



ugr | Universidad
de **Granada**

Grado en Ingeniería Informática. Tercero.

Problema del Aprendizaje de Pesos en Características (APC): Parte 2.

Nombre de la asignatura:

Metaheurísticas. Grupo 2: Viernes de 17:30 a 19:30.

Realizado por:

Néstor Rodríguez Vico. DNI: 75573052C.

email: nrv23@correo.ugr.es



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN.**

Granada, 5 de junio de 2017.

Índice

1. Descripción del problema.	3
2. Descripción de la aplicación de los algoritmos empleados al problema.	4
3. Pseudocódigo de los algoritmos.	8
3.1. Enfriamiento simulado.	8
3.2. ILS.	10
3.3. Evolución Diferencial.	12
3.3.1. DE/Rand/1.	12
3.3.2. DE/current-to-best/1.	14
4. Descripción en pseudocódigo del algoritmo de comparación.	15
5. Proceso de desarrollo y manual de usuario.	15
6. Experimentos y análisis de resultados.	16
6.1. Sobreaprendizaje.	21
6.2. Estudio de la convergencia.	25
6.3. Posible mejora DE/current-to-best/1.	28
6.4. Estudio de la evolución de la temperatura del algoritmo Enfriamiento Simulado.	30
6.5. Características reducidas.	33
7. Bibliografía.	35

1. Descripción del problema.

El problema del Aprendizaje de Pesos en Características (APC) es un problema de búsqueda con codificación real en el espacio n -dimensional, para n características. Consiste obtener un vector de pesos que permita ponderar las características asociadas a un dato con la intención de obtener un mejor porcentaje de clasificación de datos futuros. Tenemos una muestra de datos $X = \{x_1, x_2, \dots, x_n\}$, cada dato del conjunto x_i está formado por un conjunto de características $\{c_1, c_2, \dots, c_m, CLASE\}$. El problema consiste en obtener un vector de pesos $W = \{w_1, w_2, \dots, w_m\}$ donde el peso w_i pondera la característica c_i .

Para obtener unos resultados más fiables en esta tarea vamos a usar la técnica de validación llamada *5-folds*, es decir, generaremos 5 particiones y aprenderemos con un conjunto formado por cuatro de ellas (conjunto al que me voy a referir como *train*) y validaremos con la partición restante (conjunto al que me voy a referir como *test*). Luego repetiremos el proceso pero cambiando la partición que dejamos como *test*, obteniendo así 5 resultados con distintos conjuntos de *train* y *test*.

Como clasificador se ha implementado el *kNN* (con $k=1$), el cual a cada dato le asigna la clase del vecino más cercano a dicho dato. El vecino más cercano será aquel con el cual se obtenga la menor *distancia euclídea* calculada de la siguiente forma:

$$d_e(e_1, e_2) = \sqrt{\sum_n^{i=1} (e_1^i - e_2^i)^2} \quad (1)$$

Nuestra función de evaluación se va a calcular en base a dos valores: el primero va a ser el rendimiento del clasificador *1-NN* usando el vector de pesos W calculado por cada algoritmo y el posterior cálculo del porcentaje asociado y el segundo va a ser la tasa de reducción del vector de pesos calculado por cada algoritmo. Consideraremos que un peso se ha reducido si es menor que 0.1 y, por lo tanto, la característica asociada a ese peso no es relevante. La tasa de clasificación se calcula como:

$$tasa_clas = 100 * \frac{n^o \text{ instancias bien clasificadas de } T}{n^o \text{ instancias en } T} \quad (2)$$

La tasa de reducción se calcula como:

$$tasa_reduc = 100 * \frac{n^o \text{ pesos menores que } 0,1}{n^o \text{ de pesos}} \quad (3)$$

La tasa de agregación se calcula como:

$$tasa_agregacion = \alpha * tasa_clas + (1 - \alpha) * tasa_reduc \quad \text{con } \alpha = 0,5 \quad (4)$$

Los distintos algoritmos se van a ejecutar sobre tres conjuntos de datos distintos; *sonar*, *spambase* y *wdbc*. Todo el proceso de normalización de los datos y partición de los conjuntos lo he realizado en *R*.¹

2. Descripción de la aplicación de los algoritmos empleados al problema.

Una solución para nuestro problema es un vector de n componentes, donde n es el número de características de cada dato de nuestro conjunto. En cada componente tendremos un valor real $w_i \in [0, 1]$:

- Un valor 1 en la componente w_i indica que la característica i se considera completamente en el cálculo de la distancia.
- Un valor menor que 0.1 en la componente w_i indica que la característica i no se considera en el cálculo de la distancia.
- Cualquier otro valor intermedio gradúa el peso asociado a cada característica y pondera su importancia.

De una manera gráfica tendríamos lo siguiente:

$$W = \begin{array}{|c|c|c|c|c|} \hline w_1 & w_2 & \dots & w_{n-1} & w_n \\ \hline \end{array}$$

$$x_i = \begin{array}{|c|c|c|c|c|} \hline c_1 & c_2 & \dots & c_{n-1} & c_n \\ \hline \end{array}$$

↓ ↓ ... ↓ ↓

En el caso de algoritmos que sólo traten con una solución usarán un vector como el descrito. En algoritmos que usen una población de soluciones, como son los algoritmos genéticos y meméticos, trataremos con un vector de vectores solución (lo que se podría ver como una matriz en la cual cada fila es una solución, es decir, un vector de pesos W).

Como función objetivo vamos medir el rendimiento del clasificador *1-NN* con los pesos calculados por cada algoritmo. La forma de proceder va a ser siempre la misma:

1. Ejecutamos el algoritmo para un conjunto de datos de entrenamiento, *train* y obtenemos un vector de pesos, W .
2. Llamamos al clasificador *1-NN* con el conjunto de prueba, *test*, y el vector de pesos obtenido por el algoritmo, W , el cual nos devuelve el conjunto de etiquetas, *clasificacion*, asociado a los datos del conjunto *test*.

¹He adjuntado el fichero .R, llamado *GenerarDatos.R* para que se puedan generar las particiones y los conjuntos de datos dentro de la carpeta *Data*, los cuales son leídos por mi algoritmos.

3. Calculamos la tasa de agregación en función de la tasa de clasificación y de la tasa de reducción del vector de pesos obtenido.

El pseudocódigo del clasificador *1-NN* es el siguiente:

```

1: function KNN(train, clases_train, conjunto_test, pesos)
2:   for i = 0, i < test.size(), i += 1 do
3:     vecino_mas_cercano = vecinoMasCercano(train, test[i], i, pesos)
4:     clasificacion[i] = clases_train[vecino_mas_cercano]
5:   end for
6: end function
7:   ▷ El argumento clasificacion es modificado y al finalizar el algoritmo contiene la
    clasificación calculada

```

La manera en la que se calcula el vecino más cercano es trivial. Se calcula en base a la fórmula de la *distancia Euclídea*. El vecino elegido será el que consiga una mejor distancia Euclídea con respecto al dato que estamos clasificando. El pseudocódigo de este proceso es el siguiente:

```

1: function VECINOMASCERCANO(datos, actual, posicion, pesos)
2:   mejor_distancia = 9999
3:   for i = 0, i < datos.size(), i += 1 do
4:     if i ≠ posicion then                                ▷ leave_one_out
5:       distancia_actual = distanciaEuclidea(datosi, actual)
6:       if distancia_actual < mejor_distancia then
7:         mejor_distancia = distancia_actual
8:         vecino_mas_cercano = i
9:       end if
10:      end if
11:    end for
12:    return clasificacion
13: end function

```

A la hora de calcular la distancia Euclídea debemos tener en cuenta que, en el cálculo de la distancia euclídea, no debemos contemplar los pesos que sean menor que 0.1. Esto se debe a que, si un peso es menor que 0.1, lo consideraremos 0. También debemos aplicar la técnica *leave-one-out*, la cual consiste en no tener en cuenta como posible vecino más cercano el propio dato que estamos procesando. Si no se aplicase el vecino más cercano de cada dato sería el mismo y por tanto la distancia sería 0. Basta con comprobar que los identificadores de los dos datos que estamos procesando sean distintos. Si pensamos sobre como se han generado los conjuntos de datos, vemos que un dato no puede estar en *train* y *test* simultáneamente, así que parece que no es necesario aplicar *leave-one-out*. Pero debemos tener en cuenta que nuestros algoritmos van a conocer el conjunto *test* en el proceso de cálculo del vector de pesos *W* y que cuando llamemos a nuestro clasificador *1-NN* dentro de los algoritmos lo le pasaremos el conjunto que recibe nuestro algoritmo,

train, como conjunto de entrenamiento y conjunto de prueba. Aquí es cuando surge la necesidad de aplicar *leave-one-out*.

Todo algoritmo parte de una solución inicial (o un conjunto de soluciones iniciales). Estas se van a generar de manera aleatoria, creando un vector W donde para cada componente se genera un número aleatorio en el rango $[0, 1]$. Sea cual sea el algoritmo debemos generar soluciones nuevas o modificar la que ya tenemos, por lo tanto debemos tener un operador de generación de vecino/mutación. Para ello vamos a modificar el gen i del cromosoma j siguiendo una distribución normal de media 0 y varianza σ^2 . Debemos tener en cuenta que esta distribución puede generar valores negativos, los cuales debemos truncar a 0 ya que, en nuestro problema, no tiene sentido un peso por debajo de 0. Para ello tenemos la siguiente función:

```

1: function TRUNCAR(numero)
2:   salida = numero
3:   if numero < 0 then
4:     salida = 0
5:   else if numero > 1 then
6:     salida = 1
7:   end if
8:   return salida
9: end function
```

Finalmente, como búsqueda local se ha implementado el siguiente algoritmo:

```

1: function LOCALSEARCH(train, clases_train, pesos, num_eval)
2:   indices = {1, 2, 3, ..., num_caracteristicas}
3:   pesos = solInicialAleatoria(num_caracteristicas)
4:   KNN(train, clases_train, train, clasificacion, pesos)
5:   porcentaje_ant = tasaAcierto(clases_train, clasificacion)
6:   i_aux = 0
7:   while num_eval < 15000 && num_vecinos < 20 * num_caracteristicas do
8:     mejora = false
9:     for i = 0, i < num_caracteristicas && !mejora, i += 1 do
10:       sol_nueva = pesos
11:       elegido = random ∈ [i_auxiliar, indices.size() - 1]
12:       posicion_auxiliar = indices[i_auxiliar]
13:       indices[i_auxiliar] = indices[elegido]
14:       indices[elegido] = posicion_auxiliar
15:       modificarPeso(sol_nueva, indices[i_auxiliar])
16:       num_vecinos += 1
17:       i_auxiliar += 1
18:       if i_auxiliar == num_caracteristicas then
19:         i_auxiliar = 0           ▷ Se han cambiado todas, empezamos de cero
20:       end if
```

```

21:      KNN(train, clases_train, train, clasificacion, sol_nueva)
22:      porcentaje_nuevo = tasaAggregacion(clases_train, clasificacion, sol_nueva,
0.5)
23:      num_eval += 1
24:      if porcentaje_nuevo > porcentaje_ant then
25:          pesos = sol_nueva
26:          porcentaje_ant = porcentaje_nuevo
27:          mejora = true
28:          num_vecinos = 0
29:      end if
30:  end for
31: end while
32: return pesos
33: end function

```

La búsqueda local implementada se trata de *búsqueda primero el mejor*. El vector *indices* nos indica en qué orden se van a modificar las componentes. En cada paso queremos modificar una componente aleatoria que no hayamos modificado antes, es decir, que esté desde la última que se modificó en *índices*, *i_auxiliar* y la última que se modificó, *indices.size() - 1*. De esta manera tenemos una forma aleatoria de modificar las componentes.

```

1: function SOLINICIALALEATORIA(size)
2:   for i = 0, i < size, i += 1 do
3:     solucion_inicial[i] = random ∈ [0, 1]
4:   end for
5:   return solucion_inicial
6: end function

```

```

1: function MODIFICARPESO(cromosoma, gen_mutar)
2:   cromosoma[gen_mutar] += random ∈ distribucion_normal(0, 0.09)
3:   numero = truncar(cromosoma[gen_mutar])
4:   cromosoma[gen_mutar] = numero
5: end function

```

```

1: function TASAAGREGACION(correctas, calculadas, solucion, alpha)
2:   porcentaje = tasaAcierto(correctas, calculadas)
3:   reduccion = tasaReduccion(solucion)
4:   return alpha*porcentaje+(1-alpha)*reduccion
5: end function

```

La función *tasaAggregacion* calcula como de buena es una solución en función de su tasa de acierto y de su tasa de reducción.

```

1: function TASAACIERTO(correctas, calculadas)
2:   for i = 0, i < num_caracteristicas, i += 1 do
3:     if correctas[i] == calculadas[i] then
4:       correctos += 1
5:     end if
6:   end for
7:   return (correctos * 1.0 / solucion.size())*100.0
8: end function

```

La función *tasaAcierto* simplemente calcula cuantas etiquetas calculadas por nuestro clasificador *1-NN* en función del vector de pesos, *W*, son correctas.

```

1: function TASAREDUCCION(solucion)
2:   for i = 0, i < solucion.size(), i += 1 do
3:     if solucion[i] < 0.1 then
4:       reducidos += 1
5:     end if
6:   end for
7:   return (reducidos * 1.0 / solucion.size())*100.0
8: end function

```

La función *tasaReducción* simplemente calcula cuantos pesos son menores que 0.1.

3. Pseudocódigo de los algoritmos.

Una aclaración para todos los algoritmos descritos a continuación: ciertos algoritmos no devuelven nada de forma explícita (usando *return*) sino que modifican los parámetros que reciben cuando son llamados. Esto está hecho por comodidad de no tener que declarar las variables que recogen las salidas de los algoritmos y porque ciertos algoritmos deben “devolver” varias cosas, como puede ser el vector de pesos calculado y el número de evaluaciones empleadas, siendo imposible hacer un *return* de varios objetos sin tener que crear una estructura auxiliar que los almacene.

3.1. Enfriamiento simulado.

```

1: function ENFRIAMIENTOSIMULADO(train, clases_train, pesos, num_eval)
2:   num_caracteristicas = train[0].size()
3:   max_vecinos = 10 * num_caracteristicas
4:   max_exitos = 0.1 * max_vecinos
5:   num_enfriamientos = 15000 / max_vecinos
6:   solucion = solInicialAleatoria(num_caracteristicas)
7:   KNN(train, clases_train, train, clasificacion, solucion)
8:   tasa_actual = tasaAcierto(clases_train, clasificacion)
9:   num_eval += 1
10:  mejor_solucion = solucion
11:  mejor_tasa = tasa_actual

```

```

12: t_ini = (0.3*(mejor_tasa/100.0))/(-log(0.3))
13: t_fin = 0.001
14: while t_fin > t_ini do
15:     t_fin = t_fin * 0.001
16: end while
17: beta = (t_ini - t_fin) / (num_enfriamientos*t_ini*t_fin)
18: t_actual = t_ini
19: while num_exitos > 0 && num_eval < 15000 && t_actual > t_fin do
20:     num_exitos = 0
21:     vecino = 0
22:     while num_exitos < max_exitos && vecinos < max_vecinos do
23:         sol_nueva = solucion
24:         gen = random ∈ [0, num_catacteristicas)
25:         mutarSolucion(sol_nueva, gen)
26:         num_eval += 1
27:         vecinos += 1
28:         KNN(train, clases_train, train, clasificacion, sol_nueva)
29:         tasa_nueva = tasaAgregacion(clases_train, clasificacion, sol_nueva, 0.5)
30:         mejora = tasa_nueva - tasa_actual
31:         if mejora/100.0 > 0 || random ∈ [0,1] < exp(-
            (mejora/100.0)/(t_actual/100)) then
32:             tasa_actual = tasa_nueva
33:             num_exitos += 1
34:             solucion = sol_nueva
35:             if tasa_nueva > mejor_tasa then
36:                 mejor_solucion = sol_nueva
37:                 mejor_tasa = tasa_actual
38:             end if
39:         end if
40:     end while
41:     t_actual = t_actual / (1 + beta * t_actual);
42: end while
43: return mejor_solucion
44: end function

```

La temperatura inicial se calcula como:

$$T_0 = \frac{\mu C(S_0)}{-\ln(\phi)} \quad (5)$$

donde T_0 es la temperatura inicial, $C(S_0)$ es el coste de la solución inicial y $\phi = 0.3$. La temperatura final, T_f , se fija a 10^{-3} . Como esquema de enfriamiento se usará el esquema de Cauchy modificado:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k} \quad \beta = \frac{T_0 - T_f}{MT_0 T_f} \quad (6)$$

donde M es el número de enfriamientos a realizar, T_0 es la temperatura inicial y T_f es la temperatura final.

3.2. ILS.

```

1: function ILS(train, clases_train, pesos, num_eval)
2:   num_caracteristicas = train[0].size()
3:   indices = {1, 2, 3, ..., num_caracteristicas}
4:   solucion = solInicialAleatoria(num_caracteristicas)
5:   KNN(train, clases_train, train, clasificacion, solucion)
6:   tasa_actual = tasaAcierto(clases_train, clasificacion)
7:   mejor_solucion = solucion
8:   tasa_mejor = tasa_actual
9:   num_mutaciones = 0.1 * num_caracteristicas
10:  localSearchILS(train, clases_train, solucion)
11:  for i = 0, i < 14, i += 1 do
12:    for i = 0, i < num_caracteristicas && !mejora, i += 1 do
13:      elegido = rand() % (indices.size() - i) + i
14:      posicion_auxiliar = indices[i]
15:      indices[i] = indices[elegido]
16:      indices[elegido] = posicion_auxiliar
17:    end for
18:    for i = 0, i < num_mutaciones && !mejora, i += 1 do
19:      mutarPosicion(solucion, indices[i])
20:    end for
21:    localSearchILS(train, clases_train, solucion)
22:    KNN(train, clases_train, train, clasificacion, solucion)
23:    tasa_actual = tasaAcierto(clases_train, clasificacion)
24:    if tasa_actual > tasa_mejor then
25:      mejor_solucion = solucion
26:      tasa_mejor = tasa_actual
27:    end if
28:    solucion = mejor_solucion
29:  end for
30:  return mejor_solucion
31: end function

```

En el caso de ILS no tenemos una condición de parada explícita como pasa en el resto de los algoritmos en los cuales se usa el número de evaluaciones. En ILS hacemos 15 búsquedas locales, cada una de ellas con 1000 evaluaciones. De esta manera conseguimos las 15000 evaluaciones del resto de los algoritmos.

```

1: function LOCALSEARCHILS(train, clases_train, pesos)
2:   indices = {1, 2, 3, ..., num_caracteristicas}
3:   pesos = solInicialAleatoria(num_caracteristicas)
4:   KNN(train, clases_train, train, clasificacion, pesos)
5:   porcentaje_ant = tasaAcierto(clases_train, clasificacion)
6:   i_aux = 0
7:   while num_eval < 1000 do
8:     mejora = false
9:     for i = 0, i < num_caracteristicas && !mejora, i += 1 do
10:       sol_nueva = pesos
11:       elegido = random ∈ [i_auxiliar, indices.size() - 1]
12:       posicion_auxiliar = indices[i_auxiliar]
13:       indices[i_auxiliar] = indices[elegido]
14:       indices[elegido] = posicion_auxiliar
15:       modificarPeso(sol_nueva, indices[i_auxiliar])
16:       num_vecinos += 1
17:       i_auxiliar += 1
18:       if i_auxiliar == num_caracteristicas then
19:         i_auxiliar = 0           ▷ Se han cambiado todas, empezamos de cero
20:       end if
21:       KNN(train, clases_train, train, clasificacion, sol_nueva)
22:       porcentaje_nuevo = tasaAgregacion(clases_train, clasificacion, sol_nueva,
0.5)
23:       num_eval += 1
24:       if porcentaje_nuevo > porcentaje_ant then
25:         pesos = sol_nueva
26:         porcentaje_ant = porcentaje_nuevo
27:         mejora = true
28:         num_vecinos = 0
29:       end if
30:     end for
31:   end while
32:   return pesos
33: end function

```

Para la búsqueda local empleada en ILS se usa el siguiente operador de mutación, el cual muta un gen en función de una distribución normal de media 0 y $\sigma^2 = 0,16$.

```

1: function MUTARPOSICION(solucion, posicion)
2:   solucion[posicion] += random ∈ distribucion_normal(0, 0.16)
3:   numero = truncar(solucion[posicion])
4:   solucion[posicion] = numero
5: end function

```

3.3. Evolución Diferencial.

Una par de aclaraciones:

- El proceso de recombinación empleado es común para los dos modelos:

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & \text{if } rand_j \in [0, 1] \leq crossover \text{ or } j = j_{rand} \\ x_{j,i,g} & \text{en otro caso} \end{cases}$$

- El proceso de reemplazamiento también es común, se va a hacer uno a uno, comparando el *padre_i* con el *hijo_{1i}*
- En cuanto al proceso de mutación es donde difieren los dos modelos.
 - DE/Rand/1 va a usar el siguiente proceso de mutación:

$$V_{i,G} = X_{r_1,G} + F (X_{r_2,G} - X_{r_1,G})$$

- DE/current-to-best/1 va a usar el siguiente proceso de mutación:

$$V_{i,G} = X_{i,G} + F (X_{best,G} - X_{i,G}) + F (X_{r_1,G} - X_{r_2,G})$$

3.3.1. DE/Rand/1.

```

1: function EVOLUCIONDIFERENCIALRAND(train, clases_train)
2:   crossover = 0.5
3:   indices = {1, 2, 3, ..., num_caracteristicas}
4:   poblacion = poblacionInicial(num_caracteristicas, 50)
5:   evalPoblacion(train, clases_train, poblacion, pcts_poblacion, num_eval)
6:   while num_eval < 15000 do
7:     for i = 0, i < poblacion.size(), i += 1 do
8:       indices_generados = 0
9:       i_aux = 0
10:      while indices_generados < 4 do
11:        elegido = random ∈ [i_aux, indices.size() - 1]
12:        posicion_auxiliar = indices[i_aux]
13:        indices[i_aux] = indices[elegido]
14:        indices[elegido] = posicion_auxiliar
15:        if indices[i_aux] != i then ▷ Si es distinto del que estamos procesando
16:          indices_generados += 1
17:          i_aux += 1
18:        end if
19:      end while
20:      padre1 = poblacion[indices[0]]
21:      padre2 = poblacion[indices[1]]
22:      padre3 = poblacion[indices[2]]
23:      gen_elegido = random ∈ [0, num_catacteristicas)
24:      for j = 0, j < num_caracteristicas, j += 1 do
25:        random = random ∈ [0, 1]
```

```
26:         if random < crossover o gen_elegido == j then
27:             numero = padre1[j] + 0.5 * (padre2[j] - padre3[j])
28:             numero = truncar(numero)
29:             hijo[j] = numero
30:         else
31:             hijo[j] = poblacion[i][j]
32:         end if
33:         hijos[i] = hijo
34:     end for
35: end for
36: evalPoblacion(train, clases_train, hijos, pcts_hijos, num_eval)
37: for i = 0, i < poblacion.size(), i += 1 do
38:     if pcts_poblacion[i] < pcts_hijos[i] then
39:         poblacion[i] = hijos[i]
40:         pcts_poblacion[i] = pcts_hijos[i]
41:     end if
42: end for
43: end while
44: max = 0
45: pos_max = 0
46: for i = 0, i < pcts_poblacion.size(), i += 1 do
47:     if pcts_poblacion[i] > max then
48:         max = pcts_poblacion[i]
49:         pos_max = i
50:     end if
51: end for
52: return poblacion[pos_max]
53: end function
```

3.3.2. DE/current-to-best/1.

```

1: function EVOLUCIONDIFERENCIALBEST(train, clases_train)
2:   crossover = 0.5
3:   indices = {1, 2, 3, ..., num_caracteristicas}
4:   index_mejores = {1, 2, 3, ..., num_caracteristicas}
5:   poblacion = poblacionInicial(num_caracteristicas, 50)
6:   evalPoblacion(train, clases_train, poblacion, pcts_poblacion, num_eval)
7:   sort(index_mejores segun pcts_poblacion) ▷ Mantenemos ordenada la poblacion
8:   best_padre = poblacion[index_mejores[0]]
9:   while num_eval < 15000 do
10:    for i = 0, i < poblacion.size(), i += 1 do
11:      indices_generados = 0
12:      i_aux = 0
13:      while indices_generados < 3 do
14:        elegido = random ∈ [i_aux, indices.size() - 1]
15:        posicion_auxiliar = indices[i_aux]
16:        indices[i_aux] = indices[elegido]
17:        indices[elegido] = posicion_auxiliar
18:        if indices[i_aux] != i then ▷ Si es distinto del que estamos procesando
19:          indices_generados += 1
20:          i_aux += 1
21:        end if
22:      end while
23:      padre1 = poblacion[indices[0]]
24:      padre2 = poblacion[indices[1]]
25:      gen_elegido = random ∈ [0, num_catacteristicas)
26:      for j = 0, j < num_caracteristicas, j += 1 do
27:        random = random ∈ [0, 1]
28:        if random < crossover o gen_elegido == j then
29:          numero = poblacion[i][j] + 0.5 * (best_padre[j] - poblacion[i][j]) +
0.5 * (padre1[j] - padre2[j])
30:          numero = truncar(numero)
31:          hijo[j] = numero
32:        else
33:          hijo[j] = poblacion[i][j]
34:        end if
35:        hijos[i] = hijo
36:      end for
37:    end for
38:    evalPoblacion(train, clases_train, hijos, pcts_hijos, num_eval)
39:    for i = 0, i < poblacion.size(), i += 1 do
40:      if pcts_poblacion[i] < pcts_hijos[i] then
41:        poblacion[i] = hijos[i]

```

```

42:           pcts_poblacion[i] = pcts_hijos[i]
43:       end if
44:   end for
45:   sort(index_mejores segun pcts_poblacion)      ▷ Mantenemos ordenada la
   poblacion
46:   best_padre = poblacion[index_mejores[0]]
47: end while
48: return poblacion[index_mejores[0]]
49: end function

```

4. Descripción en pseudocódigo del algoritmo de comparación.

-
- 1: pesos = algoritmo a comparar(train, clases_train)
 - 2: KNN(train, clases.train, test, clasificacion, pesos)
 - 3: porcentaje = tasaAgregacion(clases_test, clasificacion, pesos, 0.5)
-

La idea es sencilla, obtenemos los pesos usando un cierto algoritmo de los descritos anteriormente, llamamos al clasificador *1-NN* con los datos de *train* como conjunto de entrenamiento y *test* como conjunto de prueba y obtenemos la clasificación para los datos de *test*, *clasificación*. Finalmente, calculamos la tasa de agregación, comparando la clasificación obtenida por nuestros algoritmos con la clasificación real, *clases-test*, para obtener un porcentaje de acierto y calculando el número de pesos que han sido reducidos.

5. Proceso de desarrollo y manual de usuario.

Todo el código ha sido desarrollado por mí, usando como apoyo los seminarios y transparencias de teoría. Como páginas de consulta de Internet principalmente he usado cplusplus, sobre todo para la funciones de librerías externas, como puede ser para la generación de datos siguiendo una distribución normal. El proceso de desarrollo ha sido más fácil que en la primera práctica, ya que todo el proceso inicial ya estaba listo. Lo primero que he hecho ha sido adaptar la práctica anterior a esta, es decir, cambiar los algoritmos para que se adapten a la nueva forma de evaluación y generar las nuevas particiones de trabajo. Toda la práctica se ha desarrollado en C++ excepto el proceso de manipulación, normalización y partición de datos, el cual se ha hecho en R. Una vez hecho todo el proceso preliminar, implementé los algoritmos nuevos de esta práctica.

Para poder reproducir una ejecución de la práctica se ha proporcionado un *makefile*, el cual realiza tanto la creación de carpetas necesarias como la compilación. Una vez compilado, se ejecuta mediante el comando *./bin/main* (suponiendo que se use un entorno UNIX). Se ejecutarán todos los algoritmos para todas las particiones de todos los conjuntos de datos.

Los conjuntos de datos leídos por mis algoritmos se han aportado dentro de la carpeta `./Data/csv`. Igualmente se ha entregado el fichero `GenerarDatos.R`, el cual genera los conjuntos de datos.

6. Experimentos y análisis de resultados.

Los conjuntos de datos usados han sido los siguientes:

- **Sonar:** Conjunto de datos de detección de materiales mediante señales de sonar, discriminando entre objetos metálicos y rocas. 208 ejemplos con 60 características que deben ser clasificados en 2 clases.
- **Spambase:** Conjunto de datos de detección de SPAM frente a correo electrónico seguro. 460 ejemplos con 57 características que deben ser clasificados en 2 clases.
- **Wdbc (Wisconsin Database Breast Cancer):** Esta base de datos contiene 30 características calculadas a partir de una imagen digitalizada de una aspiración con aguja fina (FNA) de una masa en la mama. Se describen las características de los núcleos de las células presentes en la imagen. La tarea consiste en determinar si un tumor encontrado es benigno o maligno (M = maligno, B = benigna). 569 ejemplos con 30 características que deben ser clasificados en 2 clases.

Se debe tener en cuenta que para que la ejecución de nuestros algoritmos sea uniforme independientemente del conjunto de datos usado, estos mismos han sido modificados para tratarlos de manera uniforme. Las etiquetas han sido cambiadas por `-1` y `+1` y en todos los conjuntos se ha puesto la clase al final de cada ejemplo.

Varios algoritmos dependen números aleatorios para su ejecución, por eso se ha fijado como semilla mi DNI, `75573052`.

Una vez aclarado todo, veamos los resultados. Primero voy a mostrar un conjunto de tablas en las que se recogen los resultados obtenidos para todas las particiones de datos en los tres conjuntos de prueba. Veamos dichos resultados:

Sonar						wdbc						Spambase					
%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T		
1-NN																	
Set 1	90.4762	0	45.2381	0	94.7826	0	47.3913	0.002	86.0215	0	43.0108	0.003					
Set 2	88.0952	0	44.0476	0	93.913	0	46.9565	0.002	87.0968	0	43.5484	0.003					
Set 3	88.0952	0	44.0476	0	93.913	0	46.9565	0.003	78.4946	0	39.2473	0.002					
Set 4	87.8049	0	43.9024	0	95.6522	0	47.8261	0.002	87.0968	0	43.5484	0.002					
Set 5	87.8049	0	43.9024	0	95.614	0	47.807	0.002	89.2473	0	44.6237	0.002					
Media	88.4553	0	44.2276	0	94.775	0	47.3875	0.0022	85.5914	0	42.7957	0.0024					
Relief																	
Set 1	83.3333	15	49.1667	0.001	97.3913	3.33333	50.3623	0.01	87.0968	26.3158	56.7063	0.008					
Set 2	90.4762	11.6667	51.0714	0.001	93.913	3.33333	48.6232	0.006	88.172	22.807	55.4895	0.008					
Set 3	90.4762	11.6667	51.0714	0.002	95.6522	10	52.8261	0.006	84.9462	22.807	53.8766	0.008					
Set 4	80.4878	16.6667	48.5772	0.001	94.7826	0	47.3913	0.008	84.9462	22.807	53.8766	0.007					
Set 5	87.8049	10	48.9024	0.001	97.3684	0	48.6842	0.006	93.5484	22.807	58.1777	0.008					
Media	86.5157	13	49.7578	0.0012	95.8215	3.33333	49.5774	0.0072	87.7419	23.5088	55.6254	0.0078					
Enfriamiento Simulado - ES																	
Set 1	85.7143	21.6667	53.6905	3.407	95.6522	26.6667	61.1594	12.857	87.0968	15.7895	51.4431	15.677					
Set 2	88.0952	20	54.0476	3.482	91.3043	30	60.6522	12.473	81.7204	17.5439	49.6321	16.085					
Set 3	90.4762	21.6667	56.0714	3.291	93.913	26.6667	60.2899	13.052	76.3441	22.807	49.5756	15.038					
Set 4	80.4878	28.3333	54.4106	3.271	94.7826	30	62.3913	12.882	84.9462	15.7895	50.3679	15.886					
Set 5	87.8049	23.3333	55.5691	3.458	92.9825	26.6667	59.8246	12.762	81.7204	24.5614	53.1409	15.129					
Media	86.5157	23	54.7578	3.3818	93.7269	28	60.8635	12.8052	82.3656	19.2982	50.8319	15.563					

	Sonar				wdbc				Spambase			
	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T
Iterative Local Search - ILS												
Set 1	85.7143	13.3333	49.5238	34.41	95.6522	23.3333	59.4928	125.997	84.9462	19.2982	52.1222	156.357
Set 2	92.8571	13.3333	53.0952	34.136	93.913	10	51.9565	134.404	91.3979	14.0351	52.7165	159.861
Set 3	88.0952	26.6667	57.381	32.012	96.5217	26.6667	61.5942	125.821	75.2688	15.7895	45.5291	156.919
Set 4	85.3659	11.6667	48.5163	34.921	96.5217	10	53.2609	134.91	77.4194	15.7895	46.6044	158.461
Set 5	90.2439	15	52.622	34.312	95.614	10	52.807	133.286	80.6452	19.2982	49.9717	153.8
Media	88.4553	16	52.2276	33.9582	95.6445	16	55.8223	130.9	81.9355	16.8421	49.3888	157.08
Differential Evolution - DE/rand/1												
Set 1	78.5714	90	84.2857	26.51	85.2174	96.6667	90.942	111.932	78.4946	94.7368	86.6157	128.736
Set 2	78.5714	88.3333	83.4524	26.097	86.087	93.3333	89.7101	106.425	83.871	92.9825	88.4267	127.037
Set 3	76.1905	90	83.0952	26.055	91.3043	93.3333	92.3188	116.936	87.0968	91.2281	89.1624	127.394
Set 4	75.6098	90	82.8049	26.506	93.0435	93.3333	93.1884	110.226	83.871	91.2281	87.5495	125.289
Set 5	85.3659	86.6667	86.0163	26.559	88.5965	93.3333	90.9649	103.186	88.172	92.9825	90.5772	128.649
Media	78.8618	89	83.9309	26.3454	88.8497	94	91.4249	109.741	84.3011	92.6316	88.4663	127.421
Differential Evolution - DE/current-to-best/1												
Set 1	85.7143	61.6667	73.6905	26.235	94.7826	76.6667	85.7246	104.004	83.871	59.6491	71.76	123.311
Set 2	85.7143	55	70.3571	26.915	91.3043	86.6667	88.9855	103.904	86.0215	63.1579	74.5897	127.963
Set 3	88.0952	56.6667	72.381	26.819	91.3043	86.6667	88.9855	96.566	86.0215	64.9123	75.4669	121.651
Set 4	73.1707	68.3333	70.752	25.751	96.5217	83.3333	89.9275	106.421	79.5699	63.1579	71.3639	122.42
Set 5	92.6829	55	73.8415	27.349	92.1053	76.6667	84.386	108.073	90.3226	66.6667	78.4946	119.948
Media	85.0755	59.3333	72.2044	26.6138	93.2037	82	87.6018	103.794	85.1613	63.5088	74.335	123.059

Sonar							wdbc							Spambase						
%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	
Local Search - LS																				
Set 1	78.5714	10	44.2857	3.95	96.5217	13.3333	54.9275	10.12	84.9462	12.2807	48.6135	14.994								
Set 2	88.0952	11.6667	49.881	5.083	92.1739	10	51.087	11.132	83.871	15.7895	49.8302	30.548								
Set 3	83.3333	6.66667	45	4.708	95.6522	6.66667	51.1594	5.491	79.5699	12.2807	45.9253	16.287								
Set 4	80.4878	3.33333	41.9106	6.843	96.5217	3.33333	49.9275	6.625	83.871	8.77193	46.3214	24.259								
Set 5	92.6829	5	48.8415	2.821	95.614	3.33333	49.4737	5.776	86.0215	14.0351	50.0283	17.201								
Media	84.6341	7.33333	45.9837	4.681	95.2967	7.33333	51.315	7.8288	83.6559	12.6316	48.1437	20.6578								
Algoritmo Genético Generacional - AGG-BLX																				
Set 1	83.3333	30	56.6667	30.604	97.3913	33.3333	65.3623	118.387	86.0215	40.3509	63.1862	137.289								
Set 2	90.4762	38.3333	64.4048	29.166	93.913	33.3333	63.6232	116.357	82.7957	38.5965	60.6961	135.754								
Set 3	85.7143	30	57.8571	30.216	95.6522	40	67.8261	115.016	80.6452	28.0702	54.3577	144.698								
Set 4	85.3659	33.3333	59.3496	30.124	97.3913	43.3333	70.3623	112.328	82.7957	40.3509	61.5733	136.894								
Set 5	82.9268	35	58.9634	29.905	94.7368	50	72.3684	112.085	89.2473	35.0877	62.1675	138.649								
Media	85.5633	33.3333	59.4483	30.003	95.8169	40	67.9085	114.835	84.3011	36.4912	60.3962	138.657								
Algoritmo Memético - AM10-0.1mej																				
Set 1	83.3333	33.3333	58.3333	29.707	96.5217	43.3333	69.9275	113.648	87.0968	35.0877	61.0922	140.307								
Set 2	85.7143	35	60.3571	29.708	95.6522	56.6667	76.1594	113.159	82.7957	28.0702	55.4329	143.185								
Set 3	88.0952	40	64.0476	29.073	94.7826	36.6667	65.7246	117.761	78.4946	38.5965	58.5456	136.842								
Set 4	82.9268	41.6667	62.2967	29.631	94.7826	40	67.3913	114.706	83.871	42.1053	62.9881	133.242								
Set 5	85.3659	38.3333	61.8496	30.126	93.8596	46.6667	70.2632	111.957	87.0968	28.0702	57.5835	145.946								
Media	85.0871	37.6667	61.3769	29.649	95.1198	44.6667	69.8932	114.246	83.871	34.386	59.1285	139.904								

	%_clas	%_red	Agreg	T
Sonar				
1-NN	88.4553	0	44.2276	0
Relief	86.5157	13	49.7578	0.0012
ES	86.5157	23	54.7578	3.3818
ILS	88.4553	16	52.2276	33.9582
DE/Rand/1	78.8618	89	83.9309	26.3454
DE/current-to-best/1	85.0755	59.3333	72.2044	26.6138
LS	84.6341	7.33333	45.9837	4.681
AGG-BLX	85.5633	33.3333	59.4483	30.003
AM10-0.1mej	85.0871	37.6667	61.3769	29.649
Media	85.4626	30.9629	58.2127	17.1814
wdbc				
1-NN	94.775	0	47.3875	0.0022
Relief	95.8215	3.33333	49.5774	0.0072
ES	93.7269	28	60.8635	12.8052
ILS	95.6445	16	55.8223	130.9
DE/Rand/1	88.8497	94	91.4249	109.741
DE/current-to-best/1	93.2037	82	87.6018	103.794
LS	95.2967	7.33333	51.315	7.8288
AGG-BLX	95.8169	40	67.9085	114.835
AM10-0.1mej	95.1198	44.6667	69.8932	114.246
Media	94.25052	35.03704	64.64379	66.01771
spambase				
1-NN	85.5914	0	42.7957	0.0024
Relief	87.7419	23.5088	55.6254	0.0078
ES	82.3656	19.2982	50.8319	15.563
ILS	81.9355	16.8421	49.3888	157.08
DE/Rand/1	84.3011	92.6316	88.4663	127.421
DE/current-to-best/1	85.1613	63.5088	74.335	123.059
LS	83.6559	12.6316	48.1437	20.6578
AGG-BLX	84.3011	36.4912	60.3962	138.657
AM10-0.1mej	83.871	34.386	59.1285	139.904
Media	84.32498	33.25537	58.79017	80.26133

Tabla 6.1: Comparativa de todos los algoritmos.

Una vez recogidos los datos, pasemos al análisis de los mismos. Lo primero que quiero comentar es que los tiempos recogidos han sido obtenidos aplicando optimización en tiempo de compilación, en concreto el nivel *-O2* ofrecido por el compilador *g++*. Como dato curioso, la ejecución completa del programa (la cual incluye la ejecución de todos los algoritmos para todos los conjuntos de datos para todas las bases de datos, la carga de los datos cada vez que son necesitados y todo el proceso de comparación

de soluciones y evaluación del vector de pesos obtenido) ha tardado *7362.1 segundos*, es decir, *dos horas y dos minutos* aproximadamente.

Analizando los resultados recogidos en la tabla 6.1 podemos ver que los mejores resultados para los tres conjuntos de datos se obtienen usando algoritmos basados en poblaciones. El mejor resultado lo obtiene el algoritmo de *Evolución Diferencial* con cruce y mutación aleatoria. En la parte extra voy a hacer una pequeña prueba para ver si se puede mejorar el algoritmo de *Evolución Diferencial current to best* para obtener unos mejores resultados.

A diferencia de la primera práctica, con la nueva función objetivo, podemos ver más diferencias entre los distintos algoritmos implementados. Podemos ver que los algoritmos de la primera práctica se diferencian en cuanto a resultados cuando eso antes no pasaba y que los algoritmos de Evolución Diferencial implementados en esta práctica son los que mejores resultados obtienen.

Si nos ceñimos a los datos recogidos en las tablas no hay mucho más que explicar que no se pueda ver claramente, así que pasemos a estudiar algunos aspectos que he visto con esta práctica.

6.1. Sobreaprendizaje.

El *sobreaprendizaje* es el efecto de sobreentrenar un algoritmo. Este efecto puede provocar que el algoritmo sea muy bueno para los datos de entrenamiento pero que luego con los datos de prueba no funcione tan bien, es decir, el algoritmo se especializa en los datos de entrenamiento y pierde la generalidad necesaria para poder realizar un buen trabajo fuera del conjunto de entrenamiento.

Para ver que algoritmos sufren de este problema simplemente debemos realizar lo siguiente:

- Calculamos el vector de pesos W usando el conjunto de entrenamiento, *train*.
- Llamamos al clasificador *1-NN* pasándole como conjunto de entrenamiento y prueba el mismo, *train*. De esta manera valoramos como se comporta W para clasificar los mismos datos con los que ha aprendido.
- Calculamos la tasa de agregación para compararlo posteriormente.
- Llamamos de nuevo al clasificador *1-NN* pasándole como conjunto de entrenamiento *train* y como conjunto de prueba *test*. De esta manera valoramos como se comporta W para clasificar unos datos nuevos
- Calculamos la tasa de agregación para compararlo posteriormente.
- Comparamos los dos porcentajes.

En la práctica anterior el sobreaprendizaje se veía reflejado en el porcentaje de clasificación. En esta práctica ocurre igual pero esto repercute en la tasa de agregación (la tasa de reducción no se ve afectada ya que depende del vector solución obtenido, no del conjunto de datos usado como *test*). En las tablas que voy a mostrar voy a representar el porcentaje de clasificación y la tasa de agregación, para tener una visión más general de lo ocurrido. Los resultados los podemos ver a continuación. Se ha creado una tabla por algoritmo.

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	84.3373	53.002	95.614	61.1404	82.3848	49.0872
1. Train-Test	85.7143	53.6905	95.6522	61.1594	87.0968	51.4431
2. Train-Train	86.1446	53.0723	96.2719	63.136	83.7398	50.6418
2. Train-Test	88.0952	54.0476	91.3043	60.6522	81.7204	49.6321
3. Train-Train	85.5422	53.6044	95.3947	61.0307	84.2818	53.5444
3. Train-Test	90.4762	56.0714	93.913	60.2899	76.3441	49.5756
4. Train-Train	86.2275	57.2804	94.9561	62.4781	85.3659	50.5777
4. Train-Test	80.4878	54.4106	94.7826	62.3913	84.9462	50.3679
5. Train-Train	82.6347	52.984	96.0613	61.364	83.1978	53.8796
5. Train-Test	87.8049	55.5691	92.9825	59.8246	81.7204	53.1409

Tabla 6.2: Sobreaprendizaje: ES

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	93.3735	53.3534	96.4912	59.9123	91.3279	55.3131
1. Train-Test	85.7143	49.5238	95.6522	59.4928	84.9462	52.1222
2. Train-Train	92.7711	53.0522	96.9298	53.4649	92.6829	53.359
2. Train-Test	92.8571	53.0952	93.913	51.9565	91.3979	52.7165
3. Train-Train	87.9518	57.3092	95.614	61.1404	91.3279	53.5587
3. Train-Test	88.0952	57.381	96.5217	61.5942	75.2688	45.5291
4. Train-Train	87.4251	49.5459	97.3684	53.6842	91.3279	53.5587
4. Train-Test	85.3659	48.5163	96.5217	53.2609	77.4194	46.6044
5. Train-Train	93.4132	54.2066	95.6236	52.8118	88.8889	54.0936
5. Train-Test	90.2439	52.622	95.614	52.807	80.6452	49.9717

Tabla 6.3: Sobreaprendizaje: ILS

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	94.5783	92.2892	90.3509	93.5088	87.8049	91.2709
1. Train-Test	78.5714	84.2857	85.2174	90.942	78.4946	86.6157
2. Train-Train	93.9759	91.1546	94.0789	93.7061	91.8699	92.4262
2. Train-Test	78.5714	83.4524	86.087	89.7101	83.871	88.4267
3. Train-Train	90.3614	90.1807	94.2982	93.8158	93.2249	92.2265
3. Train-Test	76.1905	83.0952	91.3043	92.3188	87.0968	89.1624
4. Train-Train	93.4132	91.7066	94.7368	94.0351	91.0569	91.1425
4. Train-Test	75.6098	82.8049	93.0435	93.1884	83.871	87.5495
5. Train-Train	93.4132	90.0399	93.8731	93.6032	90.7859	91.8842
5. Train-Test	85.3659	86.0163	88.5965	90.9649	88.172	90.5772

Tabla 6.4: Sobreaprendizaje: DE/rand/1

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	93.3735	77.5201	97.5877	87.1272	91.0569	75.353
1. Train-Test	85.7143	73.6905	94.7826	85.7246	83.871	71.76
2. Train-Train	92.7711	73.8855	93.4211	90.0439	88.3469	75.7524
2. Train-Test	85.7143	70.3571	91.3043	88.9855	86.0215	74.5897
3. Train-Train	92.1687	74.4177	94.7368	90.7018	92.9539	78.9331
3. Train-Test	88.0952	72.381	91.3043	88.9855	86.0215	75.4669
4. Train-Train	94.012	81.1727	96.7105	90.0219	90.2439	76.7009
4. Train-Test	73.1707	70.752	96.5217	89.9275	79.5699	71.3639
5. Train-Train	91.018	73.009	94.7484	85.7075	90.7859	78.7263
5. Train-Test	92.6829	73.8415	92.1053	84.386	90.3226	78.4946

Tabla 6.5: Sobreaprendizaje: DE/current-to-best/1

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	85.5422	47.7711	96.0526	54.693	86.9919	49.6363
1. Train-Test	78.5714	44.2857	96.5217	54.9275	84.9462	48.6135
2. Train-Train	89.1566	50.4116	96.4912	53.2456	89.4309	52.6102
2. Train-Test	88.0952	49.881	92.1739	51.087	83.871	49.8302
3. Train-Train	89.1566	47.9116	95.614	51.1404	89.1599	50.7203
3. Train-Test	83.3333	45	95.6522	51.1594	79.5699	45.9253
4. Train-Train	88.0239	45.6786	96.4912	49.9123	88.8889	48.8304
4. Train-Test	80.4878	41.9106	96.5217	49.9275	83.871	46.3214
5. Train-Train	83.2335	44.1168	96.4989	49.9161	87.8049	50.92
5. Train-Test	92.6829	48.8415	95.614	49.4737	86.0215	50.0283

Tabla 6.6: Sobreaprendizaje: LS

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	89.759	59.8795	97.1491	65.2412	87.8049	64.0779
1. Train-Test	83.3333	56.6667	97.3913	65.3623	86.0215	63.1862
2. Train-Train	87.9518	63.1426	96.0526	64.693	89.1599	63.8782
2. Train-Test	90.4762	64.4048	93.913	63.6232	82.7957	60.6961
3. Train-Train	89.759	59.8795	97.5877	68.7939	90.7859	59.428
3. Train-Test	85.7143	57.8571	95.6522	67.8261	80.6452	54.3577
4. Train-Train	87.4251	60.3792	96.7105	70.0219	89.4309	64.8909
4. Train-Test	85.3659	59.3496	97.3913	70.3623	82.7957	61.5733
5. Train-Train	92.8144	63.9072	97.1554	73.5777	89.4309	62.2593
5. Train-Test	82.9268	58.9634	94.7368	72.3684	89.2473	62.1675

Tabla 6.7: Sobreaprendizaje: AGG-BLX

	sonar		wdbc		spambase	
	%_clas	Agreg	%_clas	Agreg	%_clas	Agreg
1. Train-Train	92.7711	63.0522	97.1491	70.2412	87.8049	61.4463
1. Train-Test	83.3333	58.3333	96.5217	69.9275	87.0968	61.0922
2. Train-Train	89.759	62.3795	98.0263	77.3465	91.0569	59.5635
2. Train-Test	85.7143	60.3571	95.6522	76.1594	82.7957	55.4329
3. Train-Train	89.1566	64.5783	97.1491	66.9079	89.1599	63.8782
3. Train-Test	88.0952	64.0476	94.7826	65.7246	78.4946	58.5456
4. Train-Train	92.2156	66.9411	96.7105	68.3553	89.7019	65.9036
4. Train-Test	82.9268	62.2967	94.7826	67.3913	83.871	62.9881
5. Train-Train	91.018	64.6757	96.7177	71.6922	90.2439	59.157
5. Train-Test	85.3659	61.8496	93.8596	70.2632	87.0968	57.5835

Tabla 6.8: Sobreaprendizaje: AM10-0.1mej

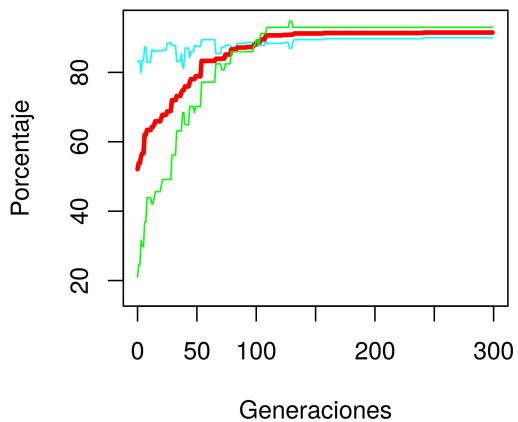
En algunos de los conjuntos las tasas de clasificación y por lo tanto de agregación para los casos en los que se aprende y se prueba con el mismo conjunto (filas de las tablas etiquetadas como *Train-Train*) tienen un valor bastante más alto que las filas en las que el conjunto de entrenamiento es distinto al conjunto de test. Esto se debe al *sobreaprendizaje*. Como ya he comentado anteriormente los algoritmos aprenden más de la cuenta y por eso obtienen unas tasas de acierto alta para conjuntos iguales pero cuando el conjunto de *test* cambia, el porcentaje decae, ya que el algoritmo no ha encontrado una solución lo suficientemente genérica capaz de clasificar correctamente los nuevos datos de *test*. Aún así podemos ver que no sucede como en la práctica anterior. Con esta nueva función objetivo en algunas particiones se da el caso de que los algoritmos no sobreaprenden y obtienen un mejor porcentaje de clasificación y por lo tanto de agregación en el conjunto de *test*.

6.2. Estudio de la convergencia.

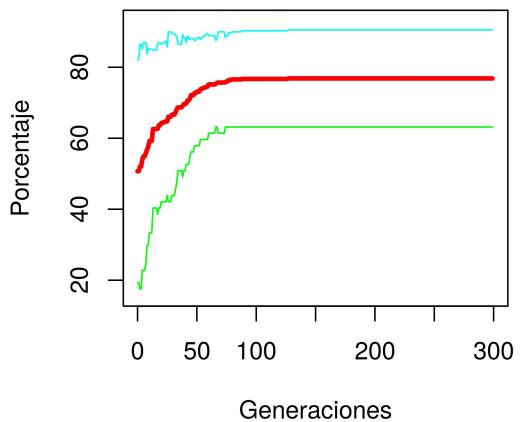
Vamos a analizar como convergen la tasa de agregación en función de la tasa de clasificación y la tasa de reducción. Para ello voy a modificar mis algoritmos para que escriban en un fichero las tres tasas de la mejor solución encontrada en cada generación y así podremos ver como se comportan con el paso de las generaciones. Para no afectar al funcionamiento de la práctica voy a realizar dichas modificaciones en un proyecto nuevo, dentro de la carpeta APC_2_analisis. Dentro de dicha carpeta están todos los datos obtenidos, un fichero llamado *AnalisisDatos.R*, el cual se encarga de analizar los datos obtenidos y pintar las gráficas necesarias para este análisis ². Voy a realizar el estudio sobre el conjunto de datos *spambase*. Los dos algoritmos que voy a tratar son los de Evolución Diferencial, ya que son los que más interesantes me parecen. Las gráficas obtenidas para el algoritmo de *Evolución Diferencial current to best* son las siguientes:

²Dicha carpeta no se ha entregado ya que no es requerido, en caso de querer verla, mándeme un correo y se lo envío.

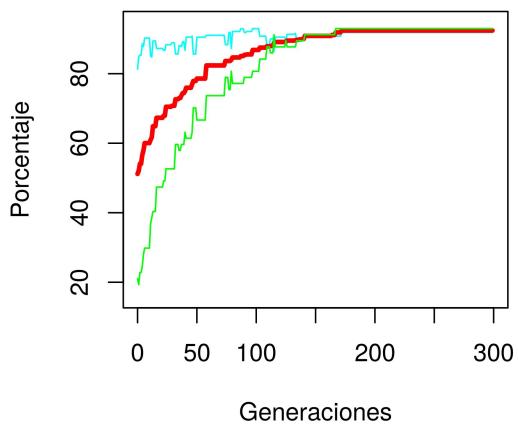
DE/Rand/1: Partición 1.



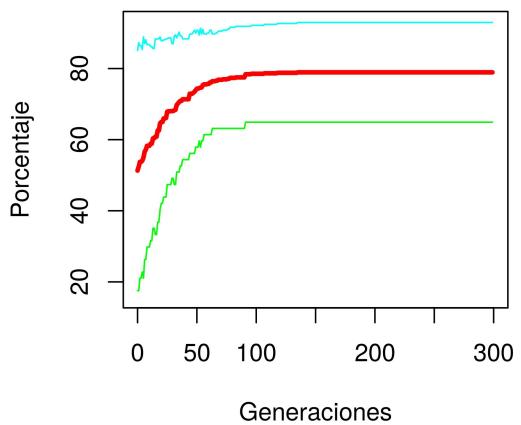
DE/current-to-best/1: Partición 1.



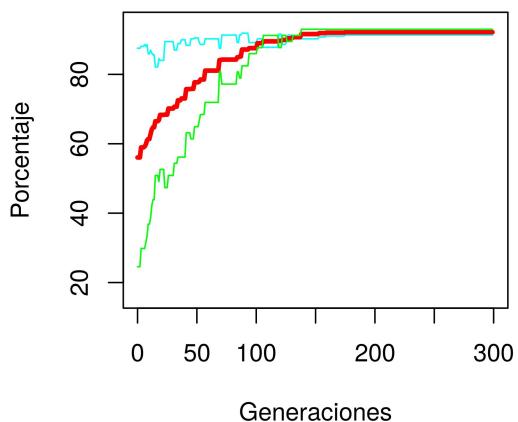
DE/Rand/1: Partición 2.



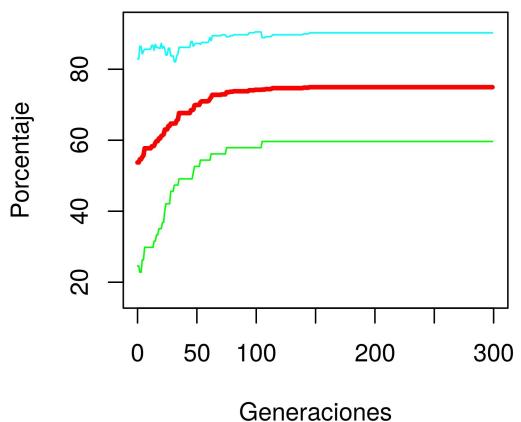
DE/current-to-best/1: Partición 2.



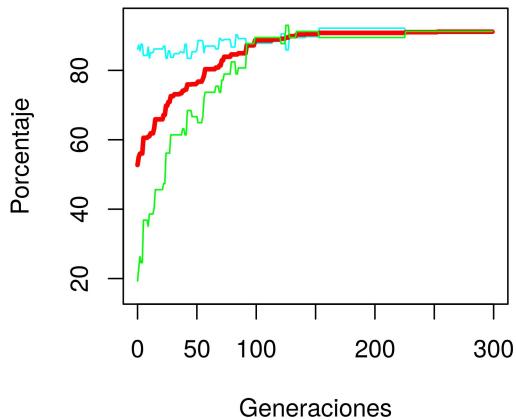
DE/Rand/1: Partición 3.



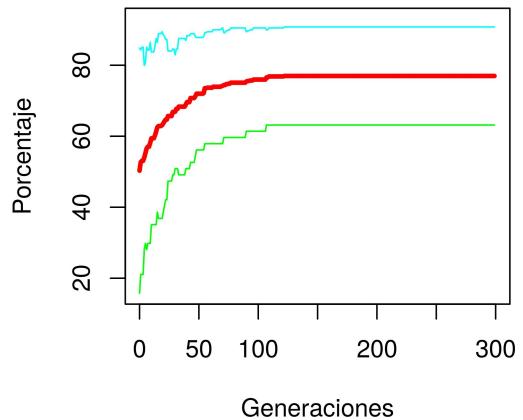
DE/current-to-best/1: Partición 3.



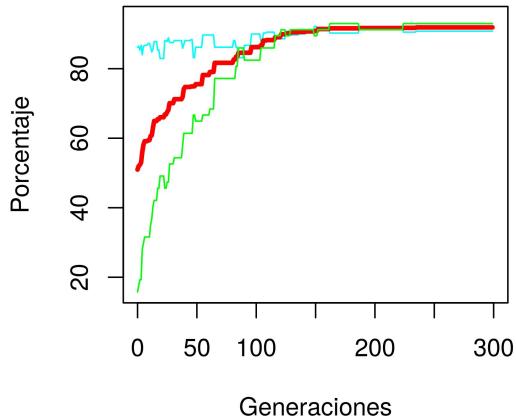
DE/Rand/1: Partición 4.



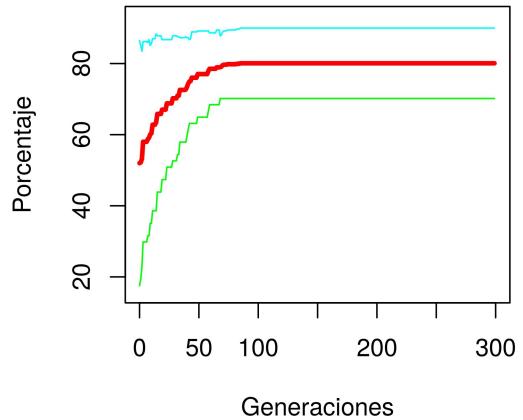
DE/current-to-best/1: Partición 4.



DE/Rand/1: Partición 5.



DE/current-to-best/1: Partición 5.



En cada gráfica se ha pintado la tasa de clasificación en color azul, la tasa de reducción en color verde y nuestra función objetivo, la tasa de reducción, en color rojo de la mejor solución de cada generación para las 5 particiones del conjunto de datos *spambase*.

Al igual que sucedía con la primera práctica los algoritmos convergen bastante rápido y se desaprovechan iteraciones. También podemos ver que las gráficas de cada algoritmo se parecen entre ellas pero que hay bastante diferencia entre las gráficas del algoritmo con cruce aleatorio frente al algoritmo con cruce por el mejor parent. Lo que sucede es que *DE/Rand/1* va avanzando muy poco a poco, ya que va cogiendo padres aleatorios y por eso tarda más en converger.

También hemos visto que con la versión aleatoria obtenemos mejores resultados, así que voy a intentar mejorar la versión *current-to-best*.

6.3. Possible mejora DE/current-to-best/1.

Esta versión del algoritmo *Evolución Diferencial* usa al mejor individuo de la población como referente para generar los descendientes de la población pero, ¿que pasa si ese mejor individuo no es lo suficientemente bueno? Lo que voy a hacer es aplicar un algoritmo de búsqueda local corta (la misma que se aplica en el algoritmo memético) para mejorar el primer mejor individuo que se elige y ver si así obtenemos unos mejores resultados. El nuevo pseudocódigo de esta versión es casi idéntico al planteado:

```
1: function EVOLUCIONDIFERENCIALBEST(train, clases_train)
2:   crossover = 0.5
3:   indices = {1, 2, 3, ..., num_caracteristicas}
4:   index_mejores = {1, 2, 3, ..., num_caracteristicas}
5:   poblacion = poblacionInicial(num_caracteristicas, 50)
6:   evalPoblacion(train, clases_train, poblacion, pcts_poblacion, num_eval)
7:   sort(index_mejores segun pcts_poblacion) ▷ Mantenemos ordenada la poblacion
8:   padreParaMejorar = poblacion[index_mejores[0]]
9:   padreMejorado = localSearch(train, clases_train, padreParaMejorar)
10:  best_padre = padreMejorado
11:  while num_eval < 15000 do
12:    ...
13:    end while
14: end function
```

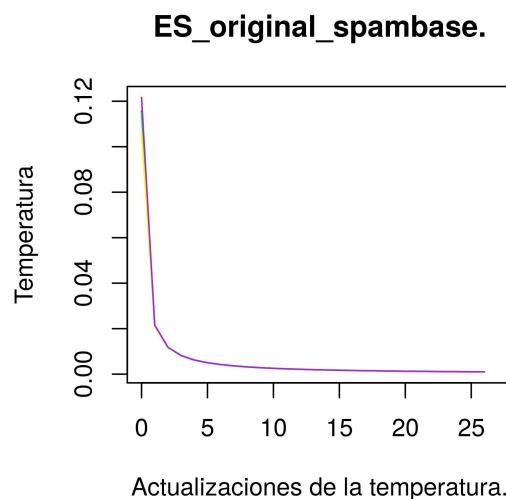
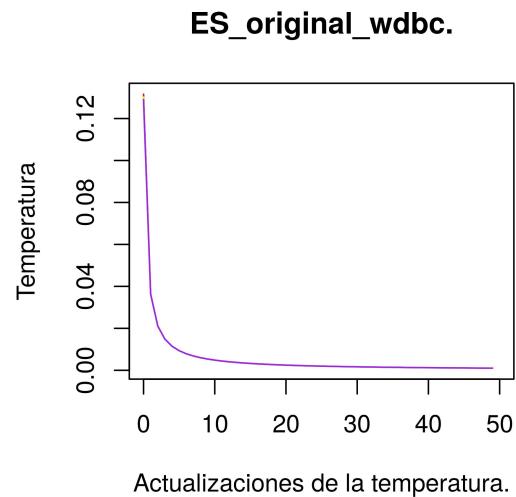
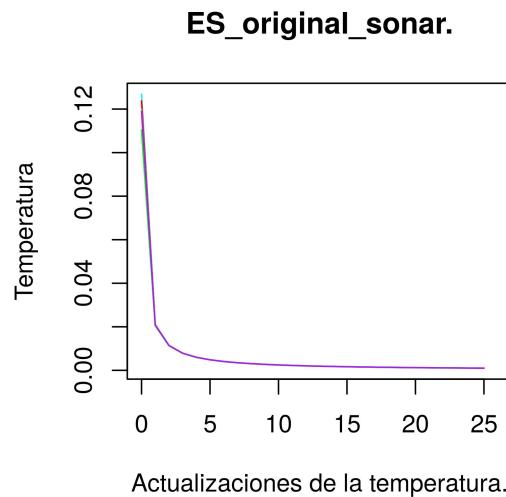
Veamos los resultados. Voy a comparar esta versión con la versión original y con la versión aleatoria:

Sonar							Spambase						
%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T		
Differential Evolution - DE/rand/1													
Set 1	78.5714	90	84.2857	26.51	85.2174	96.6667	90.942	111.932	78.4946	94.7368	86.6157	128.736	
Set 2	78.5714	88.3333	83.4524	26.097	86.087	93.3333	89.7101	106.425	83.871	92.9825	88.4267	127.037	
Set 3	76.1905	90	83.0952	26.055	91.3043	93.3333	92.3188	116.936	87.0968	91.2281	89.1624	127.394	
Set 4	75.6098	90	82.8049	26.506	93.0435	93.3333	93.1884	110.226	83.871	91.2281	87.5495	125.289	
Set 5	85.3659	86.6667	86.0163	26.559	88.5965	93.3333	90.9649	103.186	88.172	92.9825	90.5772	128.649	
Media	78.8618	89	83.9309	26.3454	88.8497	94	91.4249	109.741	84.3011	92.6316	88.4663	127.421	
Differential Evolution - DE/current-to-best/1													
Set 1	85.7143	61.6667	73.6905	26.235	94.7826	76.6667	85.7246	104.004	83.871	59.6491	71.76	123.311	
Set 2	85.7143	55	70.3571	26.915	91.3043	86.6667	88.9855	103.904	86.0215	63.1579	74.5897	127.963	
Set 3	88.0952	56.6667	72.381	26.819	91.3043	86.6667	88.9855	96.566	86.0215	64.9123	75.4669	121.651	
Set 4	73.1707	68.3333	70.752	25.751	96.5217	83.3333	89.9275	106.421	79.5699	63.1579	71.3639	122.42	
Set 5	92.6829	55	73.8415	27.349	92.1053	76.6667	84.386	108.073	90.3226	66.6667	78.4946	119.948	
Media	85.0755	59.3333	72.2044	26.6138	93.2037	82	87.6018	103.794	85.1613	63.5088	74.335	123.059	
Differential Evolution - DE/current-to-best/1 con LS													
Set 1	88.0952	53.3333	70.7143	39.726	95.6522	53.3333	74.4928	142.686	88.172	57.8947	73.0334	166.481	
Set 2	90.4762	60	75.2381	35.368	92.1739	76.6667	84.4203	138.586	77.4194	66.6667	72.043	165.038	
Set 3	90.4762	65	77.7381	34.406	96.5217	70	83.2609	137.488	83.871	68.4211	76.146	164.997	
Set 4	73.1707	63.3333	68.252	38.724	93.0435	63.3333	78.1884	136.968	83.871	61.4035	72.6372	163.259	
Set 5	85.3659	56.6667	71.0163	36.964	94.7368	73.3333	84.0351	137.06	89.2473	63.1579	76.2026	164.011	
Media	85.5168	59.6667	72.5918	37.0376	94.4256	67.3333	80.8795	138.558	84.5161	63.5088	74.0125	164.757	

Podemos ver que, a pesar de coger un padre mejor para arrancar la búsqueda, no obtenemos unos resultados mejores que la versión original y, mucho menos, unos resultados mejores que la versión aleatoria.

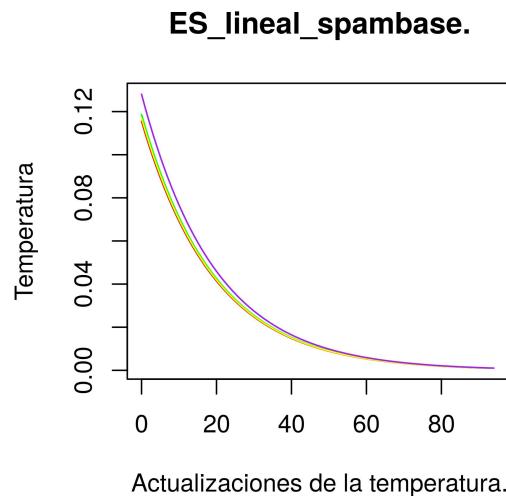
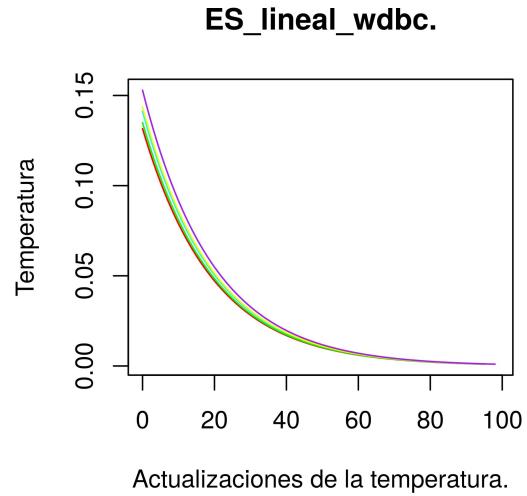
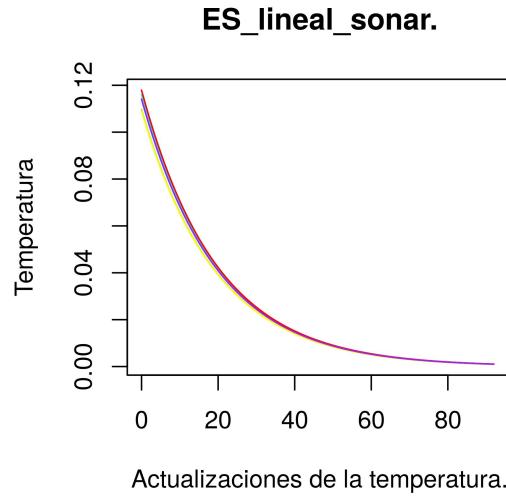
6.4. Estudio de la evolución de la temperatura del algoritmo Enfriamiento Simulado.

A continuación voy a realizar un pequeño estudio de como va evolucionando la temperatura con el paso de las iteraciones. Veamos la siguiente gráfica:



Como podemos ver, la temperatura decae muy rápidamente, lo cual hace que sea más difícil aceptar una solución peor que la mejor solución encontrada.

En vez de usar como esquema de enfriamiento el de Cauchy voy a usar un esquema lineal, en el cual la temperatura nueva es un 95 por ciento de la temperatura actual: $t_{k+1} = 0,95 * t_k$. Si pintamos las gráficas de la evolución de la temperatura con este nuevo esquema obtenemos las siguientes:



Podemos ver como el descenso de la temperatura es mucho menos pronunciado pero, ¿que pasa con los resultados del nuevo algoritmo de Enfriamiento Simulado? Vamos a comparar nuestro algoritmo usando los dos esquemas de enfriamiento ya comentados:

	Sonar				wdbc				Spambase			
	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T	%_clas	%_red	Agreg	T
Enfriamiento Simulado - Esquema de Cauchy												
Set 1	85.7143	21.6667	53.6905	4.704	96.5217	20	58.2609	17.789	78.4946	26.3158	52.4052	21.02
Set 2	88.0952	20	54.0476	4.858	90.4348	23.3333	56.8841	15.611	83.871	24.5614	54.2162	18.525
Set 3	90.4762	21.6667	56.0714	4.478	94.7826	30	62.3913	16.511	74.1936	22.807	48.5003	19.866
Set 4	80.4878	28.3333	54.4106	4.921	94.7826	23.3333	59.058	16.851	82.7957	17.5439	50.1698	20.163
Set 5	87.8049	23.3333	55.5691	4.477	93.8596	16.6667	55.2632	16.722	83.871	22.807	53.339	18.778
Media	86.5157	23	54.7578	4.6876	94.0763	22.6667	58.3715	16.6968	80.6452	22.807	51.7261	19.6704
Enfriamiento Simulado - Esquema de Lineal												
Set 1	88.0952	23.3333	55.7143	17.846	94.7826	33.3333	64.058	30.001	90.3226	21.0526	55.6876	65.291
Set 2	83.3333	18.3333	50.8333	17.15	94.7826	26.6667	60.7246	30.192	87.0968	22.807	54.9519	66.437
Set 3	83.3333	21.6667	52.5	16.265	93.913	26.6667	60.2899	32.948	78.4946	29.8246	54.1596	74.156
Set 4	90.2439	20	55.122	17.01	97.3913	20	58.6957	32.668	80.6452	24.5614	52.6033	67
Set 5	90.2439	18.3333	54.2886	20.225	94.7368	30	62.3684	30.968	89.2473	26.3158	57.7816	66.462
Media	87.0499	20.3333	53.6916	17.6992	95.1213	27.3333	61.2273	31.3554	85.1613	24.9123	55.0368	67.8692

Podemos ver que, hablando de tiempos, el esquema lineal es bastante más lento, ya que tarda más en enfriarse (tal y como hemos visto en las gráficas). En cuanto a porcentajes el esquema lineal obtiene unos mejores resultados para *wdbc* y *spambase* y con cierto margen mientras que en el caso de *sonar* se comporta algo peor.

6.5. Características reducidas.

Finalmente, voy a hacer un pequeño comentario acerca de cuales son las características que han sido reducidas para ciertos conjuntos de datos. En concreto voy a mostrar las características reducidas por el algoritmo *DE/Rand/1* (ya que es el que más reduce y el que mejores resultados obtiene) para las tres bases de datos usadas en esta práctica. En las figuras voy a mostrar las características reducidas con un asterisco. Los resultados son los siguientes:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
sonar_part_1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*													
sonar_part_2	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*															
sonar_part_3	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*															
sonar_part_4	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*															
sonar_part_5	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
wdbc_part_1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*													
wdbc_part_2	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
wdbc_part_3	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
wdbc_part_4	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
wdbc_part_5	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
spambase_part_1	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*													
spambase_part_2	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
spambase_part_3	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
spambase_part_4	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														
spambase_part_5	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*														

Como podemos ver, en el caso de *sonar* hay ciertas características que casi nunca se reducen, como puede ser la característica 12, la cual sólo se consigue reducir en una partición. En el caso de *wdbc* sucede algo similar, las características que 24 y 29 se reducen en 3 de las 5 particiones. En *wdbc* pasa algo curioso, para la partición 1 sólo es “necesaria” una característica para tener una buena tasa de clasificación, la característica 28. Donde se nota más el peso de una característica importante es en *spambase*, ya que la característica 7 nunca se reduce, lo cual nos lleva a pensar que es una característica muy determinante a la hora de realizar la predicción de la clase.

7. Bibliografía.

No se ha usado ninguna bibliografía concreta más allá de los apuntes de teoría y seminarios aportados de la signatura.

Para el código desarrollado en *R* se ha usado como apoyo los propios manuales de *R* (mediante el comando `?*`, donde `*` es la instrucción de la que se quiere obtener información).

Para el código desarrollado en *C++*, como ya se ha comentado anteriormente, se ha usado la página cplusplus para consultar ciertas funciones.