Metaheurísticas (2015-16)
Grado en Ingeniería Informática
Universidad de Granada

PRÁCTICA 3: BÚSQUEDAS CON TÉCNICAS BASADAS EN POBLACIONES: ALGORITMOS GENÉTICOS.

Miguel López Campos
54120359W
miguelberja@correo.ugr.es
Grupo Viernes 18:30

12 de mayo de 2016

Índice

Índice de figuras

1. Descripción del problema

El problema que estamos abordando es la selección de características. Este problema es muy útil en el campo de "machine learning".

Tenemos un conjunto de datos de entrenamiento y otro de validación, ambos etiquetados o clasificados. Lo que queremos hacer es 'aprender' una función que a partir de las características del conjunto de datos de entrenamiento, nos permita estimar el etiquetado de otros vectores de características. Lo que nosotros queremos hacer es eliminar las características que no son relevantes en el problema, eliminando de esta manera ruido en el conjunto de datos y mejorando la eficiencia de nuestro clasificador. Es decir, no sólo mejoraremos el tiempo, si no muy probablemente la calidad de nuestras soluciones también (en cuanto al error se refiere).

La gran dificultad de este problema radica en el gran número de soluciones posibles, llevándonos al punto de que un algoritmo Greedy que nos garantice la solución óptima podría llevarnos días de ejecución para determinados problemas. Es por esto por lo que tenemos que usar Metaheurísticas. Necesitamos soluciones buenas (aunque no sea la mejor) en un tiempo menor.

Nosotros usaremos para clasificar el algoritmo 3NN. Lo que hace este algoritmo es calcular la distancia euclídea entre el vector de características al cual queremos estimar una clase y el resto de vectores de características del conjunto de entrenamiento. Lo que hace el 3NN es coger los 3 elementos menos distantes y la clase mayoritaria entre esos 3 será la estimación que haremos.

Validaremos con la técnica 5x2 Cross Validation. Usaremos 5 particiones de los datos distintas al 50 % (y aleatorias) y aprenderemos el clasificador con una submuestra y validaremos con la otra y después al contrario. Con esta técnica tendremos el porcentaje de acierto, que nos servirá para ver la calidad de nuestro algoritmo.

Otros datos con los que valoraremos la calidad de nuestros algoritmos serán los tiempos de ejeución y los porcentajes de reducción, es decir, el porcentaje de características que hemos reducido.

Con nuestras metaheurísticas querremos optimizar la función de acierto. Es decir, queremos maximizar el acierto, siendo la función:
$tasaclass = 100 * \frac{n^{o}instanciasbienclasificadas}{n^{o}instanciasTotal}$

2. Descripción de los aspectos comunes de los algoritmos

La práctica ha sido desarrollada en C++.

1. Representación de las soluciones. Para representar las soluciones utilizaremos un array de booleanos. Será común a todos los algoritmos. Si la componente i es true, esto indicará que la característica i se tendrá en cuenta (no ha sido eliminada).

2. Función objetivo. La función que queremos optimizar se trata del porcentaje de acierto de estimaciones de clases, descrita en el apartado anterior.

   En pseudocódigo es la siguiente:

```
funcion_objetivo(conjunto_training, caracteristicas_activas)
begin
  Para todo elemento i del conjunto_test
  begin
    elemento <- elemento i del conjunto_training
        clase <- 3NN(conjunto_training-{elemento}, elemento,
            caracteristicas_activas)

        Si la clase estimada por 3NN se corresponde a la clase
            real -> aciertos++
  end

  promedio <- aciertos/tama o conjunto_test

  devolver promedio
end
```

3. Función clasificadora. Como función clasificadora usaremos el algoritmo 3NN, descrito anteriormente.

El pseudocódigo es el siguiente:

```
3NN(conjunto_training, vector_caracteristicas,
    caracteristicas_activas)
begin
  Para cada vector i de caracteristicas de training
  begin

    array_distancias.a adir(distanciaeuclidea(i,
       vector_caracteristicas, caracteristicas_activas))

   end

  minimo1 <- minimo(array_distancias)
  minimo2 <- minimo(array_distancias-minimo1)
  minimo3 <- minimo(array_distancias-minimo1-minimo2)

  Si la clase de vector_caracteristicas[minimo2]==clase de
     vector_caracteristicas[minimo3] entonces
    La clase del vector de caracteristicas es esa
  Si no
    La clase del vector de caracteristicas es la clase de
       vector_caracteristicas[minimo1]

  devolver clase del vector de caracteristicas

end
```

4. Función para la generación de soluciones aleatorias

```
PRE: solucion est  inicialmente entero a falso

Generar_solucion_aleatoria(solucion, tamanio_solucion)
begin
  indices_disponibles <- [0...tamanio_solucion-1]
  caracteristicas_a_cambiar <- Random(0, tamanio_solucion-1)

  Para i=0 hasta caracteristicas_a_cambiar
  begin
    caracteristica <- Random(0, indices_disponibles.length-1)
    solucion[indices_disponibles[caracteristica]] <- true
    indices_disponibles-{caracteristica}    //Elimino el indice
       de la caracteristica para no volver a cambiarla
  end

  devolver solucion
end
```

5. Antes de trabajar con cualquier algoritmo hay que normalizar los conjuntos de datos.

6. El criterio de parada para los algoritmos genéticos es que realicen 15000 evaluaciones de la función objetivo.

7. Para reducir los tiempos he seguido las recomendaciones del seminario. En el modelo generacional, para decidir si una pareja cruza o no, he calculado el número esperado de cruces (la esperanza matemática) y he realizado ese número (Probabilidad de cruce * (Número de cromosomas/2)). Igual para la mutación, donde he calculado el número de mutaciones esperado y he realizado las mutaciones a cromosomas aleatorios y genes aleatorios. Para emparejar he aprovechado la aletoriedad en la fase de selección.

8. Para cada algoritmo he plantado el mismo valor de semilla para una correspondiente iteración.

9. He usado para tomar tiempos y para crear números aleatorios las funciones dadas en decsai.

3. Algoritmo de comparación SFS

El algoritmo de comparación SFS es muy simple. Primero se genera una solución con todo a falso. A partir de aquí, exploramos todo el vector solución y cogemos la característica con la que vayamos a obtener mayor ganancia. Una vez la escojamos, volvemos a realizar otra iteración cogiendo la siguiente característica que nos de más ganancia y así sucesivamente. El algoritmo acaba cuando ya no haya mejora en una búsqueda completa sobre el vector solución.

La descripción en pseudocódigo del algoritmo es la siguiente:

```
SFS(training, Solucion, Tasa_Solucion)
begin
  mejor_solucion <- 0.0
  mejora <- true

  Mientras mejora
  begin
    mejora <- false
    S_tmp <- Solucion

    Para i=0 hasta S_tmp.length
    begin
      Si S_tmp[i] es false entonces
        S_tmp[i] <- true
        S_tmp_tasa <- funcion_objetivo(training, S_tmp)

        Si S_tmp_tasa es mejor que mejor_solucion entonces
          mejor_solucion <- S_tmp_tasa
          mejora <- true
```

```
            Solucion <- S_tmp

          S_tmp[i] <- false
      end
    end

    devolver Solucion y mejor_solucion //Por referencia

end
```

4. Algoritmo genético generacional

En los algoritmos genéticos generacionales, en cada nueva iteración, se reemplaza toda la población por la nueva. Los parámetros usados son 0.7 para la probabilidad de cruce y 0.001 para la probabilidad de mutación. Ambas probabilidades son las dadas por el guión de la práctica. Como criterio de parada he puesto que se realicen un máximo de 15000 evaluaciones de la función objetivo. Para mantener el elitismo, me aseguro de que la mejor solución de una población anterior se encuentra en la nueva población cambiándola por la peor de esta última (en el caso de que sea mejor).

El mecanismo de selección que he empleado consiste en realizar tantos concursos binarios aleatorios como cromosomas hay en la población (en nuestro caso 30). Los ganadores de estos concursos binarios serán los cromosomas de nuestra población seleccionada. La descripción en pseudocódigo es la siguiente:

```
seleccionar(poblacion, tasas_poblacion, seleccion, tasas) //Por
    referencia
begin
  seleccion <- Array() //Array de cromosomas (vectores solucion)
  tasas <- Array() //Tasas de los seleccionados

  disponibles <- 0..poblacion.length-1 //Array de  ndices  para
      controlar que no compite un cromosoma consigo mismo

  Para i=0 hasta poblacion.length
  begin
    aux <- disponibles

    random1 <- Random(0, aux.length-1) //Random devuelve un entero
    aux <- aux-{random1} //Me aseguro de que no cojo el mismo
        cromosoma
    random2 <- Random(0, aux.length-1)

    aux <- disponibles

```

```
18      Si tasas_poblacion[random1] es mejor que tasas_poblacion[random2]
            entonces
19        seleccion.aniadir(poblacion[random1])
20        tasas.aniadir(tasas_poblacion[random1])
21      si no entonces
22        seleccion.aniadir(poblacion[random2])
23        tasas.aniadir(tasas_poblacion[random2])
24
25    end
26
27    devolver seleccion y tasas //Por referencia
28
29  end
```

Para la fase de cruce lo que hago es de la selección obtenida de la fase de selección, cruzo el primero con el segundo, el segundo con el tercero, etc. hasta cubrir el número esperado de cruces, calculado con la fórmula $ProbabilidadCruce * (NumeroCromosomas/2)$. Lo que hago es elegir dos puntos de corte (asegurándome de que no son el mismo porque si no no cruzarían nada) y los elementos del cromosoma (o vector solución) que están entre esos dos puntos de corte son los que se intercambiarán un padre y otro, es decir, los nuevos cromosomas serán ellos mismos pero cada uno con la parte de central del otro. Estos puntos de corte además tienen que ir desde la segunda componente del array hasta la penúltima. La función de cruce es la siguiente:

```
1   cruce(padre1, padre2) //por referencia
2   begin
3     posibles <- 0..padre1.length-1 //Numero de caracteristicas del
          vector solucion
4
5     corte1 <- posibles[Random(1, posibles.length-2)]
6     posibles <- posibles-{corte1}
7     corte2 <- posibles[Random(1, posibles.length-2)]
8
9     Si corte1 > corte2 entonces
10      hago swap entre corte1 y corte2
11
12    aux1 <- padre1[corte1..corte2] //Cojo la parte central del array
13    aux2 <- padre2[corte1..corte2]
14
15    padre1 <- padre1-padre1[corte1..corte2] //Elimino la parte central
          del array
16    padre2 <- padre2-padre2[corte1..corte2]
17
18    padre1.insertar(aux2, corte1) //Inserto la parte central del otro
          padre a partir de cote1
19    padre2.insertar(aux1, corte1)
20
21
22    devolver padre1 y padre2 //por referencia
```

```
23
24  end
```

Para la mutación, al igual que con los cruces, calculo el número esperado de mutaciones que se van a realizar. Para ello uso la fórmula $ProbabilidadMutacion * numeroCromosomas * numeroGenes$. Siendo la probabilidad de mutación 0.001. La función con la que muto es la siguiente:

```
1  mutar(solucion, i) //Muto la componente i de solucion
2  begin
3    Si solucion[i] es true entonces lo pongo a false
4    si no lo pongo a true
5
6    devolver solucion
7
8  end
```

El algoritmo completo en pseudocódigo es el siguiente:

```
1  //PRE: Solucion inicialmente es entero falso
2
3  AGG(training, Solucion, Tasa_Solucion)
4  begin
5    poblacion <- Array() //Array de vectores solucion
6    tasas_poblacion <- Array() //Array de las tasas de la poblacion
7
8    Para i=0 hasta 30
9    begin
10     tmp <- generar_solucion_aleatoria(Solucion, Solucion.length)
11     poblacion.aniadir(tmp)
12     tasa <- funcion_objetivo(training, tmp)
13     tasas_poblacion.aniadir(tasa)
14   end
15
16
17   Para i=0 hasta 15000
18   begin
19     //Guardo el mejor elemento de la poblacion
20     indice_mejor <- maximo(tasas_poblacion)
21     mejor <- poblacion[indice_mejor]
22
23     Seleccion <- Array() //Array de vectores solucion
24     Tasas_Seleccion <- Array() //Array de tasas
25
26
27     //Fase de seleccion
28     seleccionar(poblacion, tasas_polacion, Seleccion, Tasas_Seleccion
          ) //Por referencia
```

```
29
30      //Fase de cruce
31      n_cruces <- 0.7*30/2
32
33      Para j=0 hasta n_cruces
34      begin
35        padre1 <- Seleccion[2*j]
36        padre2 <- Seleccion[2*j+1]
37
38        cruce(padre1, padre2)
39      end
40
41      //Fase de mutaci n
42      n_mutaciones <- 0.001*30*Seleccion[1].length
43
44      Para j=0 hasta n_mutaciones
45      begin
46        r1 <- Random(0, Seleccion.length-1)
47        r2 <- Random(0, Seleccion[r1].length-1)
48
49        mutar(Seleccion[r1], r2)
50      end
51
52
53      //Fase de reemplazamiento
54      Para j=0 hasta 30
55      begin
56        tasas_Seleccion[j] <- funcion_objetivo(training, seleccion[j])
57      end
58
59      encontrado <- Buscar(mejor) //Busco si  e s t  la mejor solucion
            anteriormente guardada
60
61      Si !encontrado entonces
62        min <- minimo(tasas_seleccion)
63
64        Si tasas_seleccion[min] < tasas_poblacion[ind_mejor] entonces
65          seleccion[min] <- mejor
66          tasas_seleccion[min] <- tasas_poblacion[ind_mejor]
67
68    end
69
70    ind_max <- maximo(tasas_poblacion)
71    solucion <- poblacion[ind_max]
72    Tasa_Solucion <- Tasas_poblacion[ind_max]
73
74    devolver solucion y Tasa_Solucion //Por referencia
75
76  end
```

NOTA: El incremento de la i en el bucle más externo lo hago de 30 en 30 ya que en cada iteración se realizan 30 evaluaciones de la función objetivo.

5. Algoritmo genético estacionario

En este modelo de algoritmo genético se eligen dos padres aleatorios de la población y se aplican los operadores genéticos sobre éstos. Este modelo es elitista y además tiene una convergencia rápida al reemplazar los peores cromosomas (soluciones) por otras mejores. En este modelo también usaremos como probabilidad de mutación 0.01 y la probabilidad de cruce será 1 ya que siempre se cruzarán los dos padres.

En la fase de selección realizo dos concursos binarios. En estos concursos controlaré que los que han participado en el primer concurso no participen en el segundo y además que un cromosoma no compita contra sí mismo. La selección la realizo con el siguiente pseudocódigo:

```
seleccionar(poblacion, tasas_poblacion, padre1, padre2) //Por
    referencia
begin
  posibles <- 0..poblacion.length-1 //No puedo repetir en los
      concursos el mismo elemento

  r1 <- Random(0,posibles.length-1)
  posibles <- posiles-{r1}
  r2 <- Random(0,posibles.length-1)
  posibles <- posiles-{r2}
  r3 <- Random(0,posibles.length-1)
  posibles <- posiles-{r3}
  r4 <- Random(0,posibles.length-1)

  posibles <- 0..poblacion.length-1

  Si tasas_poblacion[posibles[r1]] >= tasas_poblacion[posibles[r2]]
      entonces
    padre1 <- poblacion[posibles[r1]]
  si no entonces
    padre1 <- poblacion[posibles[r2]]

  Si tasas_poblacion[posibles[r3]] >= tasas_poblacion[posibles[r4]]
      entonces
  padre2 <- poblacion[posibles[r3]]
  si no entonces
  padre2 <- poblacion[posibles[r4]]

  devolver padre1 y padre2 //Por referencia
end
```

La fase de cruce la hago igual que en el modelo generacional. El pseudocódigo es el siguiente:

```
cruce(padre1, padre2) //por referencia
begin
        posibles <- 0..padre1.length-1 //Numero de caracteristicas
           del vector solucion

        corte1 <- posibles[Random(1, posibles.length-2)]
        posibles <- posibles-{corte1}
        corte2 <- posibles[Random(1, posibles.length-2)]

        Si corte1 > corte2 entonces
        hago swap entre corte1 y corte2

        aux1 <- padre1[corte1..corte2] //Cojo la parte central del
           array
        aux2 <- padre2[corte1..corte2]

        padre1 <- padre1-padre1[corte1..corte2] //Elimino la parte
           central del array
        padre2 <- padre2-padre2[corte1..corte2]

        padre1.insertar(aux2, corte1) //Inserto la parte central del
           otro padre a partir de cote1
        padre2.insertar(aux1, corte1)


        devolver padre1 y padre2 //por referencia

end
```

La mutación también uso el mismo modelo que en el generacional:

```
mutar(solucion, i) //Muto la componente i de solucion
begin
        Si solucion[i] es true entonces lo pongo a false
        si no lo pongo a true

        devolver solucion
end
```

La descripción en pseudcódigo del algoritmo completo (AGE) es la siguiente:

```
//PRE: Solucion inicialmente es entero falso

AGG(training, Solucion, Tasa_Solucion)
begin
  poblacion <- Array() //Array de vectores solucion
  tasas_poblacion <- Array() //Array de las tasas de la poblacion

```

```
  Para i=0 hasta 30
  begin
    tmp <- generar_solucion_aleatoria(Solucion, Solucion.length)
    poblacion.aniadir(tmp)
    tasa <- funcion_objetivo(training, tmp)
    tasas_poblacion.aniadir(tasa)
  end


  Para i=0 hasta 15000
  begin
    padre1 <- Array solucion
    padre2 <- Array solucion

    //FASE DE SELECCI N
    seleccionar(poblacion, tasas_poblacion, padre1, padre2)

    //FASE DE CRUCE
    cruce(padre1, padre2)

    //FASE DE MUTACI N
    random1 <- Rand() //Numero aleatorio entre 0 y 1 sin incluir el 1
    random2 <- Rand()

    Si random1 <= 0.001 entonces
      r1 <- Random(0, padre1.length-1)
      mutar(padre1, r1)

    Si random2 <= 0.001 entonces
      r1 <- Random(0, padre2.length-1)
      mutar(padre2, r2)

    //FASE DE REEMPLAZAMIENTO
    tasa_padre1 <- funcion_objetivo(training, padre1)
    tasa_padre2 <- funcion_objetivo(training, padre2)

    min1 <- minimo(tasas_poblacion)
    min2 <- minimo(tasas_poblacion-tasas_poblacion[min2])

    Si tasa_padre2 >= tasa_padre1 entonces
      Hago swap entre padre1 y padre2

    Si tasa_padre1 >= tasas_poblacion[min1] entonces
      tasas_poblacion[min1] <- tasa_padre1
      poblacion[min1] <- padre1

      Si tasa_padre2 >= tasas_poblacion[min2] entonces
        tasas_poblacion[min2] <- tasa_padre2
        poblacion[min2] <- padre2
```

```
57
58        si no si tasa_padre1 >= tasas_poblacion[min2] entonces
59          tasas_poblacion[min2] <- tasa_padre1
60          poblacion[min2] <- padre1
61
62      end
63
64      ind_max <- maximo(tasas_poblacion)
65      Solucion <- poblacion[ind_max]
66      Tasa_Solucion <- tasas_poblacion[ind_max]
67
68      devolver Solucion y Tasa_Solucion //por referencia
69
70  end
```

NOTA: En el bucle más externo voy incrementando la i de 2 en 2 ya que en cada iteración se realizan 2 evaluaciones de la función objetivo.

6. Aspectos técnicos de la práctica

La práctica ha sido desarrollada en C++. El código ha sido implementado basándome en los pseudocódigos de las transparencias de clase (adaptándolos al problema). Cada uno de los algoritmos está implementado en un cpp diferente. Dentro de estos cpp tenemos las funciones de evaluación, así como de lectura de los ficheros de datos. Cada algoritmo por lo tanto se evaluará en un ejecutable distinto. Para compilar el código simplemente hay que usar make (hay un makefile implementado) y en la carpeta bin se crearán los ejecutables. Cada ejecutable tendrá como salida los datos de las ejecuciones de los algoritmos.

Los ficheros de datos que he usado son los que hay subidos en la plataforma de la asignatura, a excepción de movement_libras, cuyo fichero de datos he tenido que descargarlo de la web dada en las transparencias ya que para leer el que había en la plataforma tuve problemas (pero el contenido de los datos es el mismo).

Para la toma de tiempos he usado las funciones dadas en decsai. También para generar números aleatorios he usado las funciones dadas por los profesores.

7. Experimentos y análisis

A continuación las tablas con los resultados de los experimentos realizados:

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	96,49	X	0,01	66,11	X	0,02	62,69	X	0,04
Partición 1-2	96,48	X	0,02	77,78	X	0,01	62,69	X	0,03
Partición 2-1	97,19	X	0,01	70,56	X	0,01	65,29	X	0,04
Partición 2-2	96,48	X	0,02	76,11	X	0,02	62,18	X	0,03
Partición 3-1	95,09	X	0,01	72,78	X	0,01	65,80	X	0,03
Partición 3-2	96,48	X	0,01	79,44	X	0,01	65,29	X	0,04
Partición 4-1	96,14	X	0,02	68,89	X	0,01	64,25	X	0,04
Partición 4-2	96,13	X	0,01	71,67	X	0,01	63,21	X	0,03
Partición 5-1	94,74	X	0,02	76,67	X	0,01	62,18	X	0,04
Partición 5-2	97,89	X	0,01	75,00	X	0,01	64,77	X	0,03
Media	96,31	X	0,01	73,50	X	0,01	63,84	X	0,04

Figura 7.1: Experimentos para 3NN

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	94,04	83,33	2,28	61,11	92,22	9,59	68,39	97,84	77,75
Partición 1-2	89,08	90,00	1,57	68,33	93,33	8,44	64,77	98,92	44,45
Partición 2-1	95,79	83,33	2,28	67,78	82,22	19,25	69,95	98,20	64,86
Partición 2-2	95,77	90,00	1,60	71,67	87,78	14,08	61,14	97,84	75,09
Partición 3-1	95,09	86,67	1,93	72,22	86,67	15,35	75,65	97,12	96,14
Partición 3-2	94,37	86,67	1,94	70,56	87,78	14,12	66,32	98,92	43,52
Partición 4-1	89,12	93,33	1,21	61,67	92,22	9,56	69,43	98,20	65,06
Partición 4-2	94,37	86,67	1,96	72,78	91,11	10,62	67,88	96,76	106,66
Partición 5-1	95,09	90,00	1,57	68,89	90,00	11,78	67,88	96,76	107,14
Partición 5-2	96,48	90,00	1,58	75,00	91,11	10,69	69,95	98,56	53,85
Media	93,92	88,00	1,79	69,00	89,44	12,35	68,14	97,91	73,45

Figura 7.2: Experimentos para SFS (algoritmo de comparación)

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	95,09	76,67	217,26	70,00	36,67	205,24	66,32	50,72	632,36
Partición 1-2	96,48	40,00	220,97	78,33	17,78	205,21	65,80	67,63	611,56
Partición 2-1	95,79	33,33	226,08	68,33	50,00	211,79	70,98	65,47	597,37
Partición 2-2	96,13	26,67	220,95	72,78	35,56	213,90	63,21	65,83	615,88
Partición 3-1	95,09	30,00	219,13	73,89	43,33	211,29	66,32	64,75	607,26
Partición 3-2	96,13	40,00	221,66	80,00	48,89	205,49	66,32	45,68	601,50
Partición 4-1	96,50	23,33	219,28	68,89	42,22	207,21	63,73	59,35	602,44
Partición 4-2	96,48	16,67	224,34	75,00	47,78	210,08	68,91	78,77	600,78
Partición 5-1	94,04	26,67	226,40	78,89	52,22	206,70	63,21	56,47	597,81
Partición 5-2	97,18	40,00	231,28	74,44	21,11	210,89	65,29	79,86	592,70
Media	95,89	35,33	222,74	74,06	39,56	208,78	66,01	63,45	605,97

Figura 7.3: Experimentos para AGG

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	95,79	60,00	205,68	69,44	15,56	207,64	64,25	70,86	606,67
Partición 1-2	95,42	26,67	208,04	77,78	48,89	205,27	63,21	62,95	603,86
Partición 2-1	95,44	30,00	206,67	68,33	68,89	205,51	68,91	74,46	597,48
Partición 2-2	96,48	23,33	207,56	76,67	36,67	206,22	65,80	33,09	589,78
Partición 3-1	95,44	20,00	209,67	72,22	47,78	212,21	67,88	89,20	565,31
Partición 3-2	96,13	10,00	212,45	77,78	63,33	217,97	62,18	27,34	563,43
Partición 4-1	96,49	50,00	217,57	69,44	37,78	210,48	66,32	84,89	564,89
Partición 4-2	96,13	30,00	216,29	72,22	4,44	207,89	67,36	47,48	563,46
Partición 5-1	94,04	33,33	207,90	76,11	45,56	210,36	66,84	66,19	565,08
Partición 5-2	97,54	26,67	207,67	73,33	38,89	207,37	68,91	84,89	564,51
Media	95,89	31,00	209,95	73,33	40,78	209,09	66,17	64,14	578,45

Figura 7.4: Experimentos para AGE

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
3-NN	96,31	X	0,01	73,5	X	0,01	63,84	X	0,04
SFS	93,92	88,00	1,79	69,00	89,44	12,35	68,14	97,91	73,45
AGG	95,89	53,33	222,74	74,06	39,56	208,78	66,01	63,45	605,97
AGE	95,89	31	209,95	73,33	40,78	209,09	66,17	64,14	578,45

Figura 7.5: Comparación de los algoritmos

Como vemos, los dos algoritmos genéticos mejoran a SFS en las dos primeras bases de datos. SFS no presenta apenas diversidad, por lo que probablemente caiga en un óptimo local pronto. Además, 3-NN nos muestra que para el vector de características completo, los resultados suelen ser mejores que con reducción sobre este, lo que podría significar que estas bases de datos tienen poco ruido en sus datos. Esto implica que una reducción más alta, como la que realiza SFS (pues la solución inicial que empieza a explotar es entera falsa), lleve hacia resultados peores que, en este caso, los algoritmos genéticos, que como vemos presentan menor tasa de reducción y mayor tasa de clase. Estos algoritmos, obviamente, presentan una mayor diversidad de exploración.

Lo mismo pasa para movement libras. Mientras SFS presenta una tasa de reducción del 89 %, AGG y AGE presentan una del 40 % aproximadamente ambas. Como podemos observar, estos dos algoritmos también nos da para esta base de datos un resultado de hasta un 5 % mejor que SFS (en tasa de clasificación).

Para Arrhythmia cambia bastante la cosa. 3NN obtiene un resultado del 63 % de tasa de clasificación, mientras que tras aplicar los algoritmos de búsqueda siempre solemos mejorar la tasa. Esto es un indicativo de que la base de datos contiene mucho ruido entre sus datos, por lo que la reducción de estos datos ruidosos mejora la tasa de clasificación. En este caso SFS sí se muestra mejor que AGG y AGE y también con una tasa de reducción mucho mejor (más de un 30 % superior), lo que nos puede llevar a la conclusión de que en este caso, empezar con una solución totalmente vacía nos llevará a soluciones mejores que comenzando con soluciones aleatorias (como hacemos en los algoritmos genéticos).

Como conclusión hay que decir que AGG y AGE probablemente obtendrían mejor resultado para Arrhythmia que SFS en el caso de que se empezara con una población algo más vacía, es decir, con vectores solución más vacíos, ya que como hemos visto en prácticas anteriores para Arrhythmia hay que reducir bastante el vector solución por ser tan ruidoso.

En cuanto a los tiempos, vemos como los algoritmos genéticos son mucho más lentos que SFS, algo normal cuando los genéticos van a realizar seguras 15000 evaluaciones de la función objetivo mientras que SFS probablemente acabe en un óptimo local mucho antes de llegar a ese número de evaluaciones. AGE es ligeramente más rápido que AGG y puede deberse a que en AGG se realizan más mutaciones y cruces y por esto esta diferencia temporal.

A continuación unos gráficos ilustrativos de la comparación de los algoritmos para cada base de datos.

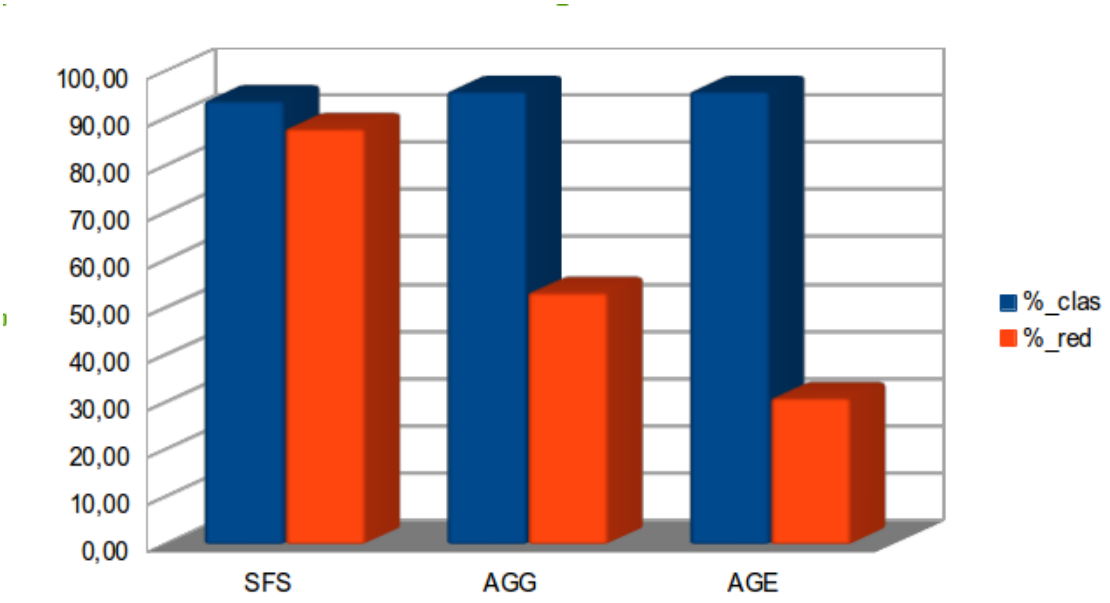

Figura 7.6: Gráfica comparativa para WDBC

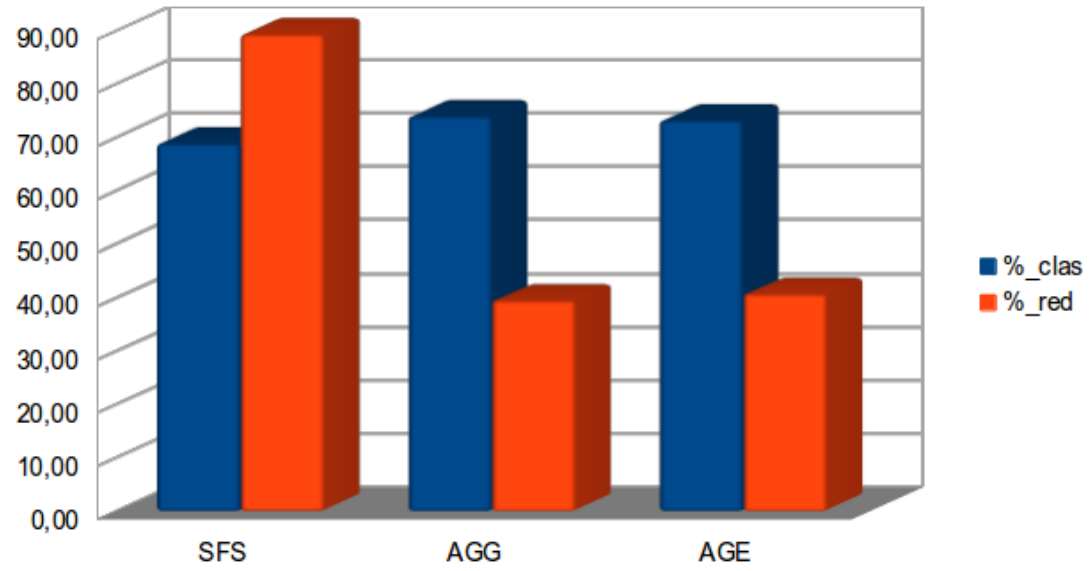

Figura 7.7: Gráfica comparativa para movement libras

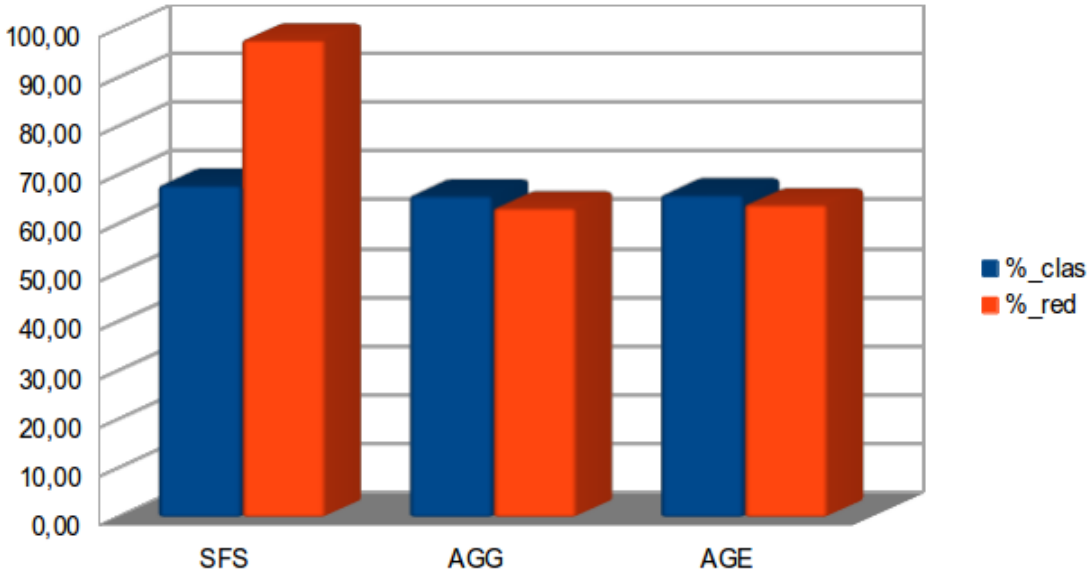

Figura 7.8: Gráfica comparativa para Arrhythmia