

# PRÁCTICA 1: BÚSQUEDA POR TRAYECTORIAS PARA EL PROBLEMA DE LA SELECCIÓN DE CARACTERÍSTICAS

---

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

9 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 ejecución 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:

$$tasa_{class} = 100 * \frac{n^{\circ}instanciasbienclasificadas}{n^{\circ}instanciasTotal}$$

## 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:

```
1  funcion_objetivo(conjunto_training, conjunto_test,
2     características_activas)
3  begin
4     Para todo elemento i del conjunto_test
5     begin
6         elemento <- elemento i del conjunto_test
7         clase <- 3NN(conjunto_training, elemento,
8             características_activas)
9
10         Si la clase estimada por 3NN se corresponde a la clase
11             real -> aciertos++
12     end
13
14     promedio <- aciertos/tamaño conjunto_test
15
16     devolver promedio
17 end
```

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

El pseudocódigo es el siguiente:

```
1  3NN(conjunto_training, vector_caracteristicas,
2     características_activas)
3  begin
4     Para cada vector i de características de training
5     begin
6         array_distancias.aadir(distanciaeuclidea(i,
7             vector_caracteristicas, características_activas))
8     end
```

```

9
10     minimo1 <- minimo(array_distancias)
11     minimo2 <- minimo(array_distancias-minimo1)
12     minimo3 <- minimo(array_distancias-minimo1-minimo2)
13
14     Si la clase de vector_caracteristicas[minimo2]==clase de
       vector_caracteristicas[minimo3] entonces
15         La clase del vector de caracteristicas es esa
16     Si no
17         La clase del vector de caracteristicas es la clase de
       vector_caracteristicas[minimo1]
18
19     devolver clase del vector de caracteristicas
20
21 end

```

4. Antes de trabajar con cualquier algoritmo hay que normalizar los conjuntos de datos.
5. Todos los algoritmos (menos SFS) tendrán como criterio de parada que se hayan explorado como mucho 15000 soluciones distintas. En la búsqueda tabú he reducido a 250 este número por el mucho tiempo que tarda. En la búsqueda local se añade como condición que cuando no se mejore la solución, pare. En SFS la condición es mientras la solución mejore.
6. El operador de generación de vecinos en búsqueda local, búsqueda tabú y enfriamiento simulado 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.
7. La solución inicial en SFS será el vector solución puesto entero a false. En cambio en los demás algoritmos se generará una solución aleatoria.
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 búsqueda local

La descripción en pseudocódigo del algoritmo es la siguiente:

```
1  busqueda_local(training, test)
2  begin
3      Solucion <- Generar_solucion_aleatoria
4      coste_solucion <- funcion_objetivo(training, test, Solucion)
5
6      Mientras que se encuentre una solución mejor y no se superen 15000
          soluciones exploradas
7      begin
8          S <- Solucion
9
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 coste_solucion entonces
17                 //S mejora a Solucion
18                 Solucion <- S
19                 coste_solucion <- coste_S
20
21         end
22     end
23
24     devolver Solucion y coste_solucion
25
26
27 end
```

La exploración del entorno la realizo cambiando aleatoriamente una componente del vector solución (con la función flip). Para asegurarme de que esta solución nueva del entorno no se repite dentro de este entorno, tengo un vector de booleanos (repetidos) que la función flip se encarga de comprobar. Si es true para el índice generado aleatoriamente, vuelvo a generar otro número aleatorio. Así hasta que llegue a una solución no explorada del entorno. 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:

```
1 Flip(Solucion, Repetidos)
2 begin
3
4   Mientras no sea valido
5     a <- Aleatorio entre 0 y (tamano de Solucion)-1
6     Si Repetidos[a] es falso entonces
7       Cambio Solucion[a] a falso si es verdadero o a verdadero si es
         falso
8       pongo Repetidos[a] a verdadero
9       pongo valido a verdadero
10    end
11
12    devolver Solucion
13 end
```

Para la exploración del entorno en general realizo lo siguiente:

```
1 Mientras que no genere todos los vecinos y no se mejore la solucion
2   S <- Mejor Solucion
3   S <- flip(S, repetidos)
4
5   Si la componente cambiada por flip no es la que devolveria a S a
     ser la solucion anteriormente explorada entonces
6     Coste_S <- funcion_objetivo(training, test, S)
7
8     Si Coste_S es mejor que el coste de la mejor solucion
9     entonces
10      actualizo la mejor solucion
11      paso a explorar entorno de la nueva mejor solucion
12
13
14   Si el contador de soluciones es igual al tama o de S-1
15     entonces
16     He explorado todo el entorno de S
17 end
```

## 4. Algoritmo de enfriamiento simulado

La descripción del algoritmo en pseudocódigo es la siguiente:

```
1  enfriamiento_simulado(training, test)
2  begin
3      Solucion <- Generar_solucion_aleatoria
4      coste_solucion <- funcion_objetivo(training, test, Solucion)
5      S <- Solucion
6      coste_S <- coste_solucion
7
8      T0 <- Inicializacin T0
9      T <- T0
10     TF <- 0.003
11     max_vecinos <- 2*tamano de solucion
12     max_exitos <- 0.1*max_vecinos
13
14     Mientras haya exitos o mientras que no se exploren mas de 15000
       soluciones
15     begin
16
17         vecinos generados <- 0
18         contador exitos <- 0
19
20         Mientras vecinos generados < max_vecinos y contador exitos <
           max_exitos
21         begin
22             s' <- S
23             s' <- flip(s')
24             s'_coste <- funcion_objetivo(training, test, s')
25
26             incremento vecinos generados
27             incremento soluciones totales
28
29             incremento_coste <- coste_s' - coste_S
30
31             Si incremento_coste > 0 o Unif(0,1) <= exp(-incremento_coste/T)
32             entonces
33                 S <- S'
34                 coste_S <- S'_coste
35
36                 Si S'_coste > coste_solucion
37                 entonces
38                     Solucion <- S'
39                     coste_solucion <- S'_coste
40                     incremento exitos
41         end
42
43     Actualizo T //Enfriamiento
```



```

44     end
45
46     devolver(Solucion, coste_solucion)
47 end

```

Unif(0,1) es un número aleatorio entre 0 y 1. max vecinos probé a iniciarlo con 10\*tamañosolucion, pero los tiempos eran extremadamente malos y lentos. Por lo tanto lo puse como 2\*tamañosolucion.

La temperatura para inicializarla sigo el procedimiento dado por el guión de prácticas, donde  $T_0 = \frac{\mu C(S_0)}{-\ln(\phi)}$  donde  $\mu = \phi = 0,3$ . Para el enfriamiento, he seguido el esquema también dado en el guión donde:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

siendo  $\beta = \frac{T_0 - T_f}{M * T_0 * T_f}$ , donde  $M = 15000 / \text{maxvecinos}$

Para el enfriamiento simulado he realizado una modificación sobre la función flip. Ahora en lugar de meter un vector de 'repetidos' meto el último índice que ha sido modificado, para no volver atrás sobre nuestros pasos.

## 5. Búsqueda tabú básica

El algoritmo en pseudocódigo es el siguiente:

```
1  busqueda_tabu(training, test)
2  begin
3      Solucion <- Generar_solucion_aleatoria
4      coste_solucion <- funcion_objetivo(training, test, Solucion)
5
6      lista_Tabu <- array de enteros
7
8      S <- Solucion
9      coste_S <- coste_Solucion
10
11     Mientras no hayamos superado el limite de soluciones generadas
12     begin
13         coste_mejor_vecino <- 0
14         mejor_vecino <- vacio
15
16         Para i desde 1 hasta 30
17         begin
18             vecino <- S
19             vecino <- flip(vecino)
20             coste_vecino <- funcion_objetivo(training, test, vecino)
21
22             Si el movimiento con el que hemos generado el vecino es valido
23             entonces
24                 si coste_vecino mejor que coste_mejor_vecino
25                 entonces
26                     mejor_vecino <- vecino
27                     coste_mejor_vecino <- coste_vecino
28
29                 Si la lista tabu esta llena
30                 entonces
31                     extraigo el movimiento mas antiguo e introduzco el nuevo
32                 si no
33                 entonces
34                     introduzco el nuevo movimiento
35
36
37                 Si coste_vecino es mejor que coste_solucion
38                 Solucion <- vecino
39                 coste_solucion <- coste_vecino
40
41             Si el movimiento no es valido
42             entonces
43                 Si coste_vecino es mejor que coste_solucion
44                 Solucion <- vecino
45                 coste_solucion <- coste_vecino
```

```

46     mejor_vecino <- vecino
47     coste_mejor_vecino <- coste_vecino
48
49     end
50
51     S <- mejor_vecino
52     coste_S <- coste_mejor_vecino
53
54     end
55
56     devolver Solucion y coste_solucion
57 end

```

En el número de soluciones máximo generadas probé con 15000 pero los tiempos eran muy lentos. Después fui probando y puse 250, que nos da unos tiempos razonables así como unas soluciones medio razonables.


La lista tabú se trata de un vector de enteros en los que guardo el índice de la componente del vector solución que hemos modificado con flip. El tamaño será de  $n/3$  (tamaño del vector solución entre 3). Para manejarla realizo lo siguiente:

```

1  movimiento <- flip(S) // S es modificada por referencia y flip
   //devuelve un entero
2
3  Para i desde 0 hasta (tamaño de lista tabu)-1
4  begin
5      Si lista_tabu[i] es igual que movimiento
6      entonces es no valido y salgo del bucle
7  end
8
9  Si es valido
10 entonces
11     .... //Acciones de actualizacion de vecino, etc.
12     Si la lista esta llena
13     entonces
14         Elimino el primer elemento
15         Introduzco al final el nuevo movimiento
16     si no
17     entonces
18         Introduzco al final el nuevo movimiento

```

## 6. Experimentos y análisis

de resultados/3NN.jpg  ../../../../../Escritorio/capturas de resultados/3NN.jpg