segundos de media es una solución aceptable tanto en tiempo como en resultados.

Conociendo la naturaleza del conjunto WDBC podemos deducir el por qué de estos resultados: con tan solo 30 características distintas y una distinción de las dos clases lo suficientemente pronunciada, no es necesario explorar en profundidad el conjunto de soluciones para obtener una aproximación de calidad. Los algoritmos greedy y de búsqueda local de escalada siguen las soluciones más simples, que en este caso nos llevan a una buena clasificación. Sin embargo, algoritmos más costosos computacionalmente no logran mejorar en profundidad estas soluciones ya de por sí buenas, desperdiciando tiempo de cálculo sin obtener resultados.

Concluímos pues que para este conjunto, las heurísticas 3NN, SFS y búsqueda local trabajan mejor, al obtener resultados claramente superiores. No obstante, si queremos profundizar en la tasa de aciertos debido a carácter crítico del problema (pues recordemos que se trata de la predicción de tumores), deberíamos optar por la solución que más aciertos proporcione.

6.1.3.2. Libras

Para el conjunto Libras tenemos unos resultados muy similares entre distintas metaheurísticas a la hora de comparar la tasa de aciertos: van desde el 67.67 % de SFS, pasando por búsqueda local, búsqueda tabú, enfriamiento simulado y el 69.94 % del 3NN -no llegando al 3 % de diferencia entre el mejor y el peor-. Nuevamente tenemos unos tiempos excesivos para las heurísticas de enfriamiento simulado y búsqueda tabú, sin ser el porcentaje de reducción nada a destacar, lo que hace pensar que no deberían ser considerados.

El resto de las heurísticas tienen un comportamiento similar al anterior: mientras que la solución *greedy* lleva más tiempo pero reduce más las características consideradas, el 3NN es veloz pero no proporciona reducción alguna. Entre ambos tenemos la búsqueda local, poco costosa computacionalmente y con una reducción que no destaca, pero ronda el 45 %, no muy lejana a la de SFS, siendo 3 veces más veloz.

Estos resultados nos hacen ver que el conjunto de datos puede ser bien aproximado mediante una solución *greedy*, lo que nos hace pensar que hay unas ciertas características muy relevantes a la hora de clasificar, que son rápidamente detectadas por este algoritmo. La búsqueda local actúa de una forma similar, eliminando características que empeoran y tomando las relevantes. Sin embargo, aquellas que hacen una búsqueda más exhaustiva (ES y BT) no logran obtener una mejora sustancial al no haber una gran diversificación de los máximos locales.

Las conclusiones son similares a las anteriores: si queremos enfocarnos en la obtención de buenos resultados, 3NN es nuestro algoritmo. Si queremos reducir el número de características, una solución *greedy* o la búsqueda local si queremos reducir el tiempo computacional son buenas opciones.

6.1.3.3. Arrhythmia

En este conjunto nos encontramos con unos resultados similares entre distintas heurísticas. La peor de ellas es la búsqueda local, con un 61 % de aciertos, seguida de el 3NN, la búsqueda tabú, el enfriamiento simulado, y finalmente SFS, con un 63.2 % de aciertos. Nuevamente tenemos una diferencia pequeña entre algoritmos a la hora de clasificar, llevándonos a considerar el resto de datos: tiempo y reducción, Salvo 3NN, el tiempo es similar todas las heurísticas, si bien las iteraciones de búsqueda tabú y enfriamiento simulado han sido reducidas respecto a conjuntos más pequeños.

Al haber similaridad en los tiempos, el porcentaje de reducción nos hace pensar en SFS como mejor solución, pero podemos ver un detalle importante: si bien la clasificación es superior en media, en el último conjunto de datos se obtiene el peor resultado con diferencia, siendo la tasa de aciertos de solo un 13.4 %. Puede que, en determinados casos, esta forma de operar no sea consistente, si bien habría que analizar esta anomalía con más detalle y seguir experimentando para comprobar si se repite.

Reparando en el resto de heurísticas, nuevamente 3NN nos ofrece los mejores resultados respecto al tiempo, con una pérdida de clasificación de menos del 3 % respecto al mejor algoritmo. Si queremos enfatizar en la clasificación y SFS nos parece arriesgado por la falta de datos, enfriamiento simulado ofrece buenos resultados con una reducción media del 50 %.

6.1.3.4. Conclusiones

Al contrario de lo que cabría esperar enfriamiento simulado y búsqueda tabú no ofrecen una mejorar sustancial respecto al resto de metaheurísticas. Esto puede deberse a dos cosas: o bien los máximos locales son muy próximos y muy similares en el conjunto de soluciones, o la implementación no logra hacer que se escape de dichos máximos locales, haciendo muy similar el comportamiento de estos algoritmos a una búsqueda local común. Se podría probar con otros parámetros de temperatura y lista tabú para averiguar a qué se debe dicho comportamiento.

6.2. Práctica 2

6.2.1. Casos empleados

La búsqueda local, el KNN y SFS actúan de la misma forma que lo hacían en la práctica anterior. Para las nuevas heurísticas, se han considerado el siguiente número de iteraciones:

- Búsqueda Multiarranque Básica: 25 iteraciones para WDBC y Libras, 15 para Arrhythmia.
- ILS: 25 iteraciones para WDBC y Libras, 15 para Arrhythmia.
- GRASP: 25 iteraciones para WDBC y Libras, 5 para Arrhythmia.

La reducción drástica del número de iteraciones de GRASP en Arrhythmia se debe a que la combinación del *greedy* aleatorizado con búsqueda local daba unos tiempos desorbitados, por lo que incluso con 10 iteraciones el tiempo de cómputo hubiera sido de varios días, carga que he considerado excesiva para mi ordenador y para analizar el resultado.

6.2.2. Resultados obtenidos

6.2.2.1. Búsqueda multiarranque básica

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.961267	0.548387	520.84	0.761111	0.505494	1855.819	0.661458	0.476702	13529.467
Par. 1-2	0.947368	0.419354	435.117	0.666666	0.494505	1855.654	0.634020	0.491039	12528.987
Par. 2-1	0.943661	0.516129	453.813	0.616666	0.49505	2101.774	0.65625	0.526881	13443.13
Par. 2-2	0.961403	0.451612	488.919	0.694444	0.461538	2324.778	0.649484	0.512544	11885.384
Par. 3-1	0.961267	0.354838	448.817	0.633333	0.439560	1802.019	0.65625	0.501792	14101.123
Par. 3-2	0.954385	0.516129	435.097	0.727777	0.450549	1869.898	0.623711	0.480286	12983.228
Par. 4-1	0.964788	0.451612	483.847	0.716666	0.494505	2154.312	0.65625	0.498207	14172.048
Par. 4-2	0.947368	0.419354	438.896	0.688888	0.582417	1892.106	0.639175	0.473118	13787.059
Par. 5-1	0.957746	0.419354	501.287	0.705555	0.527472	1817.816	0.640625	0.555555	10712.914
Par. 5-2	0.957894	0.419354	610.749	0.666666	0.494505	1965.027	0.628865	0.508960	13299.687
Media	0.955714	0.451612	481.738	0.687777	0.494559	1963.920	0.644608	0.502508	13044.302

6.2.2.2. GRASP

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	
Par. 1-1	0.961267	0.645161	1295.43	0.738888	0.780219	3760.475	0.661458	0.921146	8382.469
Par. 1-2	0.933333	0.806451	1490.397	0.627777	0.857142	3991.714	0.639175	0.953405	7947.838
Par. 2-1	0.929577	0.870967	1410.849	0.677777	0.780219	4296.327	0.703125	0.946236	7914.902
Par. 2-2	0.936842	0.709677	1323.872	0.655555	0.835164	4602.47	0.603092	0.942652	9354.319
Par. 3-1	0.936619	0.774193	1239.071	0.566666	0.879120	4255.26	0.645833	0.953405	10407.414
Par. 3-2	0.957894	0.774193	1131.465	0.716666	0.813186	4714.016	0.639175	0.924731	9760.299
Par. 4-1	0.943661	0.774193	1212.803	0.7	0.780219	4710.952	0.682291	0.931899	7879.459

	Wdbc			Libras			Arrh		
Par. 4-2	0.957894	0.612903	1356.695	0.638888	0.824175	4477.438	0.659793	0.939068	6992.339
Par. 5-1	0.947183	0.774193	1350.464	0.638888	0.824175	5076.974	0.619791	0.913978	8748.161
Par. 5-2	0.950877	0.806451	1252.194	0.683333	0.780219	5007.034	0.664948	0.939068	7600.59
Media	0.945514	0.754838	1306.324	0.664443	0.815383	4489.26	0.651868	0.936558	8498.779

6.2.2.3. ILS

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.961267	0.483870	448.229	0.788888	0.494505	1433.859	0.65625	0.508960	15092.196
Par. 1-2	0.936842	0.354838	380.117	0.622222	0.516483	1378.875	0.628865	0.487455	15122.0
Par. 2-1	0.940140	0.548387	315.282	0.677777	0.626373	1298.029	0.630208	0.519713	16170.674
Par. 2-2	0.961403	0.451612	411.372	0.716666	0.505494	1625.896	0.587628	0.512544	13207.481
Par. 3-1	0.954225	0.516129	412.446	0.661111	0.472527	1309.65	0.630208	0.501792	17187.111
Par. 3-2	0.961403	0.419354	410.948	0.716666	0.582417	1295.078	0.659793	0.505376	15812.339
Par. 4-1	0.982394	0.322580	401.152	0.7	0.538461	1375.185	0.645833	0.483870	17198.123
Par. 4-2	0.954385	0.451612	385.896	0.705555	0.560439	1563.133	0.597938	0.516129	15317.483
Par. 5-1	0.950704	0.387096	493.03	0.716666	0.472527	1048.437	0.635416	0.526881	17972.678
Par. 5-2	0.954385	0.677419	392.836	0.683333	0.472527	1127.548	0.659793	0.483870	13419.544
Media	0.955714	0.461289	405.130	0.698888	0.524175	1345.569	0.633193	0.504659	15649.962

6.2.2.4. Tabla comparativa

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
3-NN	0,960099	0	0,407	0,699444	0	0,407	0,619190	0	1,644
SFS	0,935343	0,845161	37,356	0,676666	0,898900	172,623	0,632044	0,974910	1679,708
BLMB	0,955714	0,451612	481,738	0,687777	0,494559	1963,920	0,644608	0,502508	13044,302
GRASP	0,945514	0,754838	1306,324	0,664443	0,815383	4489,26	0,651868	0,936558	8498,779
ILS	0,955714	0,461289	405,130	0,698888	0,524175	1345,569	0,633193	0,504659	15649,962

6.2.3. Análisis de resultados

6.2.3.1. WDBC

Para este conjunto, por orden de peor a mejor tenemos los algoritmos SFS, GRASP, ILS/BLBM y 3NN, que estan todos entre el 93.5 % y el 96 %. Estos resultados tan similares y que rozan el 100 % nos hacen ver que la clasificación es casi totalmente correcta usando cualquier solución inicial, y si nos fijamos en los datos de la práctica 1 para la búsqueda local, deducimos que es dicha búsqueda la que nos da tan buenos resultados, casi imposibles de mejorar.

Los tiempos tan excesivos de las nuevas técnicas, sin suponer ninguna mejora en el porcentaje de clasificación, hacen que ninguna de estas heurísticas deba ser tenida en consideración para este conjunto. Si queremos priorizar el porcentaje de clasificación, deberíamos elegir 3NN, mientras que si queremos reducir las características sin aumentar demasiado el tiempo, SFS es la solución que más se ajusta a estos requerimientos.

6.2.3.2. Libras

Nuevamente tenemos unos resultados muy similares entre heurísticas. GRASP es la peor, con un 66.4 % de clasificación, pasando por SFS, BLMB, ILS y 3-NN, con casi un 70 %. Al igual que antes, los tiempos de las soluciones multiarranque son muy superiores a las otras dos soluciones, y sin ser mejores, no son realmente considerables para su utilización. Ya que los tres algoritmos se parecen,

se puede concluir que la solución inicial o la mutación no es realmente importante, sino que la búsqueda local es la parte relevante de cada algoritmo.

Para la elección de un algoritmo, tenemos el mismo caso que en WDBC: si queremos priorizar velocidad, es recomendable usar 3-NN, mientras que si queremos reducir es mejor usar SFS.

6.2.3.3. Arrhythmia

Los resultados en arrhythmia son similares a los anteriores, aunque en este caso 3-NN es el peor de los algoritmos, con un 61.9 % de tasa de clasificación. Le siguen SFS, ILS, BLMB y GRASP, con un 65.1 % de clasificación. Que GRASP sea el mejor algortimo era de esperar, pues los resultados de SFS eran buenos, con una iteración con un resultado pésimo que no aparece en este caso, presumiblemente mejorada por la búsqueda local. Esto puede suponer una forma de mejorar los fallos que tenía SFS, con un porcentaje de reducción muy similar. No obstante, el tiempo de ejecución es muy superior, aun realizando únicamente 5 iteraciones.

ILS y BLMB, con un tiempo de ejecución desorbitado y sin mejorar ninguna de los porcentajes, se descartan rápidamente como opciones. 3-NN, si bien tiene el menor porcentaje de clasificación, tiene un tiempo mínimo en comparación a los demás. SFS ofrece una gran reducción con tiempos 5 veces menores que GRASP, pero tiene comportamientos extremos en algunos casos. Si queremos asegurar resultados sin tener en cuenta el tiempo, GRASP es la mejor solución.

6.2.3.4. Conclusiones

El excesivo tiempo de las heurísticas multiarranque, sin que esto suponga una mejora sustancial en casi ninguno de los casos, las hace poco viables a la hora de utilizarlas frente a SFS o 3-NN. Únicamente GRASP, por su naturaleza *greedy*, tiene algún beneficio en arrhythmia. Entre ellas, los resultados son bastante similares, siendo ILS un poco más rápida que el resto, posiblemente debido a que el algoritmo de búsqueda local no tiene que realizar muchas iteraciones al haber mutado en un 10 % de caraterísticas de una buena solución. GRASP es algo más lenta, al incluir la parte voraz, pero ofrece un buen porcentaje de reducción, y la búsqueda multiarranque básica ofrece una mejora poco sustancial sobre su versión de la práctica 1.

6.3. Práctica 3

6.3.1. Casos empleados

A continuación se detallan el número de evaluaciones de la función de clasificación en cada uno de los dos algoritmos y en cada conjunto de datos:

- Versión generacional elitista: 2500 evaluaciones para los 3 conjuntos de datos.
- Versión estacionaria: 2500 evaluaciones para WDBC y Libras, 1500 para Arrhythmia.

6.3.2. Resultados obtenidos

6.3.2.1. Versión generacional elitista

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0,971830	0,451612	850,549	0,794444	0,439560	823,928	0,635416	0,412186	3349,957
Par. 1-2	0,954385	0,193548	809,412	0,65	0,439560	766,487	0,628865	0,480286	2997,903
Par. 2-1	0,940140	0,258064	770,255	0,661111	0,472527	828,738	0,671875	0,476702	3503,069
Par. 2-2	0,961403	0,322580	846,199	0,683333	0,461538	886,384	0,623711	0,405017	3435,162
Par. 3-1	0,947183	0,290322	821,53	0,644444	0,406593	835,348	0,65625	0,473118	2968,332
Par. 3-2	0,950877	0,419354	751,933	0,694444	0,527472	851,907	0,649484	0,444444	3141,905
Par. 4-1	0,971830	0,387096	745,691	0,694444	0,494505	780,941	0,630208	0,473118	3129,607

	Wdbc			Libras			Arrh		
Par. 4-2	0,950877	0,193548	907,488	0,694444	0,373626	816,21	0,603092	0,473118	2915,801
Par. 5-1	0,943661	0,322580	802,993	0,672222	0,516483	823,291	0,651041	0,501792	3073,172
Par. 5-2	0,950877	0,354838	751,403	0,683333	0,351648	845,864	0,654639	0,462365	2616,921
Media	0,954306	0,319354	805,745	0,687221	0,448351	825,909	0,640458	0,460214	3113,182

6.3.2.2. Versión estacionaria

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0,964788	0,387096	679,376	0,777777	0,439560	753,506	0,65625	0,487455	1923,012
Par. 1-2	0,943859	0,290322	780,192	0,666666	0,472527	700,22	0,618556	0,516129	1834,515
Par. 2-1	0,929577	0,516129	605,136	0,655555	0,461538	722,667	0,614583	0,469534	2020,61
Par. 2-2	0,957894	0,322580	684,37	0,7	0,505494	763,262	0,613402	0,498207	1847,908
Par. 3-1	0,968309	0,419354	651,542	0,661111	0,461538	765,036	0,703125	0,512544	1931,804
Par. 3-2	0,954385	0,451612	723,007	0,705555	0,571428	666,317	0,628865	0,519713	1969,693
Par. 4-1	0,975352	0,387096	701,48	0,705555	0,527472	723,376	0,635416	0,501792	1843,101
Par. 4-2	0,943859	0,290322	781,602	0,694444	0,516483	679,645	0,628865	0,512544	2160,72
Par. 5-1	0,964788	0,419354	755,097	0,688888	0,516483	748,755	0,666666	0,491039	2573,508
Par. 5-2	0,957894	0,483870	739,609	0,677777	0,472527	713,782	0,639175	0,494623	1907,512
Media	0,956070	0,396773	710,141	0,693332	0,494505	723,656	0,640490	0,500358	2001,238

6.3.2.3. Tabla comparativa

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
3-NN	0,960099	0	0,407	0,699444	0	0,407	0,619190	0	1,644
SFS	0,935343	0,845161	37,356	0,676666	0,898900	172,623	0,632044	0,974910	1679,708
AGG	0,954306	0,319354	805,745	0,687221	0,448351	825,909	0,640458	0,460214	3113,182
AGE	0,956070	0,396773	710,141	0,693332	0,494505	723,656	0,640490	0,500358	2001,238

6.3.3. Análisis de resultados

6.3.3.1. WDBC

Los resultados en este conjunto son similares en cuanto a porcentaje de clasificación: 3-NN da los mejores resultados y SFS los peores, estando todos por encima del 93.5 %. Los algoritmos genéticos son similares, rondando el 95.5 % de clasificación, con una reducción entre el 30 % y el 40 %, si bien tienen unos tiempos muy superiores a SFS y 3-NN.

Como este conjunto es fácilmente clasificable usando heurísticas sencillas, considerar otras más lentas que no suponen ninguna mejora no tiene interés. De nuevo SFS y 3-NN se presentan como opciones ganadoras para optimizar reducción y tiempo, respectivamente.

Entre las versiones genéticas, ambas son bastantes similares, si bien la versión estacionaria ofrece una mayor reducción y unos tiempos alrededor del 12 % menores que la versión generacional.

6.3.3.2. Libras

Para el conjunto Libras, tenemos, de peor a mejor, los algoritmos SFS, AGG, AGE y 3-NN como el mejor, con una mínima diferencia sobre AGE y apenas un 2 % sobre SFS. Los tiempos son similares a los obtenidos en WDBC, salvo SFS, que aumenta considerablemente, por lo que nuevamente tenemos a SFS como mejor opción para reducir características relevantes y 3-NN para clasificar y para tiempos.

Comparando los algoritmos genéticos, la versión estacionaria sigue proporcionando unos tiempos menores que la generacional, además de una mayor redución (en este caso, del 5 % de diferencia), con una clasificación ligeramente mejor, apenas un 1 %.

6.3.3.3. Arrhythmia

En este caso tenemos un resultado casi idéntico en cuanto a clasificación para los dos algoritmos genéticos, con un 64 %, que mejoran a SFS y a 3-NN por un 1 % y un 2 % respectivamente. La diferencia de tiempos con respecto a SFS ya no es tan alta, siendo de algo menos de el doble (recordemos que la versión estacionaria solo ha considerado 1500 iteraciones), si bien la mejora que proporciona no es muy alta, pero puede ser importante si queremos maximizar lo posible la tasa de clasificación.

3-NN sigue siendo la mejor opción en cuanto al tiempo, con menos de 2 segundo de ejecución, reduciendo en más de 1000 veces el tiempo. SFS es una buena opción si queremos reducir características, pues los genéticos no pasan del 50 % de reducción, aunque también alcanza unos tiempos costosos.

Es posible que la falta de iteraciones haya influido negativamente en los algoritmos genéticos, no obstante, presentan unos resultados coherentes y aceptables. Sería necesaria una mayor experimentación para ver si, con un tiempo de cómputo mayor, son capaces de mejorar los valores que en este caso nos presentan.

6.3.3.4. Conclusiones

Los algoritmos genéticos ofrecen unos resultados de clasificación intermedios, con unos tiempos de ejecución altos en comparación. La reducción es baja en comparación con los algoritmos de las prácticas 1 y 2. Entre ellos, los resultados son similares, si bien la versión estacionaria es más rápida (al no tener que trabajar con generación de nuevas poblaciones de padres) y reduce más que la generacional. No establecer el torneo binario puede haber tenido relevancia en cuanto a la reducción, pero sería necesaria más experimentación para comprobar diferencias reales, pues la clasificación es similar en ambos algoritmos.

También sería necesario comprobar cual es el número de iteraciones adecuado para cada conjunto, al igual que el tamaño de la población inicial, que podría tener un tamaño proporcional al tamaño del conjunto de datos o ir modificándose dinámicamente. Todo esto requiere de más experimentación, pero podría suponer mejoras sustanciales en los algoritmos.

6.4. Práctica 5

6.4.1. Casos empleados

En todos los algoritmos, se han considerado 5000 evaluaciones para los conjuntos WDBC y Libras, y 2000 para Arrhythmia.

6.4.2. Resultados obtenidos

6.4.2.1. Hibridación total

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.968309	0.483870	1475.656	0.755555	0.417582	2296.408	0.651041	0.473118	2640.694
Par. 1-2	0.954385	0.483870	1935.509	0.605555	0.516483	1990.636	0.628865	0.491039	4365.614
Par. 2-1	0.947183	0.483870	2131.13	0.7	0.461538	2474.745	0.609375	0.422939	3915.932
Par. 2-2	0.954385	0.387096	1988.756	0.7	0.417582	2226.754	0.613402	0.505376	3546.196
Par. 3-1	0.961267	0.354838	1797.422	0.655555	0.384615	2437.296	0.630208	0.494623	3354.049
Par. 3-2	0.947368	0.451612	1857.651	0.705555	0.516483	2199.117	0.608247	0.469534	4140.638
Par. 4-1	0.975352	0.354838	1979.882	0.722222	0.527472	2359.922	0.604166	0.462365	3400.715

	Wdbc			Libras			Arrh		
Par. 4-2	0.922807	0.451612	1993.685	0.727777	0.384615	2490.808	0.582474	0.473118	3266.041
Par. 5-1	0.961267	0.451612	2116.568	0.683333	0.428571	2466.207	0.65625	0.491039	2937.102
Par. 5-2	0.947368	0.451612	1950.786	0.672222	0.483516	2200.419	0.644329	0.494623	3530.665
Media	0.953969	0.435483	1922.704	0.692777	0.453845	2314.231	0.622835	0.477777	3509.764

6.4.2.2. Hibridación aleatoria

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.961267	0.483870	1547.838	0.777777	0.406593	2626.771	0.661458	0.462365	2538.025
Par. 1-2	0.936842	0.354838	2071.873	0.638888	0.505494	2394.317	0.608247	0.487455	2932.777
Par. 2-1	0.950704	0.580645	2217.437	0.666666	0.428571	2587.669	0.604166	0.426523	3055.093
Par. 2-2	0.957894	0.387096	2371.827	0.661111	0.472527	2610.796	0.582474	0.483870	2423.355
Par. 3-1	0.950704	0.580645	2032.163	0.611111	0.483516	2534.485	0.625	0.498207	2430.86
Par. 3-2	0.954385	0.483870	1934.785	0.711111	0.439560	2616.22	0.613402	0.469534	2633.524
Par. 4-1	0.982394	0.419354	2125.309	0.7	0.494505	2564.596	0.614583	0.462365	2898.164
Par. 4-2	0.940350	0.483870	2084.384	0.694444	0.483516	2611.534	0.582474	0.487455	3117.063
Par. 5-1	0.954225	0.516129	2208.022	0.705555	0.494505	2643.889	0.661458	0.462365	2960.372
Par. 5-2	0.954385	0.451612	1947.028	0.688888	0.406593	2664.255	0.639175	0.494623	3016.71
Media	0.954315	0.474192	2054.066	0.685555	0.461538	2585.453	0.619243	0.473476	2800.594

6.4.2.3. Hibridación del mejor

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
Par. 1-1	0.968309	0.483870	1569.547	0.761111	0.395604	2185.734	0.65625	0.476702	2505.601
Par. 1-2	0.936842	0.354838	2011.608	0.672222	0.472527	2047.942	0.618556	0.487455	2836.264
Par. 2-1	0.947183	0.451612	2068.364	0.677777	0.428571	2281.455	0.614583	0.426523	2599.139
Par. 2-2	0.954385	0.387096	2059.784	0.672222	0.428571	2130.88	0.597938	0.494623	2701.832
Par. 3-1	0.950704	0.580645	1729.08	0.622222	0.450549	2124.996	0.625	0.501792	2300.177
Par. 3-2	0.947368	0.451612	1681.924	0.7	0.516483	2100.271	0.592783	0.465949	2563.75
Par. 4-1	0.982394	0.354838	1804.475	0.705555	0.505494	2064.493	0.604166	0.462365	2605.659
Par. 4-2	0.940350	0.483870	1901.852	0.677777	0.406593	2136.437	0.628865	0.483870	2791.472
Par. 5-1	0.954225	0.516129	1860.385	0.677777	0.439560	2257.577	0.609375	0.491039	2926.222
Par. 5-2	0.954385	0.516129	1879.885	0.672222	0.439560	2114.903	0.654639	0.501792	2406.55
Media	0.953614	0.458063	1856.690	0.683888	0.448351	2144.468	0.620215	0.479211	2623.666

6.4.2.4. Tabla comparativa

	Wdbc			Libras			Arrh		
	%_clas	%_red	T	%_clas	%_red	T	%_clas	%_red	T
3-NN	0.960099	0	0.407	0.699444	0	0.407	0.619190	0	1.644
SFS	0.935343	0.845161	37.356	0.676666	0.898900	172.623	0.632044	0.974910	1679.708
Hib. total	0.953969	0.435483	1922.704	0.692777	0.453845	2314.231	0.622835	0.477777	3509.764
0.1 Rand	0.954315	0.474192	2054.066	0.685555	0.461538	2585.453	0.619243	0.473476	2800.594
0.1 Mejor	0.953614	0.458063	1856.690	0.683888	0.448351	2144.468	0.620215	0.479211	2623.666

6.4.3. Análisis de resultados

6.4.3.1. WDBC

La tasa de clasificación es casi idéntica en los tres algoritmos meméticos, un 95.4 % aproximadamente. Es ligeramente superior en 3-NN, y algo menor en SFS. Sin embargo, los tiempos distan mucho de estos dos algoritmos: si bien 3-NN apenas tarda medio segundo, y SFS no llega a los 40, 5000 evaluaciones de la función de clasificación hacen que los algoritmos meméticos se vayan a los 2000 segundos de ejecución. Esto nos hace pensar que, ante la escasa mejora que suponen, no sean una opción a tener en cuenta, salvo que queramos obtener una reducción en el conjunto de características y unos resultados alrededor de un 2 % mejores que en SFS, que es el que más reduce. Ninguna de las tres versiones de hibridación ofrece diferencias significativas respecto a otra.

6.4.3.2. Libras

Para el conjunto libras nos encontramos en un caso muy similar al anterior: 3-NN es el que mejor clasifica, con un 1 % de ventaja sobre los algoritmos meméticos, y a su vez estos mejoran sobre un 1-2 % a SFS. Los tiempos de los algoritmos meméticos vuelven a ser descomunales en comparación a los otros dos, por lo que el único escenario en el que podrían ser tenidos en cuenta es si necesitamos reducir el número de características y mejorar el porcentaje que nos ofrece SFS.

6.4.3.3. Arrhythmia

En este caso, SFS bate a los 3 algoritmos meméticos, tanto en tiempo, reducción y porcentaje de clasificación. Es posible que el bajo número de iteraciones haya hecho que para este conjunto la búsqueda local haya tenido todo el peso, sobre todo en la primera versión, donde podemos ver un aumento del tiempo respecto a las otras dos, debido, seguramente, a que aunque se han superado las 2000 iteraciones, debe completar las 10 búsquedas locales sobre la población.

No tiene sentido considerar algún algoritmo memético en lugar de SFS. Entre los tres, las tercera versión, hibridación sobre el mejor, ofrece un buen resultado con el tiempo más bajo, con una buena tasa de reducción. Nuevamente, como se ha comentado en prácticas anteriores, en caso de querer optimizar el tiempo, 3-NN es la opción a tomar, pues es 1000 veces más rápido que SFS, con pérdidas que no llegan al 1.5 % en clasificación.

6.4.3.4. Conclusiones

Ninguno de los algoritmos considerados en esta práctica supone una mejora en ninguno de los tres conjuntos, siendo lso tres similares en su comportamiento. El bajo número de iteraciones considerado, sobre todo en el conjunto Arrhythmia, hacen que la búsqueda local ocupe la mayor parte del tiempo de computación, por lo que los beneficios de la parte genética del algoritmo pueden no verse reflejados.

En los conjuntos más simples, la simpleza de WDBC hacen que cualquier algoritmo sea bueno, por lo que es normal no ver mejorar sustanciales. Para Libras sí sería interesante un análisis más profundo, con un mayor número de iteraciones y distintos tipos de hibridación, aunque ya hemos visto que según las tratadas, los resultados no suponen ninguna mejora que no se pueda obtener con otro más simple, como por ejemplo enfriamiento simulado.

7. Bibliografía

- Wiki de la gema weka en JRuby: https://githubcom/paulgoetze/weka-jruby/wiki
- Docs de Ruby
  - Array: http://ruby-doc.org/core-2.3.0/Array.html
  - Hash: http://ruby-doc.org/core-2.3.0/Hash.html
  - Enumeradores: http://ruby-doc.org/core-2.3.0/Enumerable.html
  - String: http://ruby-doc.org/core-2.3.0/String.html