



UNIVERSIDAD DE GRANADA

Práctica 3.b

Búsquedas por Trayectorias para el Problema del Aprendizaje de
Pesos en Características

Metaheurísticas

Grupo 2 (Lunes)

Pedro Ramos Suárez

76591270M

pedrors@correo.ugr.es

Doble Grado de Ingeniería Informática y Matemáticas

7 de junio de 2022

Índice

1. Problema de Aprendizaje de Pesos en Características	2
1.1. Descripción del problema	2
1.2. Datos utilizados	2
2. Aplicación de los algoritmos	3
2.1. Algoritmos para evaluar las soluciones	3
2.2. Algoritmos para calcular las distancias	4
2.3. Algoritmos para predecir las etiquetas	4
2.4. Otros algoritmos	5
2.5. Algoritmos utilizados en los algoritmos genéticos	6
3. Descripción de los algoritmos	10
3.1. Enfriamiento simulado	10
3.2. Búsqueda Multiarranque Básica (BMB)	11
3.3. Búsqueda Local Reiterada (ILS)	11
3.4. ILS-ES	12
4. Algoritmos de comparación	13
4.1. 1-NN	13
4.2. Greedy: RELIEF	13
4.3. Búsqueda Local	14
4.4. Algoritmos genéticos	16
4.5. Algoritmos meméticos	19
5. Desarrollo de la práctica	21
5.1. Manual de usuario	21
6. Experimentos y análisis de resultados	23
6.1. Resultados	23
6.2. Resultados previos	23
6.3. Resultados globales	25
6.4. Análisis de los resultados	25

1. Problema de Aprendizaje de Pesos en Características

1.1. Descripción del problema

El problema consiste en optimizar el rendimiento de un clasificador basado en el vecino más cercano a partir de la inclusión de pesos asociados a las características del problema.

El clasificador utilizado es el 1-NN (1 vecino más cercano), el cuál consiste en asignar a cada nuevo la misma etiqueta que a su vecino más cercano. Para ello, utilizamos la distancia euclídea modificada por unos pesos asociados a cada característica:

$$d(e_1, e_2) = \left(\sum_{i=1}^n w_i \cdot (e_1^i - e_2^i)^2 \right)^{\frac{1}{2}}$$

donde n es el número de características de los datos, y $W = (w_1, \dots, w_n)$ el vector de números reales entre 0 y 1 que define el peso que pondera cada una de las características.

La variante que intentaremos optimizar será la agregación, que combina tanto la precisión como la complejidad del clasificador, definida como:

$$\text{Agregación}(W) = \alpha \cdot \text{Tasa_clas}(W) + (1 - \alpha) \cdot \text{Tasa_red}(W) \quad (1)$$

donde:

$$\text{Tasa_clas} = 100 \cdot \frac{\text{n}^\circ \text{ de instancias bien clasificadas en } T}{\text{n}^\circ \text{ de instancias en } T} \quad (2)$$

$$\text{Tasa_red} = 100 \cdot \frac{\text{n}^\circ \text{ de valores } w_i < 0,1}{\text{n}^\circ \text{ de características}} \quad (3)$$

siendo T el conjunto de datos sobre los que se evalúa el clasificador, y α es un número real entre 0 y 1 que pondera la importancia entre acierto y la reducción de la solución.

1.2. Datos utilizados

Los datos utilizados serán:

1. **Ionosphere:** Conjunto de datos de radar, formado por 351 ejemplos con 34 características clasificados en dos clases.
2. **Parkinsons:** Conjunto de datos orientados a distinguir entre la presencia y ausencia de la enfermedad de Parkinson en una serie de pacientes, formado por 195 ejemplos con 22 características clasificados en dos clases.
3. **Spectf-heart:** Conjunto de datos de detección de enfermedades cardíacas a partir de imágenes médicas de termografía computerizada del corazón, formada por 349 ejemplos con 44 características clasificados en dos clases.

2. Aplicación de los algoritmos

Todos los algoritmos utilizados para resolver este problema tienen como entrada la matriz de datos de las características $D = (k \times n)$, y el vector de etiquetas asociadas a dicha matriz $y = (y_1, \dots, y_k)$, donde k es el número de ejemplos, y n el número de características. Dependiendo del algoritmo, estos datos pueden ser todos los leídos, o sólo un subconjunto de entrenamiento.

Para representar las soluciones, utilizaremos un vector de números reales entre 0 y 1, $W = (w_1, \dots, w_n)$, que define el peso que pondera cada una de las características.

La evaluación de la calidad de una solución se hará como indicada en (1), tomando como $\alpha = 0,5$, es decir, dándole la misma importancia al acierto y a la reducción de las características, por lo que será de la forma:

$$\text{Agregación}(W) = 0,5 \cdot \text{Tasa_clas}(W) + 0,5 \cdot \text{Tasa_red}(W) = \frac{\text{Tasa_clas}(W) + \text{Tasa_red}(W)}{2}$$

2.1. Algoritmos para evaluar las soluciones

El pseudocódigo de la función para calcular la tasa de clasificación es:

Algorithm 1: TASA_CLAS calcula la tasa de clasificación de una solución.

Input: El conjunto de etiquetas reales Y .

Input: El conjunto de etiquetas obtenidas de la predicción del vecino más cercano P .

Output: La tasa de clasificación como se describe en (2).

```
clas ← 0 ;
for i = 0 to length(Y) do
    if Y[i] = P[i] then
        | clas ← clas + 1 ;
    end
end
output ← clas/length(Y) ;
return output ;
```

El pseudocódigo de la función para calcular la tasa de reducción es:

Algorithm 2: TASA_RED calcula la tasa de reducción de una solución.

Input: El conjunto de pesos W .

Input: El conjunto de etiquetas obtenidas de la predicción del vecino más cercano P .

Output: La tasa de clasificación como se describe en (3).

```
red ← 0 ;
for w in W do
    if w < 0,1 then
        | red ← red + 1 ;
    end
end
output ← red/length(W) ;
return output ;
```

Por último, el pseudocódigo de la función de agregación obtenido a partir de la tasa de clasificación y la tasa de reducción es:

Algorithm 3: AGREGACION calcula la agregación de una solución.

Input: El conjunto de pesos W .

Output: La agregación como se describe en (1).

```
clas ← tasa_clas(W) ;
red ← tasa_red(W) ;
output ← (clas + red)/2 ;
return output ;
```

2.2. Algoritmos para calcular las distancias

Utilizaremos la distancia euclídea o una variante que modifica la importancia de cada componente según los pesos.

El pseudocódigo para la distancia euclídea es:

Algorithm 4: EULCIDEAN calcula la distancia entre dos puntos.

Input: Un vector con las coordenadas del primer punto P .

Input: Un vector con las coordenadas del segundo punto Q .

Output: La distancia entre los dos puntos.

```
dist ← 0 ;
for i = 0 to length(P) do
    | dist ← dist + (P[i] - Q[i])2 ;
end
output ← sqrt(dist) ;
return output ;
```

El pseudocódigo para la distancia euclídea modificada por el vector de pesos es:

Algorithm 5: WEIGHTED_EULCIDEAN calcula la distancia entre dos puntos.

Input: Un vector de pesos W .

Input: Un vector con las coordenadas del primer punto P .

Input: Un vector con las coordenadas del segundo punto Q .

Output: La distancia entre los dos puntos.

```
dist ← 0 ;
for i = 0 to length(W) do
    | if W[i] > 0,1 then
    | | dist ← dist + W[i](P[i] - Q[i])2 ;
    | end
end
output ← sqrt(dist) ;
return output ;
```

2.3. Algoritmos para predecir las etiquetas

Tenemos dos casos:

1. El conjunto de datos está dividido en entrenamiento y test. Entonces predeciremos las etiquetas del conjunto de test a partir de los resultados obtenidos en entrenamiento.
2. Los datos no están divididos en entrenamiento y test. Entonces aplicaremos el algoritmo *leave-one-out*, que predice una etiqueta a partir de todos los demás datos.

Para el pseudocódigo utilizaremos la versión de la distancia euclídea con pesos. Sin embargo, el caso que no los utiliza sería casi idéntico, pero sin tomar el vector de pesos como entrada, y utilizando la función *euclidean* en lugar de *weighted_euclidean*.

El pseudocódigo para el primer caso, en el que tenemos los datos separados en entrenamiento y test, es:

Algorithm 6: PREDICT_LABEL predice las etiquetas.

Input: Una matriz de datos de test X_{test} .

Input: Una matriz de datos de entrenamiento X .

Input: Un vector con las etiquetas de entrenamiento Y .

Input: El vector de pesos W .

Output: Un vector con la predicción de etiquetas de los datos de test.

$Y_{pred} \leftarrow \{\}$;

for x **in** X_{test} **do**

$min_dist \leftarrow length(x)$;

$neighbour \leftarrow 0$;

for $i = 0$ **to** $length(X)$ **do**

$dist \leftarrow weighted_euclidean(x, X[i], W)$;

if $dist < min_dist$ **then**

$min_dist \leftarrow dist$;

$neighbour \leftarrow i$;

end

end

$Y_{pred} \leftarrow Y_{pred} \cup \{neighbour\}$;

end

return Y_{pred} ;

Nótese que inicializamos la distancia mínima como el la longitud de x , es decir, el número de características. Esto se debe a que como los datos están normalizados, la distancia siempre será menor.

El pseudocódigo para el segundo caso, *leave-one-out*, en el que tenemos un único conjunto de datos, es:

Algorithm 7: LEAVE_ONE_OUT predice las etiquetas.

Input: Una matriz de datos X .

Input: Un vector con las etiquetas Y .

Input: El vector de pesos W .

Output: Un vector con la predicción de etiquetas.

$Y_{pred} \leftarrow \{\}$;

for x **in** X **do**

$min_dist \leftarrow length(x)$;

$neighbour \leftarrow 0$;

for $i = 0$ **to** $length(X)$ **do**

if $x \neq X[i]$ **then**

$dist \leftarrow weighted_euclidean(x, X[i], W)$;

if $dist < min_dist$ **then**

$min_dist \leftarrow dist$;

$neighbour \leftarrow i$;

end

end

end

$Y_{pred} \leftarrow Y_{pred} \cup \{neighbour\}$;

end

return Y_{pred} ;

2.4. Otros algoritmos

Por último, sólo nos queda un algoritmo que merece la pena mencionar, que es el que separa los datos en entrenamiento:

Algorithm 8: TRAIN_TEST_SPLIT divide los datos en entrenamiento y test.

Input: Una matriz de datos X .

Input: Un vector con las etiquetas Y .

Input: Un entero que indica el número de conjuntos en el que dividir los datos n .

Input: Un entero que indica qué conjunto es de test y cuáles de entrenamiento k .

Output: Un vector con los datos de entrenamiento.

Output: Un vector con los datos de test.

Output: Un vector con las etiquetas de entrenamiento.

Output: Un vector con las etiquetas de test.

$X_{train} \leftarrow \{\}$;

$X_{test} \leftarrow \{\}$;

$Y_{train} \leftarrow \{\}$;

$Y_{test} \leftarrow \{\}$;

$size \leftarrow \text{length}(X)/n$;

$remain \leftarrow \text{length}(X) \% n$;

if $k < remain$ **then**

$begin \leftarrow (size + 1)$;

$end \leftarrow begin + (size + 1)$;

else if $k == remain$ **then**

$begin \leftarrow (size + 1)$;

$end \leftarrow begin + size$;

else

$begin \leftarrow size$;

$end \leftarrow begin + size$;

end

for $i = 0$ **to** $\text{length}(X)$ **do**

if $i < begin$ **or** $i > end$ **then**

$X_{train} \leftarrow X_{train} \cup X[i]$;

$Y_{train} \leftarrow Y_{train} \cup Y[i]$;

else

$X_{test} \leftarrow X_{test} \cup X[i]$;

$Y_{test} \leftarrow Y_{test} \cup Y[i]$;

end

end

return X_{train} ;

return X_{test} ;

return Y_{train} ;

return Y_{test} ;

En caso de que el número de datos sea múltiplo del número de particiones, todas estas tendrán el mismo tamaño. En caso de que no lo sea, los primeros conjuntos de test tendrán un elemento extra.

2.5. Algoritmos utilizados en los algoritmos genéticos

La representación de las soluciones empleadas están explicadas en la sección 2.

La función objetivo es la agregación, definida en el algoritmo 3.

En los algoritmos genéticos utilizamos un algoritmo para inicializar los cromosomas a soluciones aleatorias con una distribución uniforme entre 0 y 1:

Algorithm 9: INITIALIZE inicializa los cromosomas.

Input: Un entero con el número de cromosomas a inicializar *size*.**Input:** Un entero con el número de genes de cada cromosoma *dim*.**Input:** Un generador de números aleatorios con distribución uniforme entre 0 y 1, *generator*.**Output:** Una matriz con los cromosomas inicializados.

```
solutions ← {} ;  
for i in {0, ..., size} do  
    w ← {} ;  
    for j in {0, ..., dim} do  
        w ← w ∪ generator() ;  
    end  
    solutions ← solutions ∪ w ;  
end  
return solutions ;
```

Para seleccionar los padres que cruzaremos, utilizamos un mismo algoritmo para el caso estacionario y el caso generacional, basado en el torneo binario, con la diferencia de que el caso generacional creamos tantos padres como cromosomas, y en el caso estacionario sólo generamos dos. Dicho algoritmo es:

Algorithm 10: SELECTION selecciona los padres a cruzar.

Input: Una matriz con los cromosomas *solutions*.**Input:** Un vector con el “fitness” de los cromosomas *fitness*.**Input:** Un entero con el número de padres a obtener *size*.**Output:** Una matriz con los padres a cruzar.

```
parents ← {} ;  
for i in {0, ..., size} do  
    random1 ← rand() %length(solutions) ;  
    random2 ← rand() %length(solutions) ;  
    if fitness[random1] > fitness[random2] then  
        parents ← parents ∪ solutions[random1] ;  
    else  
        parents ← parents ∪ solutions[random2] ;  
    end  
end  
return parents ;
```

Para el cruce, tenemos dos casos. El primero de ellos es utilizando Blx- α , con $\alpha = 0,3$, con una probabilidad de cruce de 0,7. Este algoritmo es:

Algorithm 11: CROSSBLX cruza los padres utilizando Blx.

Input: Una matriz con los padres *parents*.**Input:** Un generador de números aleatorios con distribución uniforme entre 0 y 1, *generator*.**Output:** Una matriz con los hijos.

```
crossParents  $\leftarrow \{\}$  ;  
crosses  $\leftarrow 0,7 \cdot \text{length}(\text{parents})/2$  ;  
for i in  $\{0, \dots, \text{crosses}\}$  do  
  parent1  $\leftarrow 2 \cdot i$  ;  
  parent2  $\leftarrow (2 \cdot i + 1) \% \text{length}(\text{parents})$  ;  
  for j in  $\{0, 1\}$  do  
    w  $\leftarrow \{\}$  ;  
    for k in  $\text{length}(\text{parents}[\text{parent1}])$  do  
      if  $\text{parents}[\text{parent1}][k] < \text{parents}[\text{parent2}][k]$  then  
        min  $\leftarrow \text{parents}[\text{parent1}][k]$  ;  
        max  $\leftarrow \text{parents}[\text{parent2}][k]$  ;  
      else  
        min  $\leftarrow \text{parents}[\text{parent2}][k]$  ;  
        max  $\leftarrow \text{parents}[\text{parent1}][k]$  ;  
      end  
      min  $\leftarrow \text{min} + 0,3 \cdot (\text{max} - \text{min})$  ;  
      max  $\leftarrow \text{max} + 0,3 \cdot (\text{max} - \text{min})$  ;  
      gene  $\leftarrow \text{min} + \text{generator}() \cdot (\text{max} - \text{min})$  ;  
      if gene  $< 0$  then  
        gene  $\leftarrow 0$  ;  
      else if gene  $> 1$  then  
        gene  $\leftarrow 1$  ;  
      end  
      w  $\leftarrow w \cup \text{gene}$  ;  
    end  
    crossParents  $\leftarrow \text{crossParents} \cup w$  ;  
  end  
end  
for i in  $\{2 \cdot \text{crosses}, \dots, \text{length}(\text{parents})\}$  do  
  crossParents  $\leftarrow \text{crossParents} \cup \text{parents}[i]$  ;  
end  
return crossParents ;
```

Para el cruce aritmético, el algoritmo es similar, pero los genes de los hijos son la media de los genes de los padres. Sin embargo, como dos padres generan dos hijos, y estos dos hijos se generaría de igual forma, obtendríamos los hijos idénticos, por lo que modificamos este algoritmo para que, en lugar de ser la media entre ambos padres, sea un valor aleatorios entre ambos padres, pero no necesariamente la media. El algoritmo es:

Algorithm 12: CROSSCA cruza los padres utilizando cruce aritmético.

Input: Una matriz con los padres *parents*.**Input:** Un generador de números aleatorios con distribución uniforme entre 0 y 1, *generator*.**Output:** Una matriz con los hijos.

```
crossParents  $\leftarrow \{\}$  ;  
crosses  $\leftarrow 0,7 \cdot \text{length}(\text{parents})/2$  ;  
for i in  $\{0, \dots, \text{crosses}\}$  do  
  parent1  $\leftarrow 2 \cdot i$  ;  
  parent2  $\leftarrow (2 \cdot i + 1) \% \text{length}(\text{parents})$  ;  
  alpha  $\leftarrow \text{generator}$  ;  
  for j in  $\{0, 1\}$  do  
    w  $\leftarrow \{\}$  ;  
    if j == 1 then  
       $\alpha \leftarrow 1 - \alpha$  ;  
    end  
    for k in  $\text{length}(\text{parents}[\text{parent1}])$  do  
      gene  $\leftarrow \alpha \times \text{parents}[\text{parent1}][k] + (1 - \alpha)\text{parents}[\text{parent2}][k]$  ;  
      w  $\leftarrow w \cup \text{gene}$  ;  
    end  
    crossParents  $\leftarrow \text{crossParents} \cup w$  ;  
  end  
end  
for i in  $\{2 \cdot \text{crosses}, \dots, \text{length}(\text{parents})\}$  do  
  crossParents  $\leftarrow \text{crossParents} \cup \text{parents}[i]$  ;  
end  
return crossParents ;
```

Por último, nos queda ver la mutación, que ocurre con una probabilidad de 0,1. El algoritmo utiliza la función *permutation*, que únicamente toma como parámetro un entero *n*, y devuelve un array con valores entre $\{0, \dots, n\}$ pero con orden aleatorio. Debido a la simplicidad de este algoritmo, no entraré en detalle en su funcionamiento. El algoritmo de mutación es:

Algorithm 13: MUTATION muta los genes.

Input: Una matriz con los padres *parents*.**Input:** Un generador de números aleatorios con distribución normal de media 0 y varianza 0,3², *generator*.**Output:** Una matriz con los hijo mutados.

```
mutations  $\leftarrow 0,1 \cdot \text{length}(\text{parents}) \cdot \text{length}(\text{parents}[0])$  ;  
perm  $\leftarrow \text{permutation}(\text{length}(\text{parents}) \cdot \text{length}(\text{parents}[0]))$  ;  
for i in  $\{0, \dots, \text{mutations}\}$  do  
  chromosome  $\leftarrow \text{perm}[i] / \text{length}(\text{parents}[0])$  ;  
  gene  $\leftarrow \text{perm}[i] \% \text{length}(\text{parents}[0])$  ;  
  parents[chromosome][gene]  $\leftarrow \text{parents}[\text{chromosome}][\text{gene}] + \text{generator}()$  ;  
  if parents[chromosome][gene] < 0 then  
    parents[chromosome][gene]  $\leftarrow 0$  ;  
  else if parents[chromosome][gene] > 1 then  
    parents[chromosome][gene]  $\leftarrow 1$  ;  
  end  
end  
return parents ;
```

3. Descripción de los algoritmos

3.1. Enfriamiento simulado

Comenzamos con el algoritmo más complicado de esta práctica, el enfriamiento simulado, aunque la mayor parte de la complejidad se debe al cálculo de la temperatura inicial, sus modificaciones y las comprobaciones.

El algoritmo es:

Algorithm 14: ES enfriamiento simulado.

Input: Una matriz con las características x .

Input: Un vector inicial con las etiquetas y .

Input: El número máximo de evaluaciones $evaluations$.

Input: Variable utilizada para calcular la temperatura inicial μ .

Input: Variable utilizada para calcular la temperatura inicial ϕ .

Output: Un vector de pesos con la mejor solución.

$w \leftarrow \text{initialize}(\text{length}(x[0]))$;

$fitness \leftarrow \text{agregation}(x, y, w)$;

$eval \leftarrow 1$;

$exitos \leftarrow 1$;

$maxNeighbours \leftarrow 10 \times \text{length}(x[0])$;

$maxExitos \leftarrow 0,1 \times maxNeighbours$;

$m \leftarrow evaluations / maxNeighbours$;

$bestFitness \leftarrow fitness$;

$bestSolution \leftarrow w$;

$T \leftarrow -\mu \times fitness / \log(\phi)$;

$T_f \leftarrow 10^{-3}$;

while $T < T_f$ **do**

 // La temperatura final tiene que ser menor que la inicial

$T_f \leftarrow T_f / 10$;

end

$\beta \leftarrow (T - T_f) / (M \times T \times T_f)$;

while $exitos > 0$ **and** $eval < evaluations$ **do**

$exitos \leftarrow 0$;

$neighbours \leftarrow 0$;

while $neighbours < maxNeighbours$ **and** $exitos < maxExitos$ **and** $eval < evaluations$ **do**

$newW \leftarrow \text{mutation}(w)$;

$neighbours \leftarrow neighbours + 1$;

$agre \leftarrow \text{agregation}(x, y, newW)$;

$\delta \leftarrow agre - fitness$;

$x \leftarrow$ Número aleatorio en una uniforme $[0,1]$;

if $\delta > 0$ **or** $x \leq \exp(\delta / T)$ **then**

$w \leftarrow newW$;

$fitness \leftarrow agre$;

$exitos \leftarrow exitos ++$;

if $fitness > bestFitness$ **then**

$bestFitness \leftarrow fitness$;

$bestSolution \leftarrow w$;

end

end

end

$T \leftarrow T / (1 + \beta \times T)$;

end

return $bestSolution$;

Utilizaremos $\mu = \phi = 0,3$ y 15000 iteraciones.

3.2. Búsqueda Multiarranque Básica (BMB)

Este algoritmo consiste en realizar la búsqueda local varias veces comenzando con una solución inicial aleatoria distinta en cada caso. De esta forma, podemos ver como afecta la solución inicial a la búsqueda local. El algoritmo es:

Algorithm 15: BMB búsqueda multiarranque básica.

Input: Una matriz con las características x .

Input: Un vector inicial con las etiquetas y .

Input: El número de soluciones iniciales $iterations$.

Input: El número máximo de evaluaciones de cada búsqueda local $evaluations$.

Input: El número máximo de evaluaciones sin modificación tras las cuales se detiene el algoritmo $maxNeighbours$.

Output: Un vector de pesos con la mejor solución.

$bestFitness \leftarrow 0$;

for $iteration$ **in** $\{0, \dots, iterations\}$ **do**

$w \leftarrow initialize(length(x[0]))$;

$w \leftarrow localSearch(x, y, evaluations, maxNeighbours)$;

$agre \leftarrow aggregation(x, y, w)$;

if $agre > bestFitness$ **then**

$bestFitness \leftarrow agre$;

$bestSolution \leftarrow w$;

end

end

return $bestSolution$;

Debido a que si utilizamos la misma semilla, la primera solución aleatoria a la que aplicamos la búsqueda multiarranque básica nos dará los mismos resultados que la búsqueda local, y como nos quedamos con la mejor solución, la búsqueda multiarranque básica siempre va a dar mejores o, al menos, iguales resultados que la búsqueda local.

3.3. Búsqueda Local Reiterada (ILS)

La búsqueda local reiterada es similar a la búsqueda local, pero tras la búsqueda local realizamos una mutación a la solución obtenida, realizamos una mutación más brusca que las de la búsqueda local, y volvemos aplicar la búsqueda local a la mejor solución entre la obtenida de la búsqueda local y la obtenida de la mutación. El pseudo-código es:

Algorithm 16: ILS búsqueda local reiterada.

Input: Una matriz con las características x .**Input:** Un vector inicial con las etiquetas y .**Input:** El número de soluciones iniciales $iterations$.**Output:** Un vector de pesos con la mejor solución.

```
 $w \leftarrow \text{initialize}(\text{length}(x[0])) ;$   
 $w \leftarrow \text{localSearch}(x, y, w) ;$   
 $fitness \leftarrow \text{agregation}(x, y, w) ;$   
 $bestFitness \leftarrow fitness ;$   
 $bestSolution \leftarrow w ;$   
for  $iteration$  in  $\{1, \dots, iterations\}$  do  
   $w \leftarrow \text{mutation}(bestSolution) ;$   
   $w \leftarrow \text{localSearch}(x, y, w) ;$   
   $fitness \leftarrow \text{agregation}(newW, y, w) ;$   
  if  $fitness > bestFitness$  then  
     $bestFitness \leftarrow bestFitness ;$   
     $bestSolution \leftarrow w ;$   
  end  
end  
return  $bestSolution ;$ 
```

Al igual que con la búsqueda multiarranque básica, antes de realizar la primera mutación tenemos la misma solución que la obtenida con la búsqueda local, por lo que los resultados siempre serán iguales o mejores.

3.4. ILS-ES

Finalmente, el último algoritmo de esta práctica será la búsqueda local reiterada pero utilizando el enfriamiento simulado en lugar de la búsqueda local. Debido a que ya hemos explicado estos dos algoritmos por separado previamente, no entraré en detalle en esta sección. El algoritmo es:

Algorithm 17: ILS búsqueda local reiterada.

Input: Una matriz con las características x .**Input:** Un vector inicial con las etiquetas y .**Input:** El número de soluciones iniciales $iterations$.**Output:** Un vector de pesos con la mejor solución.

```
 $w \leftarrow \text{initialize}(\text{length}(x[0])) ;$   
 $w \leftarrow \text{es}(x, y, w) ;$   
 $fitness \leftarrow \text{agregation}(x, y, w) ;$   
 $bestFitness \leftarrow fitness ;$   
 $bestSolution \leftarrow w ;$   
for  $iteration$  in  $\{1, \dots, iterations\}$  do  
   $w \leftarrow \text{mutation}(bestSolution) ;$   
   $w \leftarrow \text{es}(x, y, w) ;$   
   $fitness \leftarrow \text{agregation}(newW, y, w) ;$   
  if  $fitness > bestFitness$  then  
     $bestFitness \leftarrow bestFitness ;$   
     $bestSolution \leftarrow w ;$   
  end  
end  
return  $bestSolution ;$ 
```

4. Algoritmos de comparación

Debido a que todos estos algoritmos fueron estudiados e implementados en la práctica anterior, no se incluye en esta entrega. La descripción, pseudo-código y resultados obtenidos son los de la práctica anterior.

4.1. 1-NN

El primer algoritmo que utilizaremos para comparar la eficacia del algoritmo es *1-NN*, que es el algoritmo **k-NN** (*k* nearest neighbours) con $k = 1$, es decir, con un sólo vecino.

Los demás algoritmos son una versión modificada de este, en los que modificamos la distancia euclídea con un vector de pesos. Debido a ello, como no tenemos vector de pesos, no estamos realizando ninguna optimización, y los resultados solo dependen de los datos de entrada, por lo que no tiene sentido dividir los datos en entrenamiento y test (ya que no “entrenamos”), así que usamos la técnica *leave-one-out*.

El funcionamiento del algoritmo ya ha sido explicado en el algoritmo 7.

4.2. Greedy: RELIEF

El principal algoritmo que utilizaremos para comparar la eficacia es el *greedy RELIEF*, que genera un vector de pesos a partir de las distancias de cada ejemplo a su *enemigo*, que es el ejemplo de diferente clase más cercano, y a su *amigo* más cercano, que es el ejemplo de la misma clase más cercano.

El algoritmo aumentará el peso de las características que mejor separan a ejemplos enemigos, y reduce el peso en las que separan a amigos. El incremento es proporcional a la distancia entre los ejemplos en cada característica.

Algorithm 18: RELIEF

Input: Una matriz de datos X .

Input: Un vector de etiquetas Y .

Output: Un vector de pesos.

$W \leftarrow \{0, 0, \dots, 0\}$ // Tantos ceros como características tengan los datos.

```
for  $i = 0$  to  $\text{length}(X)$  do
     $\text{ind\_enemy} \leftarrow i$  ;
     $\text{ind\_friend} \leftarrow i$  ;
     $\text{dist\_enemy} \leftarrow \text{length}(X[0])$  // Al normalizar los datos, la distancia máxima
     $\text{dist\_friend} \leftarrow \text{length}(X[0])$  // es  $\sqrt{k} < k$ , siendo  $k$  el número de características
    for  $j = 0$  to  $\text{length}(X)$  do
        if  $i \neq j$  then
            if  $X[i] == X[j]$  and  $\text{weighted\_euclidean}(X[i], X[j], W) < \text{dist\_friend}$  then
                 $\text{ind\_friend} \leftarrow j$  ;
                 $\text{dist\_friend} \leftarrow \text{weighted\_euclidean}(X[i], X[j], W)$  ;
            else if  $X[i] \neq X[j]$  and  $\text{weighted\_euclidean}(X[i], X[j], W) < \text{dist\_enemy}$  then
                 $\text{ind\_enemy} \leftarrow j$  ;
                 $\text{dist\_enemy} \leftarrow \text{weighted\_euclidean}(X[i], X[j], W)$  ;
            end
        end
    end
    for  $j = 0$  to  $\text{length}(X[0])$  do
         $w[j] \leftarrow |X[i][j] - x[\text{ind\_enemy}][j]| - |X[i][j] - X[\text{ind\_friend}][j]|$  ;
    end
end
 $W \leftarrow \text{normalize\_weight}(W)$  ;
return  $W$  ;
```

donde *normalize_weight* es la función que normaliza los valores del vector de pesos, haciendo que estén entre 0 y 1, y aplica la reducción, eliminando aquellos valores que están por debajo de 0.1, es decir:

Algorithm 19: NORMALIZE_WEIGHT normaliza y aplica reducción al vector de pesos.

Input: El vector de pesos W .

Output: El vector de pesos normalizado.

$max_w \leftarrow w[0]$;

for k **in** W **do**

if $k > max_w$ **then**

$max_w \leftarrow w$;

end

end

for k **in** W **do**

if $k < 0$ **then**

$k \leftarrow 0$;

else

$k \leftarrow k/max_w$;

end

end

return W ;

4.3. Búsqueda Local

Por último, nos queda el algoritmo de Búsqueda Local, el objetivo de la práctica previa. Generamos nuevas soluciones modificando de manera aleatoria el vector de pesos. Debido a esto, hay infinitas posibles nuevas soluciones, por lo que utilizamos la técnica del Primero Mejor, con la cuál tomamos una nueva solución si es mejor que la actual.

Iniciaremos el algoritmo con una solución aleatoria generando cada componente a partir de una distribución uniforme entre 0 y 1, utilizando la función:

Algorithm 20: RANDOM_SOL genera una solución aleatoria.

Input: El tamaño del vector de pesos n .

Output: Un vector con una solución aleatoria.

$W \leftarrow \{\}$;

for $i = 0$ **to** $length(X[0])$ **do**

$e \leftarrow$ elemento aleatorio de una distribución uniforme ;

$W \leftarrow W \cup \{e\}$;

end

return W ;

En cada paso de la exploración modificamos una componente del vector de pesos distinta sin repetición hasta encontrar mejora (técnica del Primero Mejor) o hasta modificar todas las componentes sin conseguir una mejora, momento en el cual se comienza de nuevo la exploración. Para ello, necesitamos una función que nos de el orden en el que aplicar las modificaciones de forma aleatoria:

Algorithm 21: PERMUTATION genera una permutación.

Input: El tamaño del vector de pesos n .

Output: Un vector con la permutación.

$perm \leftarrow \{0, \dots, n-1\}$;

$perm \leftarrow shuffle(perm)$;

return $perm$;

Continuaremos generando candidatos hasta realizar un total de 15000 iteraciones (contando como iteración cada vez que modificamos una componente del vector de pesos), o 20 iteraciones sin obtener una solución mejor (contando como iteración modificar todas las componentes del vector de pesos).

Cabe mencionar que para obtener el número de características utilizamos $length(X[0])$, ya que, a menos que usemos un conjunto vacío (caso que no tiene sentido), siempre tendremos un elemento en esa posición.

Con todo esto, el algoritmo de búsqueda local queda de la forma:

Algorithm 22: LOCALSEARCH

Input: Una matriz de datos X .

Input: Un vector de etiquetas Y .

Input: El número máximo de evaluaciones $eval$.

Input: El número máximo de evaluaciones sin modificación tras las cuales se detiene el algoritmo $maxNeighbours$.

Output: Un vector de pesos.

$iteration \leftarrow 0$;

$iteration_mod \leftarrow 0$;

$W \leftarrow \text{random_sol}(length(X[0]))$;

while $iteration < eval$ **and** $iteration_mod < maxNeighbours$ **do**

$modified \leftarrow false$;

$perm \leftarrow \text{permutation}(length(W))$;

for $i == 0$ **to** $length(W)$ **and not** $modified$ **and** $iteration < 15000$ **do**

$s \leftarrow$ elemento aleatorio de una distribución normal de media 0 y varianza 0.3 ;

$neighbour \leftarrow \text{copy}(W)$;

$neighbour[i] \leftarrow neighbour[i] + s$;

if $neighbour[i] > 1$ **then**

$neighbour[i] \leftarrow 1$;

else if $neighbour[i] < 0,1$ **then**

$neighbour[i] \leftarrow 0$;

end

if $\text{agregation}(neighbour) > \text{agregation}(W)$ **then**

$W \leftarrow neighbour$;

$modified \leftarrow true$;

end

$iteration \leftarrow iteration + 1$;

end

if $modified$ **then**

$iteration_mod \leftarrow 0$;

else

$iteration_mod \leftarrow iteration_mod + 1$;

end

end

return W ;

4.4. Algoritmos genéticos

Para los cuatro algoritmos genéticos, la idea general es la misma:

Algorithm 23: GENETIC_ALGORITHM algoritmo genético.

Input: Una matriz con los caracteríssticas x .

Input: Un vector con las etiquetas y .

Input: El número de cromosomas $chromosomes$.

Input: El número de iteraciones $iterations$.

Input: El valor por debajo del cuál el peso se considera cero, $reduction$.

Input: La probabilidad de mutación, $mutationChance$.

Output: Una vector con los pesos de la solución.

$eval \leftarrow 0$;

$solutions \leftarrow initialize(chromosomes, length(w[0]))$;

$fitness \leftarrow \{\}$;

$uniform \leftarrow$ Un generador de distribución uniforme entre 0 y 1. ;

$normal \leftarrow$ Un generador de distribución normal de media 0 y varianza $0,3^2$. ;

for i **in** $\{0, \dots, length(solutions)\}$ **do**

$fitness \leftarrow fitness \cup aggregation(solutions[i])$;

$eval \leftarrow eval + 1$;

end

while $eval < iterations$ **do**

$size \leftarrow 2$ // En el caso estacionario

$size \leftarrow length(solutions)$ // En el caso generacional

$parents \leftarrow selection(solutions, fitness, size)$;

$crossParents \leftarrow cross(parents, uniform)$;

$crossParents \leftarrow mutation(crossParents, normal)$;

$solutions \leftarrow replace(crossParents, fitness, eval)$;

end

$ind \leftarrow 0$;

$bestFitness \leftarrow fitness[0]$;

for i **in** $\{1, \dots, length(fitness)\}$ **do**

if $fitness[i] > bestFitness$ **then**

$ind \leftarrow i$;

$bestFitness \leftarrow fitness[i]$;

end

end

return $solutions[ind]$;

El cruce utilizado puede ser el cruce aritmético o Blx, y todas las funciones utilizadas están definidas en 2.5, con la excepción de replace. Esta función depende de si estamos en el caso generacional o estacionario, siendo en el caso generacional:

Algorithm 24: REPLACE algoritmo generacional elitista utilizado para reemplazar las soluciones.

Input: Una matriz con los cromosomas obtenidos del cruce y mutación w .

Input: Un vector con la agregación de las soluciones $fitness$.

Input: Número de evaluaciones realizadas $eval$.

Output: Una matriz con los cromosomas para la siguiente generación.

```
 $bestSolution \leftarrow w[0]$  ;  
 $bestFitness \leftarrow fitness[0]$  ;  
for  $i$  in  $\{1, \dots, length(fitness)\}$  do  
    if  $fitness[i] > bestFitness$  then  
         $bestSolution \leftarrow w[i]$  ;  
         $bestFitness \leftarrow fitness[i]$  ;  
    end  
end  
 $solutions \leftarrow w$  ;  
for  $i$  in  $\{0, \dots, length(solutions)\}$  do  
     $fitness \leftarrow fitness \cup aggregation(solutions[i])$  ;  
     $eval \leftarrow eval + 1$  ;  
end  
 $ind \leftarrow 0$  ;  
 $worstFitness \leftarrow fitness[0]$  ;  
for  $i$  in  $\{1, \dots, length(fitness)\}$  do  
    if  $fitness[i] < worstFitness$  then  
         $ind \leftarrow i$  ;  
         $worstFitness \leftarrow fitness[i]$  ;  
    end  
end  
 $solutions \leftarrow solutions \setminus solutions[ind]$  ;  
 $solutions \leftarrow solutions \cup bestSolution$  ;  
return  $solutions$  ;
```

En el caso estacionario, como sólo sustituimos dos soluciones, es algo más complejo:

Algorithm 25: REPLACE algoritmo estacionario utilizado para reemplazar las soluciones.

Input: Una matriz con los dos cromosomas obtenidos del cruce y mutación w .

Input: Una matriz con los cromosomas de la generación previa $solutions$.

Input: Un vector con la agregación de las soluciones $fitness$.

Input: Número de evaluaciones realizadas $eval$.

Output: Una matriz con los cromosomas para la siguiente generación.

$agregation1 \leftarrow agregation(w[0])$;

$agregation2 \leftarrow agregation(w[1])$;

$eval \leftarrow eval + 2$;

$minPos1 \leftarrow 0$;

$minPos2 \leftarrow 1$;

if $fitness[minPos2] < fitness[minPos1]$ **then**

$minPos1 \leftarrow 1$;

$minPos2 \leftarrow 0$;

end

for i **in** $\{2, \dots, length(fitness)\}$ **do**

if $fitness[i] < fitness[minPos1]$ **then**

$minPos2 \leftarrow minPos1$;

$minPos1 \leftarrow i$;

else if $fitness[i] < fitness[minPos2]$ **then**

$minPos2 \leftarrow i$;

end

end

if $agregation1 > agregation2$ **then**

if $agregation1 > fitness[minPos2]$ **then**

$solutions[minPos2] \leftarrow w[0]$;

$fitness[minPos2] \leftarrow agregation1$;

if $agregation2 > fitness[minPos1]$ **then**

$solutions[minPos1] \leftarrow w[1]$;

$fitness[minPos1] \leftarrow agregation2$;

end

else if $agregation1 > fitness[minPos1]$ **then**

$solutions[minPos1] \leftarrow w[0]$;

$fitness[minPos1] \leftarrow agregation1$;

end

else

if $agregation2 > fitness[minPos2]$ **then**

$solutions[minPos2] \leftarrow w[1]$;

$fitness[minPos2] \leftarrow agregation2$;

if $agregation1 > fitness[minPos1]$ **then**

$solutions[minPos1] \leftarrow w[0]$;

$fitness[minPos1] \leftarrow agregation1$;

end

else if $agregation2 > fitness[minPos1]$ **then**

$solutions[minPos1] \leftarrow w[1]$;

$fitness[minPos1] \leftarrow agregation2$;

end

end

return $solutions$;

Nótese que parte de las operaciones que hacemos es para asegurarnos que $fitness[minPos1] < fitness[minPos2]$.

Aunque parece complejo, es bastante simple, aunque tenemos que tener en cuenta todos los casos dependiendo de que valor sea mayor de las dos nuevas soluciones, realizando todas las comprobaciones necesarias.

4.5. Algoritmos meméticos

Utilizamos el algoritmo generacional utilizando cruce Blx. Es muy similar al algoritmo genético, pero añadiendo la búsqueda local.

(Nota: Este código este disminuido en tamaño debido a su longitud, y siendo prácticamente igual que los algoritmos genéticos excepto por la parte de la búsqueda local, no tiene sentido, volver a detallarlo. La parte añadida se encuentra en el **if** que tiene el comentario `//Algoritmo memético`, después de la selección y mutación.)

Algorithm 26: MEMETIC_ALGORITHM algoritmo genético.

```

Input: Una matriz con los caracteríssticas x.
Input: Un vector con las etiquetas y.
Input: El número de cromosomas chromosomes.
Input: El número de iteraciones iterations.
Input: El valor por debajo del cuál el peso se considera cero, reduction.
Input: La probabilidad de mutación, mutationChance.
Input: El número de generaciones tras las cuales se aplica la búsqueda local, generationStep.
Input: El porcentaje de soluciones a las que aplicamos la búsqueda local, size.
Input: Si aplicamos búsqueda local a soluciones aleatorias o a las mejores, sort.
Output: Una vector con los pesos de la solución.
eval  $\leftarrow$  0 ;
generation  $\leftarrow$  0 ;
solutions  $\leftarrow$  initialize(chromosomes, length(w[0])) ;
fitness  $\leftarrow$  {} ;
uniform  $\leftarrow$  Un generador de distribución uniforme entre 0 y 1. ;
normal  $\leftarrow$  Un generador de distribución normal de media 0 y varianza 0,32. ;
for i in {0, ..., length(solutions)} do
    fitness  $\leftarrow$  fitness  $\cup$  agregation(solutions[i]) ;
    eval  $\leftarrow$  eval + 1 ;
end
while eval < iterations do
    generation  $\leftarrow$  generation + 1 ;
    size  $\leftarrow$  length(solutions) ;
    parents  $\leftarrow$  selection(solutions, fitness, size) ;
    crossParents  $\leftarrow$  cross(parents, uniform) ;
    crossParents  $\leftarrow$  mutation(crossParents, normal) ;
    if generation % generationStep == 0 then
        // Algoritmo memético
        memeticSize  $\leftarrow$  length(crossParents)  $\times$  size ;
        if sort then
            ind  $\leftarrow$  sortInd(fitness, memeticSize) ;
        else
            ind  $\leftarrow$  {0, ..., memeticSize} ;
        end
        for i in {0, ..., memeticSize} do
            crossParents[ind[i]]  $\leftarrow$  localSearch(crossParents[ind[i]], 2  $\times$  length(crossParents[ind[i]]), eval) ;
        end
    end
    solutions  $\leftarrow$  replace(crossParents, fitness, eval) ;
end
ind  $\leftarrow$  0 ;
bestFitness  $\leftarrow$  fitness[0] ;
for i in {1, ..., length(fitness)} do
    if fitness[i] > bestFitness then
        ind  $\leftarrow$  i ;
        bestFitness  $\leftarrow$  fitness[i] ;
    end
end
return solutions[ind] ;

```

donde nos aparecen dos funciones nuevas: `sortInd(size, fitness)` que simplemente devuelve un vector de tamaño *size* que contiene la posición de los mayores valores en *fitness*, debido a su simplicidad, no la desarrollaré, y `localSearch(solution, iter, eval)`, que es la búsqueda local, muy similar a la desarrollada en la siguiente sección, con las siguientes diferencias:

- La solución no se inicializa a una aleatoria, si no que utilizada la que pasamos por parámetro.
- Se detiene cuando realiza tantas iteraciones como *iter*, o cuando alcanza el máximo de evaluaciones del algoritmo memético.
- No se detiene tras realizar un número específico de iteraciones sin realizar ninguna modificación.

Debido a esta similitud, no la desarrollaré en esta sección.

5. Desarrollo de la práctica

La práctica ha sido implementada en C++, utilizando las siguientes bibliotecas:

- **iostream**: Para salida de datos por pantalla.
- **fstream**: Para lectura y escritura en ficheros.
- **sstream**: Para lectura y escritura en ficheros.
- **string**: Para lectura y escritura en ficheros.
- **vector**: Para almacenar los datos en un vector de vectores (que funciona como una matriz), y las etiquetas y los pesos en un vector.
- **math.h**: Para el calculo de la raíz cuadrada para la distancia euclídea.
- **stdlib.h**: Para la generación de distribuciones aleatorios.
- **random**: Para la generación de aleatorios.
- **chrono**: Para calcular el tiempo de ejecución utilizando *system_clock*.
- **regex**: Para modificar strings. Utilizado para poder exportar datos y representarlos en gráficas.
- **algorithm**: Para desordenar el orden de los datos de entrada (manteniendo cada conjunto de características con su etiqueta correspondiente).
- **io manip**: Para la salida en forma de tablas.

Los ficheros están almacenados en los siguientes directorios:

- **bin**: Contiene los archivos ejecutables.
- **src**: Contiene los archivos con el código.
- **data**: Contiene los ficheros con los datos.

Dentro de los ficheros, tenemos las constantes:

- **SEED**: Contiene la semilla usada para las generaciones aleatorias. Por defecto, 0.
- **FOLDS**: Contiene el número de particiones en las que dividir el conjunto de datos para entrenamiento y test usando k-folds cross validation. Tomará uno de estos conjuntos como test, y el resto como entrenamiento. Por defecto, 5.
- **REDUCTION**: Contiene el umbral por debajo del cual los pesos se consideran como 0. Por defecto, 0.1.
- **ITERATIONS**: Número de iteraciones que realiza el algoritmo. Utilizado para el número de soluciones iniciales en *BMB* y de iteraciones en *ILS*. Por defecto, 15.
- **EVALUATIONS**: Número máximo de evaluaciones. En los algoritmos que tienen varias iteraciones, será el número máximo de evaluaciones por iteración. Por defecto, 15000 en los algoritmos que no tienen distintas iteraciones, y 1000 en los que si tienen varias iteraciones (para que así sean $15 * 1000 = 15000$ iteraciones en total).
- **MUTATIONCHANCE**: Probabilidad de mutación. Determina el número de mutaciones que se realizan, siendo el número de elementos del vector de pesos por esta probabilidad. Por defecto 0'1.

5.1. Manual de usuario

Para obtener los ficheros ejecutables, sólo tenemos que realizar el comando *make*, el cuál compila el código fuente usando optimización *-O2*.

Los ficheros ejecutables son:

- **es**: Implementación del algoritmo de enfriamiento simulado.
- **bmb**: Implementación del algoritmo de búsqueda multiarranque básica.
- **ils**: Implementación del algoritmo de búsqueda local reiterada utilizando búsqueda local.
- **ils-es**: Implementación del algoritmo de búsqueda local reiterada utilizando enfriamiento simulado.

Los archivos de código contienen una explicación de todas las funciones, así como de los parámetros que toman de entrada y de salida. Además, al comienzo de cada uno de ellos, están definidas las constantes en caso de que se deseen modificar.

Por ejemplo, si queremos ejecutar el algoritmo *es* con los datos de *ionosphere.arff*, sería:

```
bin/es data/ionosphere.arff
```

6. Experimentos y análisis de resultados

Todos los resultados se han obtenido en un ordenador con las siguientes especificaciones:

- Sistema operativo: macOS Monterey.
- Procesador: 1,6 GHz Intel Core i5 de doble núcleo.
- Memoria: 16 GB 2133 MHz LPDDR3.
- Gráficos: Intel UHD Graphics 617 1536 MB.

6.1. Resultados

En todos los resultados, el tiempo es en segundos.

Los resultados obtenidos con el algoritmo de *enfriamiento simulado* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	94.37	91.18	92.77	258.62	92.31	90.91	91.61	75.12	77.14	93.18	85.16	265.86
Partición 2	92.86	88.24	90.55	262.21	94.87	90.91	92.89	77.90	84.29	93.18	88.73	269.45
Partición 3	81.43	88.24	84.83	262.14	92.31	90.91	91.61	78.00	88.57	90.91	89.74	270.48
Partición 4	92.86	88.24	90.55	262.32	94.87	90.91	92.89	77.99	81.43	93.18	87.31	269.34
Partición 5	94.29	88.24	91.26	260.80	87.18	90.91	89.04	78.31	81.16	93.18	87.17	269.38
Media	91.16	88.82	89.99	261.22	92.31	90.91	91.61	77.46	82.52	92.73	87.62	268.90

Tabla 1: Tabla con los resultados del algoritmo de enfriamiento simulado.

Los resultados obtenidos con el algoritmo *BMB* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	92.96	82.35	87.66	272.66	87.18	90.91	89.04	75.44	81.43	79.55	80.49	287.38
Partición 2	92.86	82.35	87.61	267.68	94.87	90.91	92.89	77.79	88.57	81.82	85.19	289.36
Partición 3	91.43	82.35	86.89	268.17	92.31	81.82	87.06	73.56	81.43	79.55	80.49	283.77
Partición 4	91.43	82.35	86.89	268.63	92.31	86.36	89.34	70.17	84.29	86.36	85.32	284.54
Partición 5	97.14	82.35	89.75	268.89	87.18	90.91	89.04	75.93	82.61	84.09	83.35	284.07
Media	93.16	82.35	87.76	269.21	90.77	88.18	89.48	74.58	83.66	82.27	82.97	285.82

Tabla 2: Tabla con los resultados del algoritmo BMB.

Los resultados obtenidos con el algoritmo *ILS* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	90.14	91.18	90.66	142.46	89.74	86.36	88.05	31.67	81.43	93.18	87.31	169.43
Partición 2	87.14	88.24	87.69	151.43	79.49	90.91	85.20	31.66	91.43	90.91	91.17	173.24
Partición 3	84.29	88.24	86.26	144.94	79.49	86.36	82.93	27.76	88.57	93.18	90.88	173.11
Partición 4	80.00	91.18	85.59	135.54	97.44	72.73	85.08	26.57	85.71	93.18	89.45	172.10
Partición 5	82.86	88.24	85.55	158.68	82.05	90.91	86.48	31.42	82.61	93.18	87.90	172.80
Media	84.89	89.41	87.15	146.61	85.64	85.45	85.55	29.82	85.95	92.73	89.34	172.14

Tabla 3: Tabla con los resultados del algoritmo ILS.

Los resultados obtenidos con el algoritmo *ILS-ES* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	90.14	91.18	90.66	168.84	84.62	86.36	85.49	49.42	80.00	93.18	86.59	174.57
Partición 2	88.57	88.24	88.40	173.49	87.18	90.91	89.04	50.79	91.43	93.18	92.31	177.11
Partición 3	88.57	91.18	89.87	173.77	87.18	90.91	89.04	50.98	81.43	90.91	86.17	176.85
Partición 4	81.43	91.18	86.30	173.25	89.74	90.91	90.33	51.21	84.29	90.91	87.60	176.63
Partición 5	92.86	88.24	90.55	173.74	100.00	90.91	95.45	51.05	84.06	90.91	87.48	178.26
Media	88.31	90.00	89.16	172.62	89.74	90.00	89.87	50.69	84.24	91.82	88.03	176.68

Tabla 4: Tabla con los resultados del algoritmo ILS-ES.

6.2. Resultados previos

Los resultados obtenidos con el algoritmo *greedy RELIEF* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	90.14	2.94	46.54	0.02	94.87	0.00	47.44	0.01	77.14	0.00	38.57	0.02
Partición 2	70.00	2.94	36.47	0.02	100.00	0.00	50.00	0.01	84.29	0.00	42.14	0.02
Partición 3	67.14	2.94	35.04	0.02	94.87	0.00	47.44	0.01	77.14	0.00	38.57	0.02
Partición 4	68.57	2.94	35.76	0.02	97.44	0.00	48.72	0.01	92.86	0.00	46.43	0.02
Partición 5	67.14	2.94	35.04	0.02	94.87	0.00	47.44	0.01	86.96	0.00	43.48	0.02
Media	72.60	2.94	37.77	0.02	96.41	0.00	48.21	0.01	83.68	0.00	41.84	0.02

Tabla 5: Tabla con los resultados del algoritmo greedy RELIEF.

Los resultados obtenidos con el algoritmo *localSearch* son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	92.96	88.24	90.60	20.36	87.18	86.36	86.77	5.21	74.29	79.55	76.92	38.65
Partición 2	92.86	76.47	84.66	27.81	97.44	50.00	73.72	1.98	88.57	56.82	72.69	19.85
Partición 3	88.57	73.53	81.05	25.90	87.18	72.73	79.95	3.46	81.43	81.82	81.62	57.76
Partición 4	82.86	91.18	87.02	35.75	89.74	72.73	81.24	2.52	87.14	77.27	82.21	30.56
Partición 5	87.14	82.35	84.75	25.35	92.31	81.82	87.06	4.34	79.71	70.45	75.08	34.85
Media	88.88	82.35	85.62	27.04	90.77	72.73	81.75	3.50	82.23	73.18	77.70	36.33

Tabla 6: Tabla con los resultados del algoritmo localSearch.

Los resultados obtenidos con el algoritmo genético generacional con cruce Blx son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	92.96	88.24	90.60	171.09	76.92	86.36	81.64	51.63	72.86	79.55	76.20	180.21
Partición 2	92.86	88.24	90.55	172.73	82.05	86.36	84.21	51.73	82.86	79.55	81.20	179.92
Partición 3	87.14	88.24	87.69	172.89	92.31	86.36	89.34	51.66	84.29	81.82	83.05	179.39
Partición 4	80.00	85.29	82.65	173.66	92.31	81.82	87.06	51.79	84.29	77.27	80.78	180.42
Partición 5	91.43	88.24	89.83	172.56	94.87	81.82	88.34	51.68	82.61	79.55	81.08	181.21
Media	88.88	87.65	88.26	172.58	87.69	84.55	86.12	51.70	81.38	79.55	80.46	180.23

Tabla 7: Tabla con los resultados del algoritmo genético generacional con cruce Blx.

Los resultados obtenidos con el algoritmo genético generacional con cruce aritmético son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	91.55	82.35	86.95	171.90	87.18	86.36	86.77	50.94	74.29	81.82	78.05	180.07
Partición 2	85.71	88.24	86.97	173.62	87.18	86.36	86.77	51.12	91.43	84.09	87.76	179.71
Partición 3	81.43	88.24	84.83	173.42	92.31	90.91	91.61	50.92	84.29	81.82	83.05	180.13
Partición 4	87.14	85.29	86.22	173.13	94.87	90.91	92.89	50.98	88.57	84.09	86.33	179.72
Partición 5	87.14	94.12	90.63	172.46	87.18	90.91	89.04	51.02	82.61	81.82	82.21	181.16
Media	86.60	87.65	87.12	172.90	89.74	89.09	89.42	51.00	84.24	82.73	83.48	180.16

Tabla 8: Tabla con los resultados del algoritmo genético generacional con cruce aritmético.

Los resultados obtenidos con el algoritmo genético estacionario con cruce Blx son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	87.32	82.35	84.84	171.37	94.87	68.18	81.53	51.41	80.00	70.45	75.23	179.69
Partición 2	87.14	64.71	75.92	174.45	92.31	72.73	82.52	51.40	88.57	79.55	84.06	177.85
Partición 3	87.14	91.18	89.16	171.12	79.49	86.36	82.93	50.77	80.00	72.73	76.36	177.98
Partición 4	90.00	85.29	87.65	171.16	89.74	77.27	83.51	51.10	87.14	81.82	84.48	177.03
Partición 5	90.00	85.29	87.65	173.68	94.87	68.18	81.53	51.43	82.61	86.36	84.49	182.23
Media	88.32	81.76	85.04	172.36	90.26	74.55	82.40	51.22	83.66	78.18	80.92	178.96

Tabla 9: Tabla con los resultados del algoritmo genético estacionario con cruce Blx.

Los resultados obtenidos con el algoritmo genético estacionario con cruce aritmético son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	90.14	88.24	89.19	169.85	82.05	81.82	81.93	51.08	84.29	88.64	86.46	176.35
Partición 2	85.71	91.18	88.45	170.59	82.05	86.36	84.21	51.07	90.00	93.18	91.59	175.32
Partición 3	87.14	88.24	87.69	171.43	92.31	72.73	82.52	51.88	78.57	86.36	82.47	176.10
Partición 4	90.00	88.24	89.12	171.05	92.31	86.36	89.34	51.17	81.43	90.91	86.17	175.25
Partición 5	90.00	88.24	89.12	171.64	82.05	81.82	81.93	51.12	84.06	88.64	86.35	178.24
Media	88.60	88.82	88.71	170.91	86.15	81.82	83.99	51.26	83.67	89.55	86.61	176.25

Tabla 10: Tabla con los resultados del algoritmo genético estacionario con cruce aritmético.

Los resultados obtenidos con el algoritmo memético aplicando búsqueda local a un 100 % de la población:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	85.92	88.24	87.08	173.97	82.05	86.36	84.21	48.55	85.71	79.55	82.63	181.73
Partición 2	84.29	88.24	86.26	176.22	94.87	90.91	92.89	51.88	87.14	84.09	85.62	185.15
Partición 3	87.14	85.29	86.22	176.21	92.31	90.91	91.61	53.12	85.71	70.45	78.08	180.63
Partición 4	90.00	88.24	89.12	174.46	92.31	86.36	89.34	52.98	87.14	65.91	76.53	180.50
Partición 5	85.71	85.29	85.50	173.31	94.87	81.82	88.34	53.01	82.61	72.73	77.67	182.03
Media	86.61	87.06	86.84	174.83	91.28	87.27	89.28	51.91	85.66	74.55	80.11	182.01

Tabla 11: Tabla con los resultados del algoritmo memético aplicando búsqueda local a un 100 % de la población.

Los resultados obtenidos con el algoritmo memético aplicando búsqueda local a un 10 % aleatorio de la población:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	87.32	91.18	89.25	155.86	92.31	90.91	91.61	44.62	81.43	93.18	87.31	171.84
Partición 2	84.29	91.18	87.73	169.06	87.18	86.36	86.77	49.53	82.86	88.64	85.75	179.17
Partición 3	85.71	91.18	88.45	166.96	87.18	90.91	89.04	50.96	84.29	86.36	85.32	175.63
Partición 4	88.57	91.18	89.87	166.82	94.87	90.91	92.89	51.14	91.43	90.91	91.17	174.31
Partición 5	90.00	91.18	90.59	165.89	87.18	86.36	86.77	51.00	86.96	79.55	83.25	178.82
Media	87.18	91.18	89.18	164.92	89.74	89.09	89.42	49.45	85.39	87.73	86.56	175.95

Tabla 12: Tabla con los resultados del algoritmo memético aplicando búsqueda local a un 10 % aleatorio de la población.

Los resultados obtenidos con el algoritmo memético aplicando búsqueda local al mejor 10 % de la población:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
Partición 1	85.92	58.82	72.37	162.08	89.74	86.36	88.05	46.30	74.29	63.64	68.96	172.92
Partición 2	91.43	85.29	88.36	172.29	82.05	86.36	84.21	49.51	85.71	34.09	59.90	186.32
Partición 3	85.71	41.18	63.45	173.79	79.49	86.36	82.93	50.53	81.43	54.55	67.99	180.85
Partición 4	82.86	88.24	85.55	168.91	87.18	90.91	89.04	50.61	87.14	34.09	60.62	181.53
Partición 5	91.43	76.47	83.95	169.89	84.62	86.36	85.49	50.30	88.41	63.64	76.02	181.94
Media	87.47	70.00	78.73	169.39	84.62	87.27	85.94	49.45	83.40	50.00	66.70	180.71

Tabla 13: Tabla con los resultados del algoritmo memético aplicando búsqueda local al mejor 10 % de la población.

6.3. Resultados globales

Finalmente, los resultados globales obtenidos son:

	Ionosphere				Parkinsons				Spectf-heart			
	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T	%_clas	%_red	Agr.	T
1-NN	86.89	0.00	43.45	0.02	96.41	0.00	48.21	0.00	86.25	0.00	43.12	0.02
RELIEF	72.60	2.94	37.77	0.02	96.41	0.00	48.21	0.01	83.68	0.00	41.84	0.02
BL	88.88	82.35	85.62	27.32	90.77	72.73	81.75	3.67	82.23	73.18	77.70	36.50
ES	91.16	88.82	89.99	261.22	92.31	90.91	91.61	77.46	82.52	92.73	87.62	268.90
BMB	93.16	82.35	87.76	269.21	90.77	88.18	89.48	74.58	83.66	82.27	82.97	285.82
ILS	84.89	89.41	87.15	146.61	85.64	85.45	85.55	29.82	85.95	92.73	89.34	172.14
ILS-ES	88.31	90.00	89.16	172.62	89.74	90.00	89.87	50.69	84.24	91.82	88.03	176.68
AGG-BLX	88.88	87.65	88.26	172.58	87.69	84.55	86.12	51.70	81.38	79.55	80.46	180.23
AGG-CA	86.60	87.65	87.12	172.90	89.74	89.09	89.42	51.00	84.24	82.73	83.48	180.16
AGE-BLX	88.32	81.76	85.04	172.36	90.26	74.55	82.40	51.22	83.66	78.18	80.92	178.96
AGE-CA	88.60	88.82	88.71	170.91	86.15	81.82	83.99	51.26	83.67	89.55	86.61	176.25
AM-(10, 1.0)	86.61	87.06	86.84	174.83	91.28	87.27	89.28	51.91	85.66	74.55	80.11	182.01
AM-(10, 0.1)	87.18	91.18	89.18	164.92	89.74	89.09	89.42	49.45	85.39	87.73	86.56	175.95
AM-(10, 0.1mej)	87.47	70.00	78.73	169.39	84.62	87.27	85.94	49.45	83.40	50.00	66.70	180.71

Tabla 14: Tabla con los resultados globales en el problema APC.

6.4. Análisis de los resultados

Todos los algoritmos en esta práctica suponen una mejora frente a la búsqueda local, aunque requieren un mayor tiempo de ejecución, principalmente debido a que la búsqueda local termina mucho antes de llegar al límite de evaluaciones.

El enfriamiento simulado, aunque es el algoritmo más complejo de todos, es el que suele obtener mejores resultados, teniendo un tiempo de ejecución similar a *BMB*, aunque superior a *ILS*.

El algoritmo *BMB*, aunque suele ser el peor de los estudiados en esta práctica, sigue suponiendo normalmente una mejora frente a la búsqueda local, lo cuál demuestra la importancia de la solución inicial en dicho algoritmo. Como siempre utilizamos la misma semilla, la solución inicial en la primera iteración siempre coincide con la de la búsqueda local, y es debido a esto que los resultados son iguales o mejores siempre que la búsqueda local termine en menos de las 1000 evaluaciones que realiza *BMB* para cada solución.

Habría que esperar que el tiempo de ejecución de *BMB* estuviese entre 10 y 15 veces el de la búsqueda local, pero debido a la semilla que utilizamos en el caso de *parkinsons.arff*, la mayoría de ejecuciones de *BMB* llega a las 1000 iteraciones, mientras que en la búsqueda local hay varias particiones en las que termina mucho antes, y por ello el tiempo es superior al esperado.

El algoritmo *ILS*, aunque no proporciona resultados tan buenos como *ES*, supone una gran mejora en tiempo, teniendo resultados muy cercanos. Por lo que, si es importante el tiempo de ejecución a la vez que la calidad de los resultados, es un algoritmo a tener en cuenta. Al igual que en el caso *BMB*, la primera ejecución en la primera iteración coincide con la búsqueda local, por lo que los resultados siempre serán iguales o mejores (siempre que la búsqueda local termine en 1000 o menos iteraciones).

La mejora en tiempo frente a *BMB* se debe a que, a diferencia de tomar soluciones iniciales aleatorias que provocan que en la mayoría de los casos lleguemos a las 1000 iteraciones, al tomar solo una mutación de una solución previa, ya estamos cerca de un máximo, por lo que muchas iteraciones de la búsqueda local se detienen antes de llegar a las 1000 iteraciones.

Cabría esperar que *ILS* proporcionara mejores soluciones que *BMB*, sin embargo, debido a que como tenemos un espacio de soluciones infinito, las mutaciones que estudiamos en *ILS* son a partir de una distribución normal con una varianza ligeramente superior a la de la búsqueda local, por lo que es muy probable que estas mutaciones sigan “atascadas” en el mismo máximo local. Sin embargo, si tomásemos semillas distintas o un problema con un número finito de soluciones, es probable que fuesen mejores.

El algoritmo *ILS-ES* es un punto medio entre *ILS* y *ES*: los resultados son mejores que *ILS* aunque suelen ser peores que *ES*, mientras que el tiempo es superior a *ILS* pero inferior a *ES*.

Finalmente, comparando con los algoritmos genéticos, *ES* y *ILS-ES* proporcionan mejores resultados, siendo el tiempo de *ILS-ES* similar y el de *ES* ligeramente superior. Los algoritmos meméticos son los únicos que pueden competir en calidad de resultados con *ES*, aunque sólo en algunos casos.

Debido a esto, de todos los algoritmos estudiados, si lo que nos interesa es la calidad de las soluciones para este problema, el mejor algoritmo es *ES* o *ILS-ES*. Si lo que nos interesa es obtener resultados decentes en poco tiempo, dependiendo de la situación será mejor elegir entre *RELIEF* o *localSearch*, dependiendo de que tan importante sea el tiempo con respecto a la calidad. Por último, si queremos un punto medio, es decir, obtener buenas soluciones sin un tiempo demasiado alto de ejecución, el mejor candidato es *ILS-ES*.