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

PRÁCTICA 4: OPTIMIZACIÓN BASADA EN COLONIAS DE HORMIGAS PARA LA SELECCIÓN DE CARACTERÍSTICAS

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

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

7. 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. Función heurística y rastros de feromona

La función heurística con la que valoraremos la calidad de cada una de las características es la dada en las transparencias del seminario:

$$\eta_{fi} = I(C,f_i) = H(C) - H(C/f_i) = \sum_{c=1}^{N_c} \sum_{f_{ij}=1}^{N_{fi}} P(c,f_{ij}) \cdot \log_2\left(\frac{P(c,f_{ij})}{P(c)\cdot P(f_{ij})}\right)$$

Donde:

- c = 1, ..., Nc son las clases del problema
- j = 1, ..., N_{fi} son los valores discretos de f_i. Si la característica es continua se discretiza en h intervalos (p.ej., h = 10).
- $P(c, f_{ij})$ es la probabilidad de que un ejemplo del conjunto de entrenamiento sea de clase c cuando la variable f_i vale f_{ij}.
- $P(c)$ es la probabilidad de la clase c en el conjunto de entrenamiento
- $P(f_{ij})$ es la probabilidad del valor fij en el conjunto de entrenamiento

A continuación describiré la implementación de la función heurística en pseudocódigo:

```
heuristica(conjunto_datos, v_heuristico, labels)
begin
  v_heuristico <- rep(conjunto_datos.caracteristicas.length, 0.0)
  n_class <- rep(labels.length, 0)

  Para cada vector de caracteristicas i del conjunto de datos
  begin
    Para cada etiqueta j de labels
    begin
      Si i.label == j entonces
        n_class[j] <- n_class[j]+1
    end
  end

  Para cada caracteristica i de cualquier vector de caracteristicas
     del conjunto de datos
  begin
    //Extraigo la caracteristica i de todos los datos
    //del conjunto de datos
    v_auxiliar <- conjunto_datos[,i]

```

```
21      max <- maximo(v_auxiliar)
22      min <- minimo(v_auxiliar)
23
24      factor <- (max-min)/10.0
25
26      matriz <- matrix(labels.lengthx10, 0)
27
28      //Esta parte la aclaro redactando mas abajo
29      Para cada etiqueta j de labels
30      begin
31
32        encontrado <- false
33        Para k=1 hasta k==10 && !encontrado
34        begin
35          Si conjunto_datos[,i] <= min+factor*k && j==conjunto_datos[,i
              ].label entonces
36            matriz[j,k-1] <- matriz[j,k-1]+1
37            encontrado <- true
38
39        end
40      end
41
42      Para cada etiqueta j en labels
43      begin
44        Para k=0 hasta 10
45        begin
46          p_i_j <- 0.0
47
48          Para h=0 hasta matriz.length
49          begin
50            p_i_j <- matriz[h,k]+p_i_j
51          end
52
53          //Probabilidad de ser de clase c en el intervalo
54          //discretizado ij
55          Si p_i_j==0 entonces
56            p_c_f <- 0
57          si no
58            p_c_f <- matriz[j,k]/p_i_j
59
60          //Probabilidad de ser del intervalo discretizado
61          p_i_j <- p_i_j/conjunto_datos.length
62
63          //probabilidad de ser de una clase
64          p_c <- n_class[j]/conjunto_datos.length
65
66          Si p_c_f != 0 entonces
67            v_heuristico[i] <- v_heuristico[i]+ p_c_f*log2(p_c_f/(p_c*
                p_i_j))
```

```
68          si no
69            v_heuristico[i] <- v_heuristico[i]+0
70
71        end
72      end
73    end
74
75    devolver v_heuristico //por referencia
76  end
```

Redactando un poco el procedimiento. Primero calculo qué número de etiquetas (de cada etiqueta) hay en el conjunto de datos. Después para cada etiqueta i de cualquier vector de características comienzo a calcular su valor heurístico. Para ello calculo cual es el máximo y el mínimo de entre los valores de esta característica para todos los datos. Calculo el factor para discretizar en 10 intervalos y calculo para cada etiqueta cuántas hay en cada intervalo, guardando estas cifras en una matriz que tiene tantas filas como posibles etiquetas y 10 columnas (una por cada intervalo). Después de esto ya incremento el valor heurístico siguiendo la expresión que he dado anteriormente.

En cuanto a la feromona tendremos dos tipos distintos. En mi código lo he representado como 2 arrays distintos. Uno de ellos es la feromona para elegir el número de características (feromona_n_car) que la hormiga seleccionará. El otro array será la feromona para la selección de cada una de las características disponibles (feromona_car).

5. Algoritmos basados en colonias de hormigas

Los algoritmos implementados han sido SHMMBL (Sistema de Hormigas Min-Max + Búsqueda Local) y SCHBL (Sistema de Colonia de Hormigas + Búsqueda local). Los aspectos comunes de los dos algoritmos son los siguientes:

- Ambos algoritmos usan la misma representación especificada anteriormente
- El criterio de parada de ambos algoritmos es no realizar más de 15000 evaluaciones de la función objetivo
- Ambos usan como función heurística la especificada en el apartado anterior.
- Los dos algoritmos tendrán los rastros de feromona especificados en el apartado anterior.
- En los dos algoritmos lanzaremos 10 hormigas. El valor de ρ será 0.2 mientras que $\alpha y \beta$ valdrán 1 y 2 respectivamente.
- Ambos algoritmos usarán una búsqueda local de una sola iteración para mejorar las soluciones dadas por cada hormiga.

- En la búsqueda local de una sola iteración que realizo en ambos algoritmos uso la función flip descrita como:

```
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
```

A continuación analizaremos los 2 algoritmos por separado especificando los parámetros concretos para cada algoritmo (los que no son comunes).

5.1. SCHBL

En el algoritmo de Sistema de Colonias de Hormigas + Búsqueda Local usaremos los siguientes parámetros:

- El valor inicial del rastro de feromona feromona_car (feromonas para la selección de cada una de las características) será 10^{-6}.
- El valor inicial del rastro de feromona feromona_n_car (para seleccionar el número de características que seleccionaremos) será $1/N_c$ donde N_c será el número de características de cada vector de características.
- Para el proceso constructivo usaremos la regla siguiente:

$$s = \begin{cases} \arg\max_{u \in J_k(r)} \{[\tau_{ru}]^{\alpha} \cdot [\eta_{ru}]^{\beta}\}, & \text{si } q \leq q_0 \\ S, & \text{en otro caso} \end{cases}$$

  donde S será una característica cogida simulando una ruleta con una distribución de probabilidad. q_0 además es una cota de probabilidad. Si generamos un aleatorio y es menor o igual que q_0 entonces cogeremos la característica con más probabilidad, si no, haremos la ruleta.
- El parámetro q_0 será 0.8.

- El aporte de feromona a realizar será la tasa de clasificación de la solución dada por la hormiga.
- El valor de φ será 0.2.

En este algoritmo se realizarán 2 actualizaciones de feromona, una local y otra global. La local se realiza sólo sobre feromona_car y se realiza cada vez que una hormiga selecciona una característica. Se describe con la expresión siguiente:

$$\tau_{rs}(t) = (1-\varphi)\cdot\tau_{rs}(t-1) + \varphi\cdot\tau_0$$

La actualización global la realizaremos según la mejor solución encontrada hasta el momento y se realizará después de que todas las hormigas hayan hecho su recorrido. La haremos tanto en feromona_car como en feromona_n_car. Seguiremos la siguiente expresión:

$$\tau_{rs}(t) = (1-\rho)\cdot\tau_{rs}(t-1) + \rho\cdot\Delta\tau_{rs}^{mejor\_global}$$

Donde la aportación de feromona será el valor de la función objetivo (la tasa de clasificación) de la mejor solución encontrada hasta el momento.

Para actualizar feromona_car, sólo aportaremos feromona en las características que estén seleccionadas por la mejor solución encontrada hasta el momento. Por otro lado para actualizar feromona_n_car, contaremos el número de características de la mejor solución y aportaremos feromona sobre la posición correspondiente a este número de características de feromona_n_car. A continuación, la descripción en pseudocódigo del algoritmo:

```
SCHBL(training, solucion, tasa_solucion, heuristica)
begin

  //Inicializo las feromonas
  //En la feromona del numero de caracteristicas
  //la posicion i correspondera a coger i+1
  //caracteristicas
  feromona_car <- rep(solucion.length, 1e-6)
  feromona_n_car <- rep(solucion.length, 1/solucion.length)

  feromona_n_car_total <- 1.0

  evaluaciones <- 0

  Mientras que evaluaciones < 15000
  begin

    //candidatos sera una matriz que contendra
```

```
19      //la lista de candidatos para cada hormiga
20      //sera un vector de vectores de enteros
21      candidatos <- Matrix(10xsolucion.length, 0..solucion.length)
22
23      //grafos sera una matriz que contendra
24      //cada uno de los vectores solucion dados
25      //por cada hormiga. INicialmente inicializadas
26      //a false
27      grafos <- Matrix(10xsolucion.length, false)
28
29      //En n_car guardaremos el numero de caracteristicas
30      //que seleccionara la hormiga i-esima y en n_car
31      //llevaremos un contador de cuantas caracteristicas
32      //ha cogido la hormiga i-esima. Ambos arrays son de
33      //tamanio 10
34      n_car <- Array(10)
35      n_car_seleccionadas <- Array(10)
36
37
38      //Pasamos a la seleccion del numero de caracteristicas
39      //que cada hormiga cogera. Lo haremos
40      //simulando que giramos una ruleta
41
42      Para cada hormiga i
43      begin
44
45        aleatorio <- Random(0,feromona_n_car_total)
46        fin <- false
47
48        Para cada posicion j de feromona_n_car && !fin
49        begin
50          aleatorio <- aleatorio-feromona_n_car[j]
51
52          Si aleatorio <= 0 entonces
53            //La hormiga i cogera j+1 caracteristicas
54            n_car[i] <- j+1
55            fin <- true
56
57        end
58      end
59
60
61      //Tengo que saber que hormiga cogera mas
62      //caracteristicas
63      //maximo devuelve el indice del maximo
64      n_car_max <- maximo(n_car)
65      n_car_max <- n_car[n_car_max]
66
67      Para i=0 hasta n_car_max
```

```
68      begin
69        Para cada una de las hormigas j
70        begin
71
72          //Controlo que la hormiga no elija mas
73          //caracteristicas de las que debe
74          Si n_car_seleccionadas[j] < n_car[j] entonces
75
76            //Inicializo un vector de probs
77            probs <- rep(solucion.length, 0.0)
78
79            suma_probabilidades <- 0.0
80
81            //Recalculo las probabilidades
82            //solamente para los candidatos
83            //Tengo que tener en cuenta los parametros
84            //alpha y beta (2 y 1)
85
86            probs[candidatos] <- heuristic[candidatos]*heuristic[
                 candidatos]*feromona_car[candidatos]
87
88            suma_probabilidades <- sum(probs)
89
90            r <- Rand() //Aleatorio entre 0 y 1
91
92
93            Si r <= 0.8 entonces
94              //Fase de seleccion de SCH
95              //Cojo la caracteristica con mas probabilidad
96              mejor_car <- maximo(probs)
97              mejor_car <- probs[mejor_car]
98
99              grafos[j,mejor_car] <- true
100
101             //Aqui realizo la evaporacin local de la
102             // feromona
103             feromona_car <- 0.8*feromona_car[mejor_car]+0.2*1e-6
104
105             //Le quito el elemento mejor_car a candidatos
106             candidatos <- candidatos-{mejor_car}
107
108           Si no entonces
109             //Fase de seleccion de SH
110             //Hago una ruleta con una distribucion de
111             //probabiliad
112
113             aleatorio <- Random(0,suma_probabilidades)
114             fin <- false
115             index <- -1
```

```
116
117             Para cada candidato k && !fin
118             begin
119               aleatorio <- aleatorio-probs[k]
120
121               Si aleatorio <= 0 entonces
122                 index <- k
123                 fin <- true
124
125             end
126
127             grafos[j,index] <- true
128
129             //Aqui realizo la evaporacin local de la
130             // feromona
131             feromona_car <- 0.8*feromona_car[index]+0.2*1e-6
132
133             //Le quito el elemento index a candidatos
134             candidatos <- candidatos-{index}
135
136           n_car_seleccionadas[j]++
137           //Fin del if
138       end
139     end
140
141     //Ahora realizaremos unaa busqueda local (con una
142     //sola iteracion)
143     //para cada solucion dada
144     //por cada una de las hormigas
145
146     Para i=0 hasta 10
147     begin
148       coste_s_act <- funcion_objetivo(training, grafos[i,])
149       evaluaciones++
150
151
152       repetidos <- [0...S.length-1]
153
154       Mientras que no se mejore y mientras que no se haya generado
            todo el entorno de S
155       begin
156         S <- flip(grafos[i,], repetidos) //Con repetidos evitamos
              repetir dos                    //soluciones de un mismo
              entorno
157
158         coste_S <- funcion_objetivo(training, S)
159         evaluaciones++
160
161         Si coste_S es mejor que coste_s_act entonces
```

```
162           //S mejora a Solucion
163             grafos[i,] <- S
164             coste_solucion <- coste_S
165
166         end
167
168         Si coste_s_act >= tasa_solucion entonces
169           solucion <- grafos[i,]
170           tasa_solucion <- coste_s_act
171       end
172
173       //A continuacion procederemos a la actualizacion
174       //global de la feromona
175
176       //Actualizacion de feromona_car
177       Para cada componente i de feromona_car
178       begin
179         fermona_car[i] <- 0.8*feromona_car[i]+0.2*solucion[i]*
              tasa_solucion
180
181       end
182
183       //Actualizacion de feromona_n_car
184       //Primero cuento cuantas caracteristicas
185       //tiene la mejor solucion
186
187       mejor_n_car <- 0
188
189       Para i=0 hasta solucion.length
190       begin
191         mejor_n_car <- mejor_n_car+1*solucion[i]
192       end
193
194       feromona_n_car_total <- 0
195
196       Para cada componente i de feromona_n_car
197       begin
198         feromona_n_car[i] <- 0.8*feromona_n_car[i]+0.2*(mejor_n_car==i
              +1)*tasa_solucion
199
200         feromona_n_car_total <- feromona_n_car_total+feromona_n_car[i]
201
202       end
203
204     end
205
206     devolver solucion y tasa_solucion
207
208   end
```

En la implementación, candidatos es un array de arrays de enteros que representan los índices de los vectores de características, donde cada 'fila' i de la matriz (de candidatos) es la lista de candidatos de la hormiga i. Siempre que cojo una característica elimino este candidato de la lista. La búsqueda local la hago con una sola iteración. Además, si mejora, la búsqueda local acaba. Esta condición de parada no estaba muy seguro de si era correcta, pero al final la he dejado. Con el vector repetidos me aseguro de que ya se ha explorado todo el entorno de la solución (o no). La función flip ya la he explicado en pseudocódigo anteriormente.

5.2. SHMMBL

En este algoritmo usaremos los siguientes parámetros y decisiones:

- El valor inicial de feromona será $\tau_{max} = C(S_{aleatoria})/\rho$ para feromona_car (que será la cota máxima de feromona) y la cota inferior de feromona será $\tau_{max}/500$.
- Cada vez que encontremos una solución que supere a la mejor global, actualizaremos $\tau_{max} = C(S_{mejor})/\rho$ y $\tau_{min} = \tau_{max}/500$.
- En este algoritmo sólo se hará actualización de la feromona de forma global (no habrá actualización local).
- Para feromona_n_car no he usado las cotas de feromona.
- En el proceso constructivo se usa la regla de selección de SH. Se hace una ruleta con una distribución de probabilidad y elegimos según esa ruleta.

La actualización de este algoritmo se basa en la regla:

$$\tau_{rs}(t) = (1-\rho)\cdot\tau_{rs}(t-1) + \Delta\tau_{rs}^{mejor}$$

es decir, para actualizar la feromona sólo nos fijaremos en la mejor solución encontrada hasta el momento. Además, como he dicho anteriormente, tendremos una cota de feromona con la que haremos que la feromona no sea ni demasiado alta ni demasiado baja, permitiendo así algo más de diversidad. Además, estas cotas se irán actualizando según vayamos encontrando mejores soluciones globales, como he explicado anteriormente.

La descripción en pseudocódigo es la siguiente:

```
1  SHMMBL(training, solucion, tasa_solucion, heuristica)
2  begin
3
4    //Primero generamos una solucion aleatoria
5    solucion <- generar_solucion_aleatoria(solucion, solucion.length)
```

```
6     tasa_solucion <- funcion_objetivo(training,solucion)
7
8     //Inicializo las feromonas
9     //En la feromona del numero de caracteristicas
10    //la posicion i correspondera a coger i+1
11    //caracteristicas
12
13    max_feromona <- tasa_solucion/0.2
14    min_feromona <- max_feromona/500.0
15
16    feromona_car <- rep(solucion.length, max_feromona)
17    feromona_n_car <- rep(solucion.length, 1/solucion.length)
18
19    feromona_n_car_total <- 1.0
20
21    evaluaciones <- 0
22
23    Mientras que evaluaciones < 15000
24    begin
25
26      //candidatos sera una matriz que contendra
27      //la lista de candidatos para cada hormiga
28      //sera un vector de vectores de enteros
29      candidatos <- Matrix(10xsolucion.length, 0..solucion.length)
30
31      //grafos sera una matriz que contendra
32      //cada uno de los vectores solucion dados
33      //por cada hormiga. INicialmente inicializadas
34      //a false
35      grafos <- Matrix(10xsolucion.length, false)
36
37      //En n_car guardaremos el numero de caracteristicas
38      //que seleccionara la hormiga i-esima y en n_car
39      //llevaremos un contador de cuantas caracteristicas
40      //ha cogido la hormiga i-esima. Ambos arrays son de
41      //tamanio 10
42      n_car <- Array(10)
43      n_car_seleccionadas <- Array(10)
44
45
46      //Pasamos a la seleccion del numero de caracteristicas
47      //que cada hormiga cogera. Lo haremos
48      //simulando que giramos una ruleta
49
50      Para cada hormiga i
51      begin
52
53        aleatorio <- Random(0,feromona_n_car_total)
54        fin <- false
```

```
55
56          Para cada posicion j de feromona_n_car && !fin
57          begin
58            aleatorio <- aleatorio-feromona_n_car[j]
59
60            Si aleatorio <= 0 entonces
61            //La hormiga i cogera j+1 caracteristicas
62              n_car[i] <- j+1
63              fin <- true
64
65          end
66        end
67
68
69        //Tengo que saber que hormiga cogera mas
70        //caracteristicas
71        //maximo devuelve el indice del maximo
72        n_car_max <- maximo(n_car)
73        n_car_max <- n_car[n_car_max]
74
75        Para i=0 hasta n_car_max
76        begin
77          Para cada una de las hormigas j
78          begin
79
80            //Controlo que la hormiga no elija mas
81            //caracteristicas de las que debe
82            Si n_car_seleccionadas[j] < n_car[j] entonces
83
84            //Inicializo un vector de probs
85              probs <- rep(solucion.length, 0.0)
86
87              suma_probabilidades <- 0.0
88
89              //Recalculo las probabilidades
90              //solamente para los candidatos
91              //Tengo que tener en cuenta los parametros
92              //alpha y beta (2 y 1)
93
94              probs[candidatos] <- heuristic[candidatos]*heuristic[
                    candidatos]*feromona_car[candidatos]
95
96              suma_probabilidades <- sum(probs)
97
98
99              //Fase de seleccion de SH
100             //Hago una ruleta con una distribucion de
101             //probabiliad
102
```

```
103             aleatorio <- Random(0,suma_probabilidades)
104             fin <- false
105             index <- -1
106
107             Para cada candidato k && !fin
108             begin
109               aleatorio <- aleatorio-probs[k]
110
111               Si aleatorio <= 0 entonces
112                 index <- k
113                 fin <- true
114
115             end
116
117             grafos[j,index] <- true
118
119             //Le quito el elemento index a candidatos
120             candidatos <- candidatos-{index}
121
122             n_car_seleccionadas[j]++
123             //Fin del if
124         end
125       end
126
127       //Ahora realizaremos unaa busqueda local (con una
128       //sola iteracion)
129       //para cada solucion dada
130       //por cada una de las hormigas
131
132       Para i=0 hasta 10
133       begin
134         coste_s_act <- funcion_objetivo(training, grafos[i,])
135         evaluaciones++
136
137
138         repetidos <- [0...S.length-1]
139
140         Mientras que no se mejore y mientras que no se haya generado
              todo el entorno de S
141         begin
142           S <- flip(grafos[i,], repetidos) //Con repetidos evitamos
                 repetir dos                  //soluciones de un mismo
                 entorno
143
144           coste_S <- funcion_objetivo(training, S)
145           evaluaciones++
146
147           Si coste_S es mejor que coste_s_act entonces
148             //S mejora a Solucion
```

```
149             grafos[i,] <- S
150             coste_solucion <- coste_S
151
152         end
153
154         Si coste_s_act >= tasa_solucion entonces
155           solucion <- grafos[i,]
156           tasa_solucion <- coste_s_act
157
158           //Actualizo las cotas de feromona
159           max_feromona <- tasa_solucion/0.2
160           min_feromona <- max_feromona/500
161       end
162
163       //A continuacion procederemos a la actualizacion
164       //de la feromona
165
166       //Actualizacion de feromona_car
167       Para cada componente i de feromona_car
168       begin
169         fermona_car[i] <- 0.8*feromona_car[i]+solucion[i]*tasa_solucion
170
171         Si feromona_car[i] > max_feromona entonces
172           feromona_car[i] <- max_feromona
173         Si no si feromona_car[i] < min_feromona entonces
174           feromona_car[i] <- min_feromona
175       end
176
177       //Actualizacion de feromona_n_car
178       //Primero cuento cuantas caracteristicas
179       //tiene la mejor solucion
180
181       mejor_n_car <- 0
182
183       Para i=0 hasta solucion.length
184       begin
185         mejor_n_car <- mejor_n_car+1*solucion[i]
186       end
187
188       feromona_n_car_total <- 0
189
190       Para cada componente i de feromona_n_car
191       begin
192         feromona_n_car[i] <- 0.8*feromona_n_car[i]+*(mejor_n_car==i+1)*
              tasa_solucion
193
194         feromona_n_car_total <- feromona_n_car_total+feromona_n_car[i]
195
196       end
```

```
197
198     end
199
200     devolver solucion y tasa_solucion
201
202 end
```

6. Experimentos y análisis

A continuación las gráficas de resultados para cada algoritmo:

	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 6.1: 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 6.2: Resultados para SFS

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	97,19	10,00	203,63	68,33	12,22	209,67	67,36	74,46	649,56
Partición 1-2	96,83	6,67	209,93	78,33	33,33	203,86	61,66	49,28	657,65
Partición 2-1	96,84	10,00	204,75	70,56	28,89	208,21	68,91	67,63	571,23
Partición 2-2	96,13	40,00	206,94	72,22	56,67	205,10	63,21	45,68	598,63
Partición 3-1	95,79	3,33	203,86	72,78	2,22	209,37	69,95	79,49	593,09
Partición 3-2	96,48	26,67	210,47	76,11	63,33	212,10	66,32	48,56	605,10
Partición 4-1	96,14	13,33	203,71	70,00	64,44	209,88	73,06	96,04	615,66
Partición 4-2	95,07	26,67	212,40	71,67	1,11	212,33	69,95	83,09	563,81
Partición 5-1	94,39	50,00	204,39	76,11	46,67	209,47	64,77	91,00	579,99
Partición 5-2	97,54	13,33	206,60	75,56	14,44	210,59	64,77	60,79	590,14
Media	96,24	20,00	206,67	73,17	32,33	209,06	67,00	69,60	602,49

Figura 6.3: Resultados para SCHBL

	Wdbc			Movement_Libras			Arritmia		
	%_clas	*%_red*	T	*%_clas*	*%_red*	T	*%_clas*	*%_red*	T
Partición 1-1	96,14	33,33	208,79	68,89	35,56	201,45	65,29	60,79	617,24
Partición 1-2	95,77	30,00	215,87	79,44	17,78	200,54	69,43	91,37	602,06
Partición 2-1	96,84	23,33	202,69	69,44	14,44	200,80	65,23	66,19	579,82
Partición 2-2	96,83	13,33	205,24	75,56	20,00	202,42	65,29	29,14	578,02
Partición 3-1	95,79	6,67	204,39	72,22	23,33	206,85	69,43	84,17	571,87
Partición 3-2	97,18	13,33	209,91	91,11	61,11	202,04	63,73	71,22	560,05
Partición 4-1	96,49	36,67	206,35	66,67	60,00	207,35	62,69	76,26	596,93
Partición 4-2	95,07	50,00	203,63	71,67	8,89	201,24	69,43	84,17	564,53
Partición 5-1	95,44	23,33	202,08	78,33	26,67	199,04	66,32	78,06	577,98
Partición 5-2	98,59	16,67	206,02	73,89	60,00	206,77	66,32	63,67	575,17
Media	96,41	24,67	206,50	74,72	32,78	202,85	66,32	70,50	582,37

Figura 6.4: Resultados para SHMMBL

Como vemos en la base de datos de Wdbc, ambos algoritmos de hormigas mejoran los resultados del algoritmo de comparación (SFS). En cambio, en el 3NN, obtenemos mejores resultados que en SCHBL. Esto podría deberse a que esta base de datos apenas tiene características ruidosas. SFS tiene una tasa de clasificación muy baja en comparación con los dos algoritmos basados en el comportamiento de hormigas. Como ya he dicho, esta base de datos por lo visto es poco ruidosa. EN SFS tenemos una tasa de reducción del 88 % por lo que hemos quitado muchas características y posiblemente algunas importantes. En las dos metaheurísticas reducimos el vector de características bastante menos, y por esto podría ser por lo que nos da mucho mejores resultados. El hecho de que en SHMMBL de resultados algo mejores en esta base de datos podría deberse a que este algoritmo intensifica mucho sobre la mejor solución (más que SCHBL), por lo que probablemente obtenga soluciones mejores.

En la segunda base de datos también las dos metaheurísticas mejoran los resultados de SFS. También puede deberse a una menor tasa de reducción, lo que nos puede hacer pensar a que esta base de datos es, como la anterior, poco ruidosa. Podemos confirmar esta teoría al ver el resulta-

do de 3NN, que sin reducir ninguna característica (obviamente) mejora también SFS. SHMMBL también mejora a todos los algoritmos, y se podría deber a, como he comentado antes, la intensificación sobre las buenas soluciones. En cuanto a los tiempos, esta base de datos tarda menos que la primera a pesar de tener más características, y podría deberse a que tiene menos instancias.

En la tercera base de datos cambia el contexto. Arrhythmia es una base de datos con gran número de características y probablemente mucho ruido. Esto lo confirma el hecho de que SFS obtenga mejor tasa de clasificación que 3NN. SFS también obtiene mejores resultados que los dos algoritmos implementados. Al comenzar a explorar desde una solución 'vacía', elimina muchas más características (cae en un óptimo local pronto) y muy probablemente muy ruidosas. Entre los dos algoritmos implementados, SHMMBL presenta una tasa de clasificación media algo menor que SCHBL y creo que se debe a que SCHBL permite más diversificación que SHMMBL, que intensifica más sobre las buenas soluciones. El hecho de que diversifique más es bueno cuando los entornos de búsqueda son tan amplios como en esta base de datos.