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

PRÁCTICA 2: BÚSQUEDA POR TRAYECTORIAS MÚLTIPLES PARA EL PROBLEMA DE LA SELECCIÓN DE CARACTERÍSTICAS

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

11 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. Todos los algoritmos tendrán como criterio de parada que se hayan hecho 25 búsquedas locales en total. La búsqueda local por su parte, como en la práctica anterior, tendrá como criterio de parada que ninguna de las soluciones vecinas a la actual mejore a esta y que no se realicen más de 15000 evaluaciones de la función objetivo.

7. El operador de generación de vecinos en búsqueda local se trata de la inversión de una componente aleatoria de una solución. Es decir, $flip(s, i)$ cambia la componente i del vector solución s a true si era false y a false si era true. i es aleatorio.

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
```

```
21
22          S_tmp[i] <- false
23        end
24      end
25
26      devolver Solucion y mejor_solucion //Por referencia
27
28  end
```

4. Algoritmo de búsqueda local

La descripción en pseudocódigo del algoritmo es la siguiente:

```
1   busqueda_local(training, Solucion, Tasa_Solucion)
2   begin
3     Tasa_Solucion <- funcion_objetivo(training, Solucion)
4
5     Mientras que se encuentre una soluci n mejor y no se superen 15000
          soluciones exploradas
6     begin
7       S <- Solucion
8
9       repetidos <- [0...S.length-1]
10      Mientras que no se mejore y mientras que no se haya generado todo
            el entorno de S
11      begin
12        S <- flip(S, repetidos) //Con repetidos evitamos repetir dos
                                  //soluciones de un mismo entorno
13
14        coste_S <- funcion_objetivo(training, test, S)
15
16        Si coste_S es mejor que Toste_Solucion entonces
17          //S mejora a Solucion
18          Solucion <- S
19          Tasa_Solucion <- coste_S
20
21       end
22
23      end
24
25      devolver Solucion y Toste_Solucion //Por referencia
26
27   end
```

La exploración del entorno la realizo cambiando aleatoriamente una componente del vector solución (con la función flip). Para asegurarme de que no realizo flip de una misma posición en el entorno de la solucion actual más de una vez, tengo un vector de enteros (repetidos) que repre-

sentan los índices del array solución y que es pasado a flip por referencia. Lo que hace flip es generar un número aleatorio r entre 0 y el tamaño de este vector 'repetidos' y cambia la posición del vector solucion que marca repetidos[r]. Posteriormente, flip elimina la posición r del vector repetidos, con lo que ya nos aseguramos que no se vuelva a cambiar esta posición del vector solución.

Cuando una solución del entorno es mejor que la solución cuyo entorno es el que estamos explorando, pasaremos a explorar ahora el entorno de esta nueva solución. Describiéndolo en pseudocódigo obtendríamos lo siguiente:

La función flip:

```
Flip(Solucion, Repetidos)
begin
  random <- Random(0, Repetidos.length-1)

  index <- Repetidos[random]

  Si Solucion[index] es true lo cambio a false
  si no lo cambio a true

  Repetidos-{random}


  devolver Solucion
end
```

Random devuelve un número aleatorio entero.

Para la exploración del entorno en general realizo lo siguiente:

```
Mientras que no genere todos los vecinos y no se mejore la solucion
  S <- Mejor Solucion
  S <- flip(S, repetidos)

  Si la componente cambiada por flip no es la que devolveria a S a
      ser la solucion anteriormente explorada entonces
    Coste_S <- funcion_objetivo(training, test, S)

    Si Coste_S es mejor que el coste de la mejor solucion
    entonces
      actualizo la mejor solucion
      paso a explorar entorno de la nueva mejor solucion


```

```
  Si el contador de soluciones es igual al tamanio de S-1
  entonces
  He explorado todo el entorno de S
end
```

5. Búsqueda multiarranque básica

La descripción del algoritmo en pseudocódigo es la siguiente:

```
BMB(training, Solucion, Tasa_Solucion)
begin
  Solucion <- generar_solucion_aleatoria(Solucion, solucion.length)
  Tasa_Solucion <- funcion_objetivo(training, Solucion)
  S' <- Solucion
  tasa_S' <- Tasa_Solucion

  Para i=1 hasta 25
  begin
    busqueda_local(training, S', tasa_S') //Argumentos por referencia

    Si tasa_S' >= Tasa_Solucion
          Solucion <- S'
          Tasa_Solucion <- tasa_S'

        S'[0..S'.length-1] <- false
        S' <- generar_solucion_aleatoria(S', S'.length)
  end

  devolver Solucion y Tasa_Solucion //Por referencia

end
```

El algoritmo como vemos es bastante sencillo. Lo que hago es crear una solución aleatoria inicial y entramos en un bucle que se repetirá 25 veces (criterio de parada dado en el guión). Ejecutamos búsqueda local sobre esta solución aleatoria y si es mejor que la mejor que teníamos hasta ahora, actualizo esta. Vuelvo a generar una solución aleatoria y se repite el procedimiento.

6. Algoritmo GRASP

Este algoritmo sigue la misma dinámica que MBM, a diferencia de que en este caso en cada iteración del algoritmo en lugar de generar una solución aleatoria, genera una solución dada por un algoritmo greedy probabilístico, que describo en pseudocódigo a continuación:

```
random_greedy(training, Solucion, Tasa_S')
begin
  S' <- Solucion
  tasa_S' <- funcion_objetivo(training, S')
  Tasa_Solucion <- tasa_S'

  LC <- [1..S'.length-1] //Array de enteros (indices de las
                         //caracteristicas)
  fin <- false

  Mientras !fin y LC no sea vacio
  begin
    ganancias <- Array()
    tasas <- Array()

    Para i=0 hasta LC.length
    begin
      S'[LC[i]] <- true
      tasa <- funcion_objetivo(training, S')
      ganancia <- tasa-tasa_S'
      tasas[i] <- tasas
      ganancias[i] <- ganancia
      S'[LC[i]] <- false
    end

    maxima_ganancia <- maximo(ganancias)
    minima_ganancia <- minimo(ganancias)

    LRC <- Array() //LRC es otro array de enteros que marca los
       indices de LC

    Para i=0 hasta LC.length
    begin
      Si ganancias[i] >= maxima_ganancia - 0.3*(maxima_ganancia-
         minima_ganancia)
      entonces
        LRC.aniadir(i)
    end

    Si LRC esta vacia entonces ponemos fin<-true
    Si no
      random <- Random(0, LRC.length-1)
      tmp <- S'
      tmp[LC[LRC[random]]] <- true // Pongo la caracteristica
         correspondiente a true
      tmp_tasa <- tasas[LRC[random]]

      Si tmp_tasa >= tasa_S'
```

```
45          S' <- tmp
46          tasa_S' <- tmp_tasa
47        si no fin <- true
48
49        LC <- LC-{elemento LRC[random]}
50      end
51
52      Solucion <- S'
53      Tasa_Solucion <- tasa_S'
54
55      devolver Solucion y Tasa_Solucion //Por referencia
56  end
```

En resumen, mi algoritmo greedy probabilístico consiste en tener una lista de candidatos que representan cada una de las características del vector de características 'solucion'. En mi representación, esta lista de candidatos (LC) se trata de un array de números enteros que cubre todos los índices del vector 'solucion', es decir, va desde 0 hasta solucion.length-1. Con esta lista de candidatos calculo la ganancia de cada una de las características de LC, poniéndolas a true una por una en el vector solucion. La ganancia la he interpretado como la diferencia entre la tasa de clase conseguida activando una determinada característica y la tasa de clase del array solucion cuando no estaba activada la característica.

Mi lista restringida de candidatos (LRC) se trata también de un array de enteros que contendrá todos los índices de 'LC'. Es decir, es lo mismo que LC pero en lugar de apuntar al vector 'solucion' apunta al array 'LC'. Para incluir un elemento en esta lista lo que hago es comprobar si la ganancia al activar la característica 'i' es mayor o igual que $Maximaganancia - \alpha(Maximaganancia - Minimaganancia)$. Nuestro α será 0.3 (dado en el guión de la práctica).

Finalmente lo que hacemos es comprobar si LRC es vacía. En el caso de que lo sea, terminamos la ejecución del algoritmo. En el caso de que no, elegimos una de las características incluidas en la LRC de forma aleatoria y en el caso de que la tasa supere la anterior, actualizamos la solución y eliminamos la característica activada de la lista de candidatos 'LC'. Se vuelve a repetir el procedimiento hasta que nos quedemos sin candidatos en LC o hasta que no se mejore la solución.

El algoritmo GRASP en pseudocódigo es el siguiente:

```
1  PRE: Inicialmente Solucion es entero falso
2
3  GRASP(training, Solucion, Tasa_Solucion)
4  begin
5    S' <- Solucion
6    Tasa_S' <- Tasa_Solucion
7
8    Para i desde 1 hasta 25
9    begin
```

```
    random_greedy(training, S', Tasa_S') //Por referencia
    busqueda_local(training, S', Tasa_S') //Por referencia

    Si Tasa_S' >= Tasa_Solucion
      Solucion <- S'
      Tasa_Solucion <- Tasa_S'

    S'[0..S'.length-1] <- false
  end

  devolver Solucion y Tasa_Solucion //por referencia
end
```

Al igual que en los algoritmos anteriores, el criterio de parada es que se repita la búsqueda local 25 veces.

7. Búsqueda local iterativa (ILS)

En este algoritmo de búsqueda se sigue una dinámica parecida a los dos anteriores. La diferencia es que en cada iteración, en lugar de comenzar comenzar con una totalmente nueva solución inicial, se realiza una mutación de la actual. Es decir, lo que haremos es crear una solución inicial de forma aleatoria. A esta solución inicial le aplicaremos la búsqueda local. Una vez actualizada la mejor solución (en el caso de que mejore), la mejor solución encontrada hasta el momento mutará y se le aplicará la búsqueda local a esta mutación. Este procedimiento se repetirá 25 veces (criterio de parada).

Mi operador de mutación consiste en cambiar 0.1*n características, siendo n el número total de características del vector solución. Si la característica aleatoria i era false pasará a ser true y viceversa. La descripción en pseudocódigo es la siguiente:

```
mutar(Solucion)
begin
  disponibles <- [0..Solucion.length-1]

  Para i=0 hasta 0.1*Solucion.length
  begin
    random <- Random(0, disponibles.length-1)

    Si Solucion[disponibles[random]] es true lo pongo a false y si no
        lo pongo a true

    disponibles <- disponibles-{elemento random}
  end

  devolver Solucion //Por referencia
end
```

Como vemos, en la función mutar vuelvo a usar un array (disponibles) de enteros que "señalarán.ª cada una de las características de Solucion.

El algoritmo ILS en sí es muy sencillo. Como he comentado anteriormente, sigue una dinámica parecida a la de los otros algoritmos. Como criterio de parada también tiene que se repita la búsqueda local un máximo de 25 veces. La descripción en pseudocódigo es la siguiente:

```
PRE: Inicialmente Solucion es entero falso

ILS(training, Solucion, Tasa_Solucion)
begin
  Solucion <- generar_solucion_aleatoria(Solucion, Solucion.length)
  Tasa_Solucion <- funcion_objetivo(training, Solucion)
  S' <- Solucion
  Tasa_S' <- Tasa_Solucion

  Para i = 1 hasta 25
  begin
    busqueda_local(training, S', Tasa_S') //Por referencia

    Si Tasa_s' es mejor que Tasa_Solucion
      Solucion <- S'
      Tasa_Solucion <- Tasa_S'

    //Hago la mutaci n sobre la mejor soluci n de las 2
    S' <- Solucion
    Tasa_S' <- Tasa_Solucion

    S' <- mutar(S')
  end

  devolver Solucion y Tasa_Solucion //Por referencia
end
```

8. 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 al goritmos 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 asignatu-

ra, 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.

9. 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 9.1: 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 9.2: Experimentos para SFS (algoritmo de comparación)

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	94,74	63,33	20,30	68,33	70,00	61,21	69,95	92,09	801,84
Partición 1-2	95,77	60,00	16,39	75,56	75,56	59,82	67,58	84,89	710,00
Partición 2-1	95,79	53,33	16,94	73,33	81,11	56,63	73,06	95,32	680,70
Partición 2-2	97,89	33,33	20,64	75,00	52,22	58,52	68,39	94,96	733,23
Partición 3-1	94,04	56,67	18,53	74,44	74,44	60,00	70,98	94,96	745,32
Partición 3-2	95,77	30,00	24,98	75,56	75,56	69,70	68,39	93,88	787,06
Partición 4-1	97,19	53,33	18,40	66,67	84,44	60,96	62,69	91,01	807,03
Partición 4-2	95,42	63,33	18,74	73,33	83,33	57,35	68,39	86,33	706,99
Partición 5-1	93,68	53,33	20,85	72,22	84,44	60,94	64,25	94,60	653,92
Partición 5-2	97,89	23,33	18,72	75,00	81,11	59,68	67,88	75,54	753,14
Media	95,82	49,00	19,45	72,94	76,22	60,48	68,16	90,36	737,92

Figura 9.3: Experimentos para BMB

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	95,09	70,00	78,98	62,78	91,11	355,34	73,06	92,81	3016,33
Partición 1-2	95,42	66,67	85,18	72,22	82,22	384,82	70,47	91,73	2768,90
Partición 2-1	96,49	70,00	85,10	68,33	84,44	365,43	73,58	95,68	3168,46
Partición 2-2	94,01	56,67	87,65	76,11	83,33	382,28	66,32	92,81	3261,09
Partición 3-1	96,49	36,67	85,81	74,44	88,89	388,51	75,13	91,73	3447,52
Partición 3-2	96,83	80,00	81,15	75,00	75,56	369,67	68,39	94,24	3414,33
Partición 4-1	96,84	73,33	77,24	70,56	84,44	392,35	68,39	94,24	2758,12
Partición 4-2	96,13	66,67	81,31	75,56	83,33	397,57	74,61	89,21	3510,77
Partición 5-1	92,28	83,33	68,50	76,11	83,33	398,81	74,61	92,81	2578,16
Partición 5-2	95,77	60,00	85,13	75,00	83,33	371,08	70,47	92,09	4471,61
Media	95,54	66,33	81,61	72,61	84,00	380,59	71,50	92,73	3239,53

Figura 9.4: Experimentos para GRASP

	Wdbc			Movement_Libras			Arrhythmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	97,19	70,00	10,60	70,56	61,11	33,30	68,39	95,32	299,39
Partición 1-2	95,07	23,33	10,68	76,11	88,89	31,43	61,66	39,93	296,58
Partición 2-1	96,49	40,00	10,79	70,56	27,78	31,79	73,06	95,32	307,07
Partición 2-2	95,77	30,00	11,06	74,44	37,78	32,27	63,21	38,85	301,65
Partición 3-1	95,09	26,67	10,89	68,33	90,00	31,19	70,99	94,96	312,41
Partición 3-2	97,89	40,00	10,72	83,89	74,44	31,94	70,47	91,73	286,06
Partición 4-1	96,49	73,33	11,08	68,33	53,33	31,77	69,95	97,84	287,26
Partición 4-2	94,72	66,67	10,99	72,78	53,33	32,13	60,62	18,71	279,29
Partición 5-1	94,39	66,67	10,47	75,56	28,89	32,72	64,25	94,60	304,83
Partición 5-2	97,89	16,67	10,74	74,44	24,44	32,30	68,39	92,45	299,13
Media	96,10	45,33	10,80	73,50	54,00	32,08	67,10	75,97	297,37

Figura 9.5: Experimentos para ILS

	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
BMB	95,82	49,00	19,45	72,94	76,22	60,48	68,16	90,36	737,92
GRASP	95,54	66,33	81,61	72,61	84	380,59	71,5	92,73	3239,53
ILS	96,1	45,33	10,8	73,5	54	32,08	67,1	75,97	297,37

Figura 9.6: Comparación de todos los algoritmos

Como podemos observar en esta última tabla, los resultados son muy parecidos para todos los algoritmos. 3-NN sin ninguna reducción en los vectores de características muestra mejor tasa de clase para WDBC y para movement libras por lo general. Esto se podría deber a que no hay mucho ruido entre las características, es decir, reduciendo las características siempre tenemos peores resultados porque quitamos datos importantes (y no ruidosos), o que no encontramos la solución óptima. En cambio, para la base de datos de Arrhythmia si vemos como obtenemos mejor tasa de clasificación tras aplicar los algoritmos de búsqueda que en el 3NN. Los datos de arrhythmia tienen un gran número de características, por lo que probablemente tengan más ruido que las otras 2 bases de datos.

En Arrhythmia también podemos ver que GRASP obtiene mucho mejores resultados que los demás algoritmos de búsqueda. Esto puede deberse a que en GRASP partimos de una solución muy buena (dada por el algoritmo greedy probabilístico) y lo que GRASP hace es aplicar búsqueda local sobre esta solución, es decir, queremos converger a una buena solución a partir de la ya dada por el greedy. En las demás bases de datos GRASP es levemente peor que los otros dos algoritmos. El hecho de que en Arrhythmia sea mejor y en las otras dos no, puede deberse al gran espacio de búsqueda que tiene esta base de datos. En las otras bases de datos, con espacios de búsqueda algo más pequeños, si puede darse que aplicando búsqueda local sobre una solución aleatoria termine dando su fruto.

En cuanto a ILS vemos como para WDBC da mejor resultado que BMB y GRASP. Esto se puede deber a que una vez encontramos una solución "medio buenaïntensificamos sobre ésta aplicando mutaciones. Como vemos esto es mejor para espacios de búsqueda reducidos, ya que en estos casos puede resultar mejor aplicar una técnica de intensificación que de diversificación.

En cuanto a los tiempos vemos que el algoritmo más lento es GRASP. Esto es normal ya que para cada iteración tenemos que aplicar un algoritmo greedy que es costoso en tiempo. Este Greedy, además, es más costoso en tiempo que SFS, pues estamos introduciendo características de forma aleatoria y no siempre la mejor. Como vemos para la base de datos de Arrhythmia, llegamos a tiempos de hasta 4000 segundos, que es algo más de una hora por ejecución. BMB e ILS tienen tiempos algo más parecidos. Aún así, BMB es más lento y podría deberse a que BMB en cada iteración genera una nueva solución aleatoria, por lo que ya tiene que aplicar búsqueda local sobre una solución totalmente random y puede tardar en dejar de mejorar mucho tiempo. En cambio en ILS partimos de una solución aleatoria a la que le aplicamos la búsqueda local. Como recordamos, búsqueda local tenía como criterio de parada que dejara de mejorar, por lo

tanto tras una ejecución de búsqueda local esa solución es más difícil que sea mejorable. Como en el resto de iteraciones la búsqueda local se aplica sobre pequeñas mutaciones de esa solución, es decir, a soluciones parecidas, entonces puede que por esta razón realice menos iteraciones (deje de mejorar antes).

A continuación dejo algunas gráficas ilustrativas de los resultados de los algoritmos para cada base de datos.

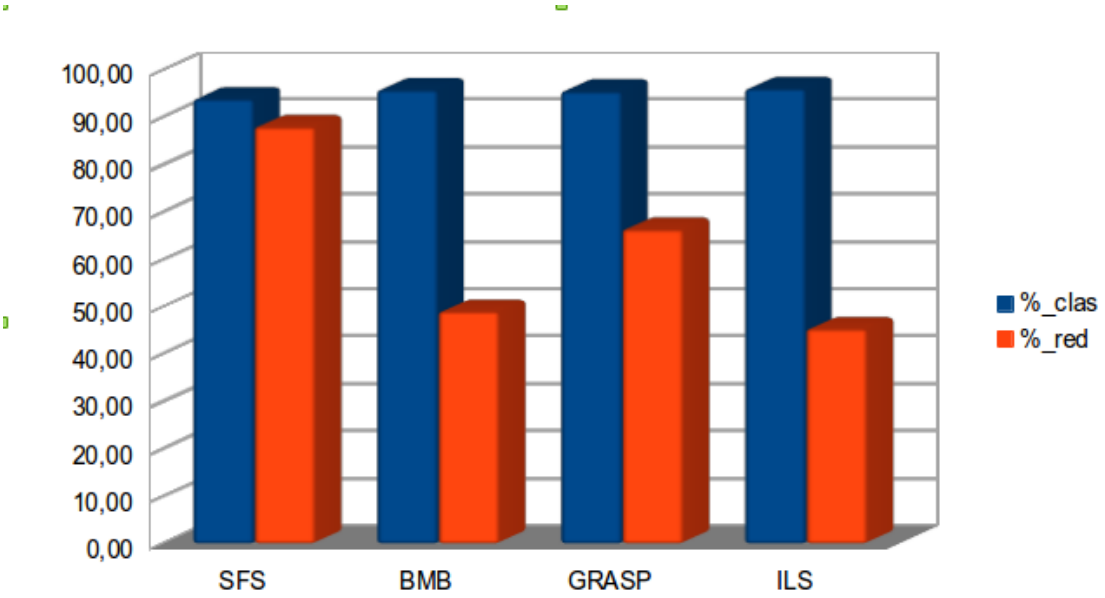

Figura 9.7: Gráfico de resultados para la base de datos WDBC

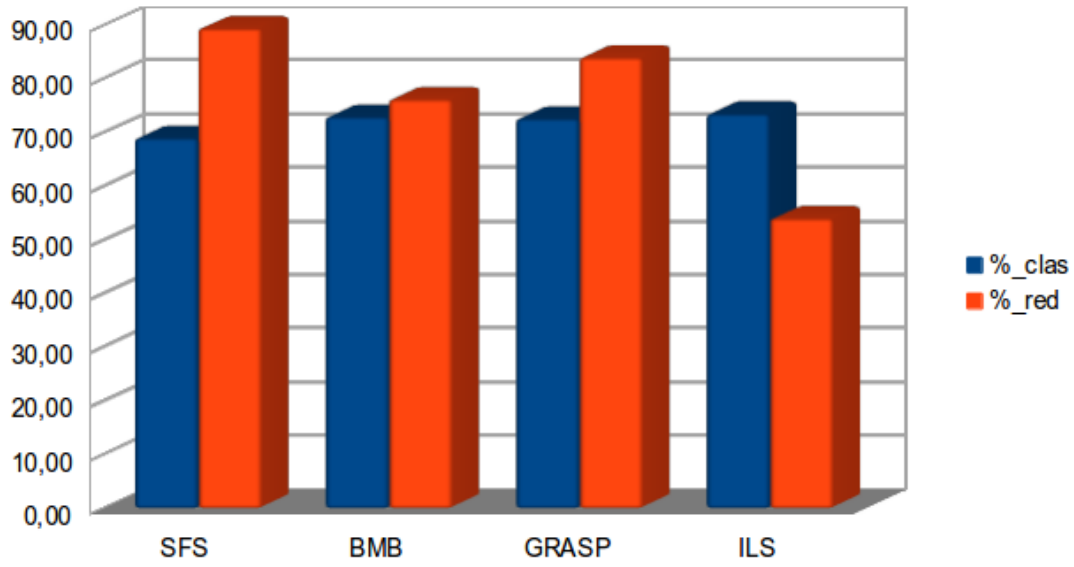

Figura 9.8: Gráfico de resultados para la base de datos movement libras

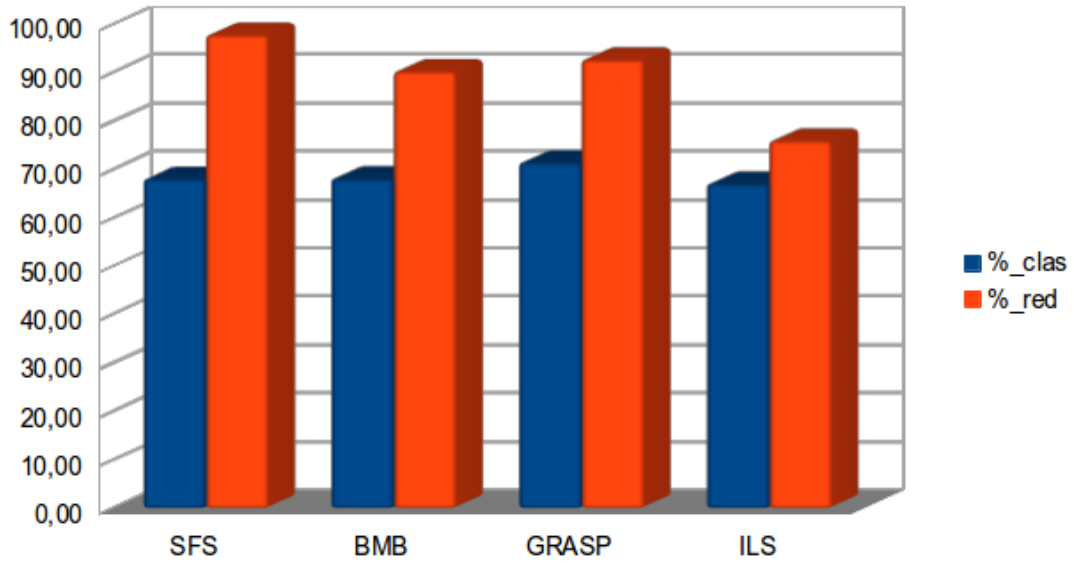

Figura 9.9: Gráfico de resultados para la base de datos arrhythmia