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

PRÁCTICA 5: BÚSQUEDAS CON TÉCNICAS HÍBRIDAS.

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

11 de julio 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 cada algoritmo he plantado el mismo valor de semilla para una correspondiente iteración.

8. 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
```

```
27
28  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:

```
1  seleccionar(poblacion, tasas_poblacion, seleccion, tasas) //Por
       referencia
2  begin
3    seleccion <- Array() //Array de cromosomas (vectores solucion)
4    tasas <- Array() //Tasas de los seleccionados
5
6    disponibles <- 0..poblacion.length-1 //Array de  ndices  para
         controlar que no compite un cromosoma consigo mismo
7
8    Para i=0 hasta poblacion.length
9    begin
10     aux <- disponibles
11
12     random1 <- Random(0, aux.length-1) //Random devuelve un entero
13     aux <- aux-{random1} //Me aseguro de que no cojo el mismo
           cromosoma
14     random2 <- Random(0, aux.length-1)
15
16     aux <- disponibles
17
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
```

```
  end

  devolver seleccion y tasas //Por referencia

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:

```
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
```

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:

```
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
```

El algoritmo completo en pseudocódigo es el 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
    //Guardo el mejor elemento de la poblacion
    indice_mejor <- maximo(tasas_poblacion)
    mejor <- poblacion[indice_mejor]

    Seleccion <- Array() //Array de vectores solucion
    Tasas_Seleccion <- Array() //Array de tasas


    //Fase de seleccion
    seleccionar(poblacion, tasas_polacion, Seleccion, Tasas_Seleccion
       ) //Por referencia

    //Fase de cruce
    n_cruces <- 0.7*30/2

    Para j=0 hasta n_cruces
    begin
      padre1 <- Seleccion[2*j]
      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 est  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      poblacion <- seleccion
69      tasas_poblacion <- tasas_seleccion
70    end
71
72    ind_max <- maximo(tasas_poblacion)
73    solucion <- poblacion[ind_max]
74    Tasa_Solucion <- Tasas_poblacion[ind_max]
75
76    devolver solucion y Tasa_Solucion //Por referencia
77
78  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 de búsqueda local

En la práctica hemos hibridado el algoritmo genético generacional con búsqueda local (con una sola iteración). La búsqueda local en pseudocódigo (de una sola iteración) es la siguiente:

```
busqueda_local(training, Solucion, Tasa_Solucion)
begin
  Tasa_Solucion <- funcion_objetivo(training, Solucion)

  S <- Solucion

  repetidos <- [0...S.length-1]

  Mientras que no se mejore y mientras que no se haya generado todo
     el entorno de S
  begin
    S <- flip(S, repetidos) //Con repetidos evitamos repetir dos
                              //soluciones de un mismo entorno

    coste_S <- funcion_objetivo(training, test, S)

    Si coste_S es mejor que Toste_Solucion entonces
    //S mejora a Solucion
      Solucion <- S
      Tasa_Solucion <- coste_S

  end

  devolver Solucion y Toste_Solucion //Por referencia

end
```

La descripción del generador de vecinos flip es la siguiente:

```
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
```

6. Algoritmo memético 1

En este algoritmo memético aplicaremos una búsqueda local a toda la población cada 10 generaciones. Llevaremos un contador de generaciones. Cuando llegue a 10 haremos búsqueda local a todos los cromosomas de la población y reinicializaremos la cuenta de generaciones. La descripción en pseudocódigo es la siguiente:

```
AM1(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


  generacion <- 0 //contador de generaciones

  Mientras no se realicen mas de 15000 evaluaciones
  begin
  //Guardo el mejor elemento de la poblacion
    indice_mejor <- maximo(tasas_poblacion)
    mejor <- poblacion[indice_mejor]

    Seleccion <- Array() //Array de vectores solucion
    Tasas_Seleccion <- Array() //Array de tasas


    //Fase de seleccion
    seleccionar(poblacion, tasas_polacion, Seleccion, Tasas_Seleccion
       ) //Por referencia

    //Fase de cruce
    n_cruces <- 0.7*30/2

    Para j=0 hasta n_cruces
    begin
      padre1 <- Seleccion[2*j]
      padre2 <- Seleccion[2*j+1]

      cruce(padre1, padre2)
    end

    //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 est 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      poblacion <- seleccion
69      tasas_poblacion <- tasas_seleccion
70
71      generacion <- generacion+1
72
73      Si generacion==10 entonces
74        Para cada soluion j de poblacion
75        begin
76          poblacion[j] <- busqueda_local(training, poblacion[j],
                tasas_poblacion[j]) //tasas_poblacion por referencia
77
78        end
79    end
80
81    ind_max <- maximo(tasas_poblacion)
82    solucion <- poblacion[ind_max]
83    Tasa_Solucion <- Tasas_poblacion[ind_max]
84
85  devolver solucion y Tasa_Solucion //Por referencia
86
87  end
```

7. Algoritmo memético 2

En este algoritmo también hibridaremos búsqueda local y el algoritmo genético generacional. Ahora la mutación se realizará sobre los cromosomas con una probabilidad p=0.1.

```
1  //PRE: Solucion inicialmente es entero falso
2
3  AM2(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   generacion <- 0
17
18   Mientras no se realicen mas de 15000 evaluaciones
19   begin
20     //Guardo el mejor elemento de la poblacion
21     indice_mejor <- maximo(tasas_poblacion)
22     mejor <- poblacion[indice_mejor]
23
24     Seleccion <- Array() //Array de vectores solucion
25     Tasas_Seleccion <- Array() //Array de tasas
26
27
28     //Fase de seleccion
29     seleccionar(poblacion, tasas_polacion, Seleccion, Tasas_Seleccion
           ) //Por referencia
30
31     //Fase de cruce
32     n_cruces <- 0.7*30/2
33
34     Para j=0 hasta n_cruces
35     begin
36       padre1 <- Seleccion[2*j]
37       padre2 <- Seleccion[2*j+1]
38
39       cruce(padre1, padre2)
40     end
41
42     //Fase de mutaci n
43     n_mutaciones <- 0.001*30*Seleccion[1].length
```

```
44
45      Para j=0 hasta n_mutaciones
46      begin
47        r1 <- Random(0, Seleccion.length-1)
48        r2 <- Random(0, Seleccion[r1].length-1)
49
50        mutar(Seleccion[r1], r2)
51      end
52
53
54      //Fase de reemplazamiento
55      Para j=0 hasta 30
56      begin
57        tasas_Seleccion[j] <- funcion_objetivo(training, seleccion[j])
58      end
59
60      encontrado <- Buscar(mejor) //Busco si est  la mejor solucion
           anteriormente guardada
61
62      Si !encontrado entonces
63        min <- minimo(tasas_seleccion)
64
65        Si tasas_seleccion[min] < tasas_poblacion[ind_mejor] entonces
66          seleccion[min] <- mejor
67          tasas_seleccion[min] <- tasas_poblacion[ind_mejor]
68
69      poblacion <- seleccion
70      tasas_poblacion <- tasas_seleccion
71
72      generacion <- generacion+1
73
74      Si generacion==10 entonces
75        Para cada solucion j de poblacion
76        begin
77          Si Rand() <= 0.1 entonces
78             poblacion[j] <- busqueda_local(training, poblacion[j],
                  tasas_poblacion[j]) //tasas_poblacion por referencia
79        end
80
81        generacion <- 0
82
83
84    end
85
86    ind_max <- maximo(tasas_poblacion)
87    solucion <- poblacion[ind_max]
88    Tasa_Solucion <- Tasas_poblacion[ind_max]
89
90    devolver solucion y Tasa_Solucion //Por referencia
```

```
91
92  end
```

8. Algoritmo memético 3

En este algoritmo memético aplicaremos búsqueda local solo a los 0.1*N mejores cromosomas, cada 10 generaciones también. La descripción en pseudocódigo es la siguiente:

```
1   //PRE: Solucion inicialmente es entero falso
2
3   AM3(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    generacion <- 0
17
18    Mientras no se realicen mas de 15000 evaluaciones
19    begin
20      //Guardo el mejor elemento de la poblacion
21      indice_mejor <- maximo(tasas_poblacion)
22      mejor <- poblacion[indice_mejor]
23
24      Seleccion <- Array() //Array de vectores solucion
25      Tasas_Seleccion <- Array() //Array de tasas
26
27
28      //Fase de seleccion
29      seleccionar(poblacion, tasas_polacion, Seleccion, Tasas_Seleccion
            ) //Por referencia
30
31      //Fase de cruce
32      n_cruces <- 0.7*30/2
33
34      Para j=0 hasta n_cruces
35      begin
36        padre1 <- Seleccion[2*j]
37        padre2 <- Seleccion[2*j+1]
38
```

```
39        cruce(padre1, padre2)
40      end
41
42  ha     //Fase de mutaci n
43      n_mutaciones <- 0.001*30*Seleccion[1].length
44
45      Para j=0 hasta n_mutaciones
46      begin
47        r1 <- Random(0, Seleccion.length-1)
48        r2 <- Random(0, Seleccion[r1].length-1)
49
50        mutar(Seleccion[r1], r2)
51      end
52
53
54      //Fase de reemplazamiento
55      Para j=0 hasta 30
56      begin
57        tasas_Seleccion[j] <- funcion_objetivo(training, seleccion[j])
58      end
59
60      encontrado <- Buscar(mejor) //Busco si  est  la mejor solucion
            anteriormente guardada
61
62      Si !encontrado entonces
63      min <- minimo(tasas_seleccion)
64
65        Si tasas_seleccion[min] < tasas_poblacion[ind_mejor] entonces
66          seleccion[min] <- mejor
67          tasas_seleccion[min] <- tasas_poblacion[ind_mejor]
68
69      poblacion <- seleccion
70      tasas_poblacion <- tasas_seleccion
71
72      Si generacion==10 entonces
73
74            tasas_aux <- tasas_poblacion
75            indices_mejores <- Array
76
77        Para j=0 hasta 0.1*poblacion.length
78        begin
79          //max devuelve el indice del maximo
80          ind <- max(tasas_aux)
81          indices_mejores.aniadir(ind)
82          tasas_aux[ind] <- -999999
83        end
84
85        Para j=0 hasta indices_mejores.length
86        begin
```

```
87          poblacion[indices_mejores[j]] <- busqueda_local(training,
                poblacion[indices_mejores[j]], tasas_poblacion[j])
88        end
89
90
91    end
92
93    ind_max <- maximo(tasas_poblacion)
94    solucion <- poblacion[ind_max]
95    Tasa_Solucion <- Tasas_poblacion[ind_max]
96
97    devolver solucion y Tasa_Solucion //Por referencia
98  end
```

Rand() genera un número aleatorio entre 0 y 1.

9. 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.

10. 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 10.1: Tabla de resultados 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 10.2: Tabla de resultados para SFS

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	94,39	66,67	198,48	72,78	38,89	224,50	65,29	47,84	810,73
Partición 1-2	96,13	33,33	199,22	75,00	44,44	224,79	61,14	71,22	702,05
Partición 2-1	95,79	33,33	204,23	68,89	52,22	230,78	69,95	68,35	835,40
Partición 2-2	94,72	26,67	199,17	75,00	55,56	202,18	67,36	69,42	849,07
Partición 3-1	96,14	50,00	197,49	72,78	31,11	217,91	67,88	69,78	718,12
Partición 3-2	96,83	33,33	209,23	79,44	35,56	223,18	65,29	55,40	616,10
Partición 4-1	95,08	36,67	198,48	70,00	34,44	237,56	64,77	54,32	688,28
Partición 4-2	95,77	16,67	199,28	71,67	51,11	218,46	64,25	70,86	785,07
Partición 5-1	94,04	33,33	208,74	78,89	54,44	224,04	59,59	56,53	745,83
Partición 5-2	98,59	30,00	198,94	75,00	35,56	215,45	63,73	59,35	711,21
Media	95,75	36,00	201,33	73,95	43,33	221,89	64,92	62,31	746,19

Figura 10.3: Tabla de resultados para el primer algoritmo memético

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	95,09	76,67	195,79	71,67	36,67	202,26	64,77	55,76	604,57
Partición 1-2	95,77	26,67	198,36	72,22	38,89	203,36	61,66	53,24	652,26
Partición 2-1	95,79	33,33	195,07	70,00	55,56	205,19	71,50	67,63	590,99
Partición 2-2	96,13	30,00	198,53	72,22	66,67	208,62	66,32	61,15	582,47
Partición 3-1	95,79	36,67	198,67	74,44	30,00	209,41	66,84	69,78	583,24
Partición 3-2	95,77	10,00	201,42	78,89	35,56	206,33	66,32	45,68	644,00
Partición 4-1	96,49	23,33	198,79	69,44	37,78	208,23	66,32	52,88	579,15
Partición 4-2	96,48	33,33	203,42	73,33	25,56	209,83	66,84	62,23	624,06
Partición 5-1	94,04	33,33	198,12	76,11	51,11	204,82	62,69	57,55	590,92
Partición 5-2	97,54	43,33	202,42	72,78	40,00	202,46	64,77	50,72	602,65
Media	95,89	34,67	199,06	73,11	41,78	206,05	65,80	57,66	605,43

Figura 10.4: Tabla de resultados para el segundo algoritmo memético

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	95,09	76,67	207,27	71,11	38,89	208,67	63,73	50,36	615,17
Partición 1-2	96,13	26,67	204,99	75,56	32,22	212,60	59,59	69,42	595,58
Partición 2-1	95,79	33,33	203,83	69,44	55,56	209,04	69,95	69,06	565,53
Partición 2-2	94,37	30,00	201,48	75,56	74,44	212,10	62,69	49,64	577,66
Partición 3-1	94,74	33,33	199,32	73,89	42,22	210,39	66,84	74,46	579,01
Partición 3-2	97,54	53,33	202,01	78,89	17,78	213,54	65,80	58,63	603,87
Partición 4-1	96,49	20,00	199,10	69,44	36,67	209,11	65,80	56,12	587,47
Partición 4-2	95,07	36,67	203,99	75,00	55,56	209,23	66,84	64,39	596,79
Partición 5-1	94,04	26,67	206,10	79,44	47,78	205,25	62,69	60,43	614,52
Partición 5-2	96,13	56,67	207,15	75,00	40,00	210,46	67,36	60,07	578,15
Media	95,54	39,33	203,52	74,33	44,11	210,04	65,13	61,26	591,38

Figura 10.5: Tabla de resultados para el tercer algoritmo memético

	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
AM-(10,1.0)	95,75	36,00	201,33	73,95	43,33	221,89	64,92	62,31	746,19
AM-(10,0.1)	95,89	34,67	199,06	73,11	41,78	206,05	65,8	57,66	605,43
AM-(10,0.1mej)	95,54	39,33	203,52	74,33	44,11	210,04	65,13	61,26	591,38

Figura 10.6: Tabla global de resultados

Como podemos observar, en la primera base de datos (WDBC) los algoritmos implementados no superan la tasa de clasificación de 3NN. Como ya he comentado en prácticas anteriores, WDBC tiene muy poco ruido y es muy posible que todas sus características o la mayoría de ellas sean

importantes. Esto justifica que cuanto más tasa de reducción haya, peor tasa de clasificación. El mejor resultado nos lo da el segundo algoritmo memético, el cual tiene la menor tasa de reducción.

En cuanto a la segunda base de datos, los tres algoritmos superan en tasa de clasificación a SFS y solo superan a 3NN el primer memético y el tercero. El hecho de que el primer algoritmo y el tercero obtengan mejores resultados que el segundo podría deberse a que el segundo puede realizar la explotación sobre soluciones no tan buenas. El primero realiza la explotación sobre todas las soluciones (incluyendo las mejores) y el tercero sobre las mejores.

En la tercera base de datos vemos una contradicción respecto a lo explicado en el anterior párrafo, ya que ahora el segundo algoritmo es mejor que los demás (aunque no superan a SFS). La posible explicación que le veo es que al realizar búsqueda local sobre soluciones que no son necesariamente las mejores, crea más diversidad, es decir, explota soluciones posiblemente peores y evita caer en los óptimos locales de las mejores. Podría tener sentido esta explicación ya que el campo de búsqueda de la base de datos de arrhythmia es muy grande. Al ser tan ruidosa esta base de datos, SFS obtiene mejores resultados (con bastante diferencia) que los demás algoritmos. SFS empieza explorando una solución vacía, por lo que su tasa de reducción es muy alta y al ser tan ruidosa la BD esto le da más calidad.