Universidad de Granada

Práctica 2.b

Técnicas de Búsquedad basadas en Poblaciones 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

19 de mayo de 2022

Índice

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:

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

donde n es el número de características de los datos, y $W = (w_1, \ldots, w_n)$ 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:

$$\text{Agregación}(W) = \alpha \cdot \text{Tasa\_clas}(W) + (1 - \alpha) \cdot \text{Tasa\_red}(W) \tag{1}$$

donde:

$$\text{Tasa\_clas} = 100 \cdot \frac{\text{nº de instancias bien clasificadas en T}}{\text{nº de instancias en T}} \tag{2}$$

$$\text{Tasa\_red} = 100 \cdot \frac{\text{nº de valores } w_i < 0{,}1}{\text{nº de características}} \tag{3}$$

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 $y = (y_1, \ldots, y_k)$, 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, $W = (w_1, \ldots, w_n)$, 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.
$dist \leftarrow 0$;
for $i = 0$ ***to*** $length(P)$ **do**
  $dist \leftarrow dist + (P[i] - Q[i])^2$;
end
$output \leftarrow sqrt(dist)$;
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.
$dist \leftarrow 0$;
for $i = 0$ ***to*** $length(W)$ **do**
  if $W[i] > 0{,}1$ **then**
    $dist \leftarrow dist + W[i](P[i] - Q[i])^2$;
  end
end
$output \leftarrow sqrt(dist)$;
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 ← {} ;
for x in X_test do
    min_dist ← length(x) ;
    neighbour ← 0 ;
    for i = 0 to length(X) do
        dist ← weighted_euclidean(x, X[i], W) ;
        if dist < min_dist then
            min_dist ← dist ;
            neightbour ← i ;
        end
    end
    Y_pred ← Y_pred ∪ {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 ← {} ;
for x in X do
    min_dist ← length(x) ;
    neighbour ← 0 ;
    for i = 0 to length(X) do
        if x ≠ X[i] then
            dist ← weighted_euclidean(x, X[i], W) ;
            if dist < min_dist then
                min_dist ← dist ;
                neightbour ← i ;
            end
        end
    end
    Y_pred ← Y_pred ∪ {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 ← {} ;
X_test ← {} ;
Y_train ← {} ;
Y_test ← {} ;
size ← length(X)/n ;
remain ← length(X) %n ;
if k < remain then
    begin ← (size + 1) ;
    end ← begin + (size + 1) ;
else if k == remain then
    begin ← (size + 1) ;
    end ← begin + size ;
else
    begin ← size ;
    end ← begin + size ;
end
for i = 0 to length(X) do
    if i < begin or i > begin then
        X_train ← X_train ∪ X[i] ;
        Y_train ← Y_train ∪ Y[i] ;
    else
        X_test ← X_test ∪ X[i] ;
        Y_test ← Y_test ∪ 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 una solución.

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

$w \leftarrow \{\}$;
for i ***in*** $\{0, \ldots, size\}$ **do**
  $w \leftarrow w \cup \text{generator}()$;
end
return w ;

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 cromosomas 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 cromosomas a obtener $size$.
Output: Una matriz con los cromosomas a cruzar.

$parents \leftarrow \{\}$;
for i ***in*** $\{0, \ldots, size\}$ **do**
  $random1 \leftarrow \text{rand}() \,\%length(solutions)$;
  $random2 \leftarrow \text{rand}() \,\%length(solutions)$;
  if $fitness[random1] > fitness[random2]$ **then**
    $parents \leftarrow parents \cup solutions[random1]$;
  else
    $parents \leftarrow parents \cup solutions[random2]$;
  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 cromosomas utilizando Blx.

Input: Una matriz con los cromosomas *solutions*.
Input: Un generador de números aleatorios con distribución uniforme entre 0 y 1, *generator*.
Output: Una matriz con los cromosomas tras el cruce.

```
newSolutions ← {} ;
crosses ← 0,7 · length(solutions)/2 ;
for i in {0, ..., crosses} do
    parent1 ← 2 · i ;
    parent2 ← (2 · i + 1) %length(solutions) ;
    for j in {0, 1} do
        w ← {} ;
        for k in length(solutions[parent1]) do
            if solutions[parent1][k] < solutions[parent2][k] then
                min ← solutions[parent1][k] ;
                max ← solutions[parent2][k] ;
            else
                min ← solutions[parent2][k] ;
                max ← solutions[parent1][k] ;
            end
            min ← min + 0,3 · (max − min) ;
            max ← max + 0,3 · (max − min) ;
            gene ← min + generator() · (max − min) ;
            if gene < 0 then
                gene ← 0 ;
            else if gene > 1 then
                gene ← 1 ;
            end
            w ← w ∪ gene ;
        end
        newSolutions ← newSolutions ∪ w ;
    end
end
for i in {2 · crosses, ..., length(solutions)} do
    newSolutions ← newSolutions ∪ solutions[i] ;
end
return newSolutions ;
```

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 cromosomas utilizando cruce aritmético.

Input: Una matriz con los cromosomas *solutions*.
Input: Un generador de números aleatorios con distribución uniforme entre 0 y 1, *generator*.
Output: Una matriz con los cromosomas tras el cruce.

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

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 $\{0, \ldots, n\}$ 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 cromosomas *solutions*.
Input: Un generador de números aleatorios con distribución normal de media 0 y varianza $0{,}3^2$, *generator*.
Output: Una matriz con los cromosomas mutados.

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

3. Descripción de los algoritmos

3.1. Algoritmos genéticos

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

Algorithm 14: GENETIC_ALGORITHM algoritmo genético.

Input: Una matriz con los caracterísstica 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 $0{,}3^2$. ;
$newSolutions \leftarrow \{\}$;
for i ***in*** $\{0, \ldots, length(solutions)\}$ **do**
    $fitness \leftarrow fitness \cup$ agregation$(solutions[i])$;
    $eval \leftarrow eval + 1$;
end
while $eval < iterations$ **do**
    $size \leftarrow 2$ `// En el caso estacionario`
    $size \leftarrow length(solutions)$ `// En el caso generacional`
    $newSolutions \leftarrow$ selection$(solutions, fitness, size)$;
    $newSolutions \leftarrow$ cross$(newSolutions, uniform)$;
    $newSolutions \leftarrow$ mutation$(newSolutions, normal)$;
    $solutions \leftarrow$ replace$(solutions, newSolutions, fitness, eval)$;
end
$ind \leftarrow 0$;
$bestFitness \leftarrow fitness[0]$;
for i ***in*** $\{1, \ldots, length(fitness)\}$ **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 15: REPLACE algoritmo generacional elitista utilizado para reemplazar las soluciones.

Input: Una matriz con los cromosomas de la generación previa $solutions$.
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*** $\{1, \ldots, length(fitness)\}$ **do**
    if $fitness[i] > bestFitness$ **then**
        $bestSolution \leftarrow w[i]$;
        $bestFitness \leftarrow fitness[i]$;
    end
end
$solutions \leftarrow w$;
for i ***in*** $\{0, \ldots, length(solutions)\}$ **do**
    $fitness \leftarrow fitness \cup \text{agregation}(solutions[i])$;
    $eval \leftarrow eval + 1$;
end
$ind \leftarrow 0$;
$worstFitness \leftarrow fitness[0]$;
for i ***in*** $\{1, \ldots, length(fitness)\}$ **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 16: REPLACE algoritmo estacionario utilizado para reemplazar las soluciones.

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

```
agregation1 ← agregation(w[0]) ;
agregation2 ← agregation(w[1]) ;
eval ← eval + 2 ;
minPos1 ← 0 ;
minPos2 ← 1 ;
if fitness[minPos2] < fitness[minPos1] then
    minPos1 ← 1 ;
    minPos2 ← 0 ;
end
for i in {2, ..., length(fitness)} do
    if fitness[i] < fitness[minPos1] then
        minPos2 ← minPos1 ;
        minPos1 ← i ;
    else if fitness[i] < fitness[minPos2] then
        minPos2 ← i ;
    end
end
if agregation1 > agregation2 then
    if agregation1 > fitness[minPos2] then
        solutions[minPos2] ← w[0] ;
        fitness[minPos2] ← agregation1 ;
        if agregation2 > fitness[minPos1] then
            solutions[minPos1] ← w[1] ;
            fitness[minPos1] ← agregation2 ;
        end
    else if agregation1 > fitness[minPos1] then
        solutions[minPos1] ← w[0] ;
        fitness[minPos1] ← agregation1 ;
    end
else
    if agregation2 > fitness[minPos2] then
        solutions[minPos2] ← w[1] ;
        fitness[minPos2] ← agregation2 ;
        if agregation1 > fitness[minPos1] then
            solutions[minPos1] ← w[0] ;
            fitness[minPos1] ← agregation1 ;
        end
    else if agregation2 > fitness[minPos1] then
        solutions[minPos1] ← w[1] ;
        fitness[minPos1] ← 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.

3.2. Algoritmos meméticos

Utilizamos el algoritmo generacional utilizando cruce arimético. 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 17: MEMETIC_ALGORITHM algoritmo genético.

Input: Una matriz con los caracterísstticas 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 ← 0 ;
generation ← 0 ;
solutions ← initialize(chromosomes, length(w[0])) ;
fitness ← {} ;
uniform ← Un generador de distribución uniforme entre 0 y 1. ;
normal ← Un generador de distribución normal de media 0 y varianza 0,3^2. ;
newSolutions ← {} ;
for i in {0, ..., length(solutions)} do
    fitness ← fitness ∪ agregation(solutions[i]) ;
    eval ← eval + 1 ;
end
while eval < iterations do
    generation ← generation + 1 ;
    size ← length(solutions) ;
    newSolutions ← selection(solutions, fitness, size) ;
    newSolutions ← cross(newSolutions, uniform) ;
    newSolutions ← mutation(newSolutions, normal) ;
    if generation %generationStep == 0 then
        // Algoritmo memético
        memeticSize ← length(crossParents) × size ;
        if sort then
            ind ← sortInd(fitness, memeticSize) ;
        else
            ind ← {0, ..., memeticSize} ;
        end
        for i in {0, ..., memeticSize} do
            crossParents[ind[i]] ← localSearch(crossParents[ind[i]], 2 × length(crossParents[ind[i]]), eval) ;
        end
    end
    solutions ← replace(solutions, newSolutions, fitness, eval) ;
end
ind ← 0 ;
bestFitness ← fitness[0] ;
for i in {1, ..., length(fitness)} do
    if fitness[i] > bestFitness then
        ind ← i ;
        bestFitness ← 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 simplicitud, 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.

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.
$W \leftarrow \{0, 0, \ldots, 0\}$ `// Tantos ceros como carcaterísticas tengan los datos.`
for $i = 0$ ***to*** $length(X)$ **do**
  $ind\_enemy \leftarrow i$;
  $ind\_friend \leftarrow i$;
  $dist\_enemy \leftarrow length(X[0])$ `// Al normalizar los datos, la distancia máxima`
  $dist\_friend \leftarrow length(X[0])$ `// es` $\sqrt{k} < k$`, siendo` k `el número de características`
  for $j = 0$ ***to*** $length(X)$ **do**
    if $i \neq j$ **then**
      if $X[i] == X[j]$ ***and*** weighted_euclidean$(X[i], X[j], W) < dist_friend$ **then**
        $ind\_friend \leftarrow j$;
        $dist\_friend \leftarrow$ weighted_euclidean$(X[i], X[j], W)$;
      else if $X[i] \neq X[j]$ ***and*** weighted_euclidean$(X[i], X[j], W) < dist_enemy$ **then**
        $ind\_enemy \leftarrow j$;
        $dist\_enemy \leftarrow$ weighted_euclidean$(X[i], X[j], W)$;
      end
    end
  end
  for $j = 0$ ***to*** $length(X[0])$ **do**
    $w[j] \leftarrow |X[i][j] - x[ind\_enemy][j]| - |X[i][j] - X[ind\_friend][j]|$;
  end
end
$W \leftarrow$ 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.
$perm \leftarrow \{0, \ldots, n-1\}$;
$perm \leftarrow \text{shuffle}(perm)$;
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.
Output: Un vector de pesos.
$iteration \leftarrow 0$;
$iteration\_mod \leftarrow 0$;
$W \leftarrow$ random_sol($length(X[0])$) ;
while $iteration < 15000$ ***and*** $iteration\_mod < 20$ **do**
    $modified \leftarrow false$;
    $perm \leftarrow$ 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 ;
        $neighbour \leftarrow$ 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 agregation($neighbour$) > 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 ;

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).
- **iomanip**: 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.
- **REDUCCTION**: Contiene el umbral por debajo del cual los pesos se consideran como 0. Por defecto, 0.1.
- **CHROMOSOMES**: Contiene el número de cromosomas (soluciones) que utilizan los algoritmos genéticos. Por defecto, 30.
- **ITERATIONS**: Número de iteraciones que realiza el algoritmo. Cada iteración se considera como una evaluación de una solución. Es decir, en el caso generacional, en el que en cada generación generamos tantas soluciones como cromosomas, cuenta como el número de cromosomas en iteraciones. Por defecto, 15000.
- **ALPHA**: Variable utilizada para el cruce Blx. Determina el tamaño del intervalo al cuál pueden pertenecer las soluciones. Por defecto, 0'3.
- **CROSS**: Probabilidad de cruce. Determina la probabilidad de que los hijos sean cruce de los padres o directamente los propios padres. Por defecto, 0'7 en el caso generacional, y 1 en el caso estacionario.
- **MUTATION**: Probabilidad de mutación. Determina la probabilidad de que muten los genes de los hijos. Por defecto 0'1.

- **OUTPUT**: Salida extendida por pantalla. En caso de ser "true", imprimirá el vector de pesos de cada iteración y el número de elementos que toma para training y test. Por defecto, "true".
- **LATEX**: Modifica la salida para que el formato sea más sencillo de importar en las tablas de latex. Por defecto, "false".
- **EXPORT**: Exporta los datos como coordenadas a la carpeta *graph/data* para poder representar los resultados en gráficas. Por defecto, "false".

Además, para las distintas variantes del algoritmo memético tenemos las constantes:

- **STEP**: Número de iteraciones tras las cuales aplicamos la búsqueda local. Por defecto, 10.

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:

- **aggBlx**: Implementación del algoritmo genético generacional con cruce Blx.
- **aggCa**: Implementación del algoritmo genético generacional con cruce aritmético.
- **ageBlx**: Implementación del algoritmo genético estacionario con cruce Blx.
- **ageCa**: Implementación del algoritmo genético estacionario con cruce aritmético.
- **am**: Implementación del algoritmo memético.

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 *aggBlx* con los datos de *ionosphere.arff*, sería:

```
bin/aggBlx 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 Montery.
- Procesador: 1,6 GHz Intel Core i5 de doble núcleo.
- Memoria: 16 GB 2133 MHz LPDDR3.
- Gráficos: Intel UHD Graphics 617 1536 MB.

Además, se han realizado con la constante *OUTPUT* en *false*, *LATEX* en *true* y *EXPORT* en false, distintas de las usado por defecto. Si se modifican, cambiará la aleatoriedad proporcionada por la semilla, por lo que los resultados serán ligeramente distintos.

6.1. Resultados

En todos los resultados, el tiempo es en segundos.

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 1: 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	90.14	82.35	86.25	19.12	84.62	81.82	83.22	4.90	78.57	79.55	79.06	39.72
Partición 2	82.86	88.24	85.55	36.62	89.74	72.73	81.24	3.67	92.86	84.09	88.47	48.57
Partición 3	84.29	52.94	68.61	16.93	79.49	81.82	80.65	5.30	84.29	81.82	83.05	43.31
Partición 4	85.71	85.29	85.50	32.01	87.18	72.73	79.95	2.89	90.00	70.45	80.23	36.66
Partición 5	84.29	91.18	87.73	49.26	87.18	81.82	84.50	5.00	82.61	86.36	84.49	45.37
Media	85.46	80.00	82.73	30.79	85.64	78.18	81.91	4.35	85.66	80.45	83.06	42.73

Tabla 2: 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 3: 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 4: 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 5: 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 6: 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 7: 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 8: 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 9: Tabla con los resultados del algoritmo memético aplicando búsqueda local al mejor 10 % de la población.

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	85.46	80.00	82.73	30.79	85.64	78.18	81.91	4.35	85.66	80.45	83.06	42.73
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 10: Tabla con los resultados globales en el problema APC.

6.2. Gráficas de los algoritmos

En esta sección añado todas las gráficas obtenidas.

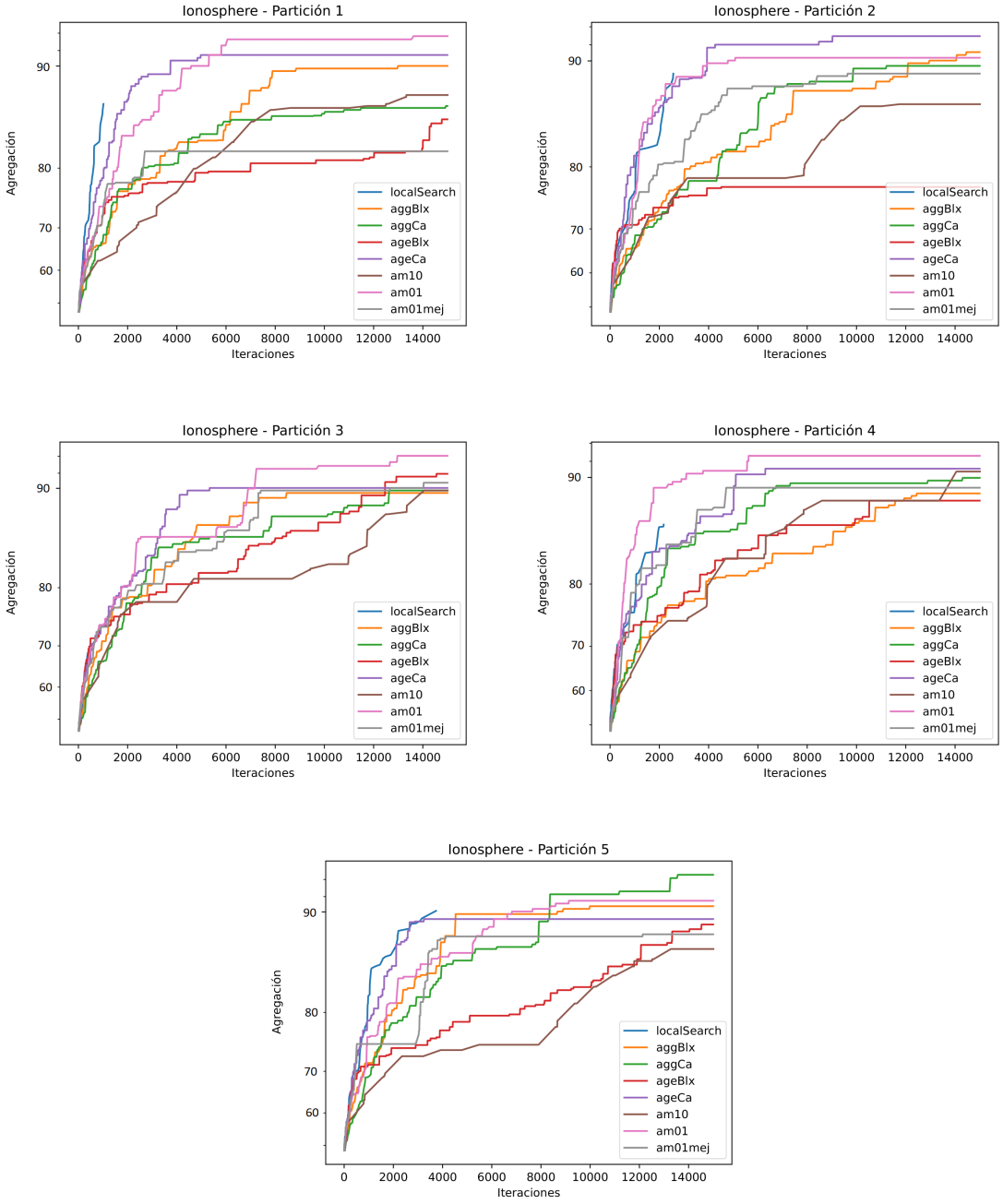

Figura 1: Gráficas de la agregación respecto al número de iteraciones con los datos de train de Ionosphere

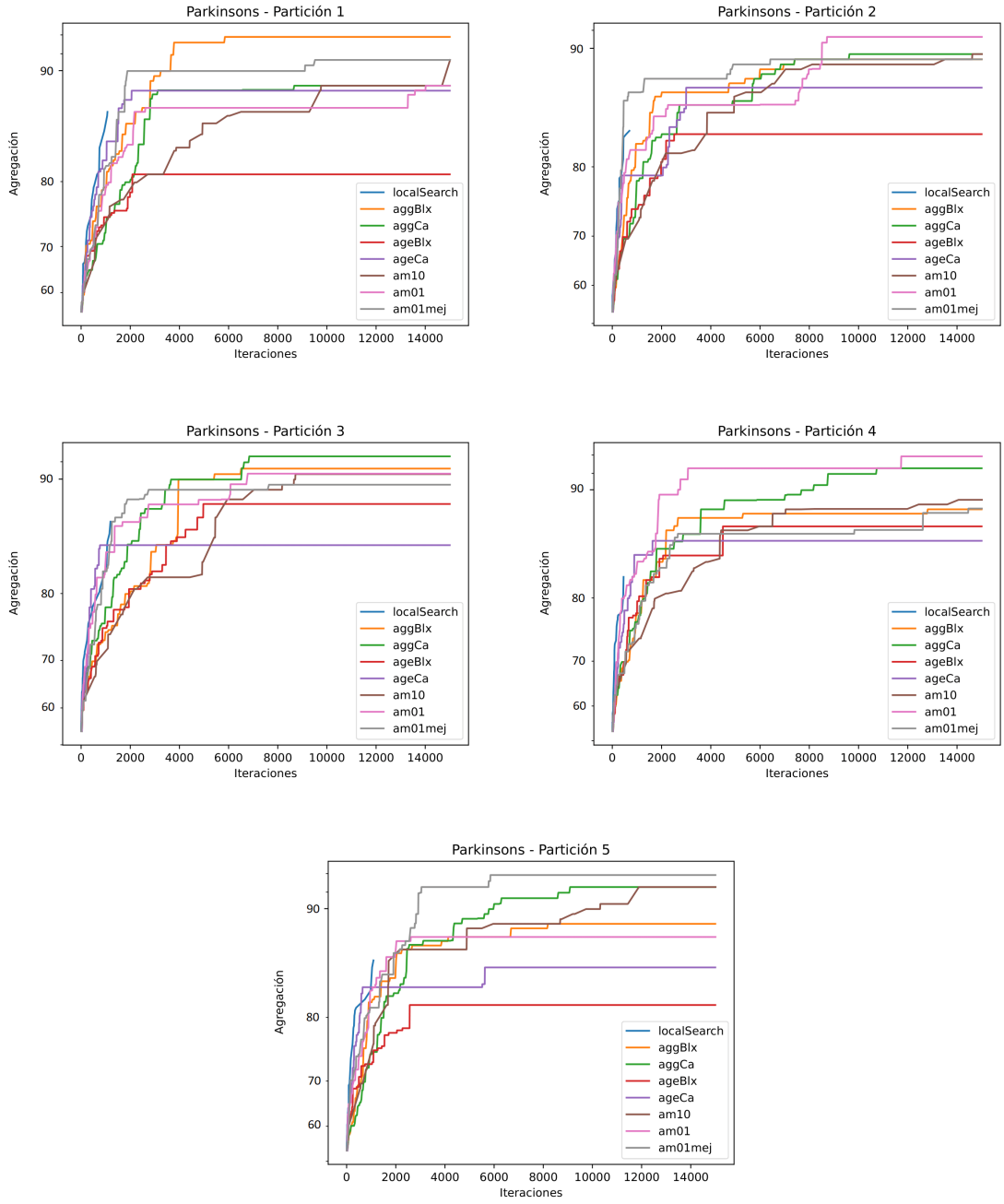

Figura 2: Gráficas de la agregación respecto al número de iteraciones con los datos de train de Parkinsons

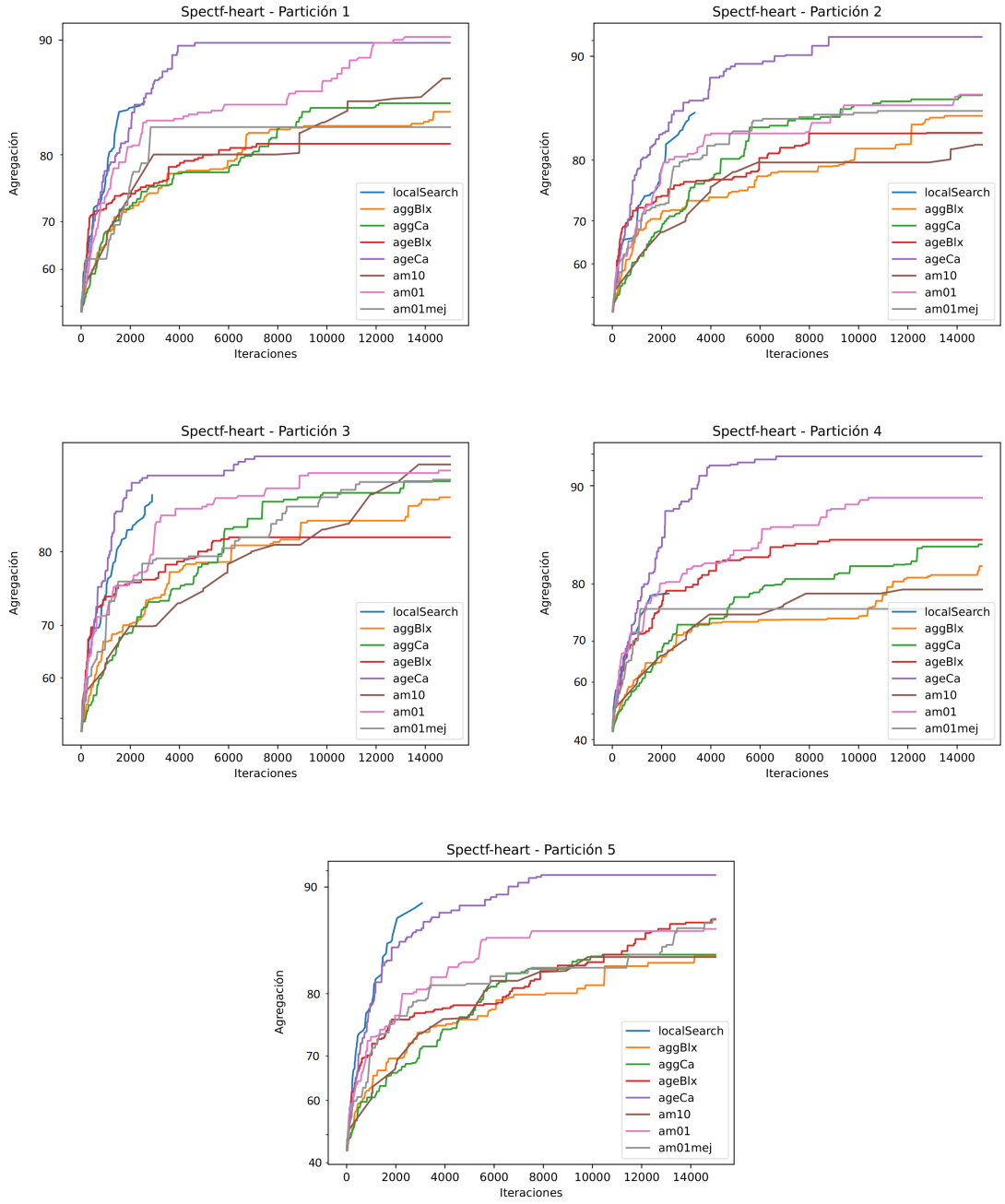

Figura 3: Gráficas de la agregación respecto al número de iteraciones con los datos de train de Spectf-heart

Hay que tener en cuenta que estas gráficas son con los datos de *train*, mientras que en los resultados de las tablas previas la clasificación es de los datos de *test*, por lo que aunque en alguna gráfica un algoritmo pueda parecer mejor, no tiene por qué serlo.

En todas las gráficas podemos observar a qué se debe que el algoritmo *localSearch* tenga un tiempo mucho inferior: se detiene mucho antes que los demás.

En todos los algoritmos, la agregación sólo aumenta. Esto se debe a que representamos sólo la agregación de la mejor solución, y en todos los algoritmos, si en una nueva generación se genera una solución peor, no reemplaza a ninguna otra, por lo que la agregación siempre aumenta con el número de iteraciones.

Todos los algoritmos tienen comportamientos similares, aumentando mucho la agregación en las primeras iteraciones (ya que como parten de soluciones aleatorias, es fácil realizar mejores), y mejorando

cada vez menos (ya que es más difícil mejorar la solución). Además, todas empiezan con una agregación entre el 40 % y el 50 %. Eso se debe a que, aunque sean soluciones aleatorias, siguen siendo una modificación del algoritmo del vecino más cercano, por lo que obtenemos resultados decentes aunque sea una mala modificación.

6.3. Análisis de los resultados

Todos los algoritmos genéticos y meméticos dan mejores resultados que el algoritmo *1-NN* y *RELIEF*, principalmente debido al aumento de la tasa de reducción, aunque es a costa de un aumento en el tiempo de ejecución.

De entre los algoritmos genéticos, los generacionales suelen proporcionan mejores resultados, principalmente aumentando la tasa de reducción, sin producir un aumento en el tiempo de ejecución. Aunque el intervalo de exploración del cruce aritmético está contenido dentro del intervalo del cruce Blx, el cruce aritmético suele dar mejores resultados ya que hay infinitas posibles soluciones, y por lo cual no las exploramos todas. Además, es más probable que una solución mejor entre dos previas ya que, por algo las considerábamos como posibles soluciones, que más alejado de ellas, por eso suele producir mejores resultados el cruce aritmético. Debido a esto, el mémetico utilizará el algoritmo genético generacional con cruce aritmético como base, el cuál mejoramos añadiendo la búsqueda local.

El algoritmo memético es el que llega a producir las mejores soluciones, aunque depende de a que conjunto de soluciones le apliquemos la búsqueda local. Otra cosa a tener en cuenta es que sólo utilizamos 10 cromosomas en el algoritmo genético frente a las 30 del generacional, lo cuál nos permite una mayor exploración de cada uno (ya que le corresponden más iteraciones), pero tenemos menos cromosomas iniciales.

Además podemos ver que es mejor aplicar la búsqueda local en el algoritmo memético a soluciones aleatorias que a las mejores. Esto se debe a que estas soluciones mejores estarán cerca de un mínimo local, por lo que la búsqueda local se queda "atascada" y no nos proporciona mejoras, mientras que aplicarle la búsqueda local a soluciones peores permite que realicemos una mayor exploración "saliendo" de estos mínimos locales.

La mayoría de resultados del algoritmo memético con un 10 % aleatorio de la población son mejores que las del genético con cruce aritmético, en el cuál está basado. Esto nos permite ver la importancia de la búsqueda local para encontrar mejores soluciones.

Por último, comparando con la búsqueda local, podemos ver que todos los algoritmos genéticos y meméticos suponen un aumento considerable en el tiempo de ejecución. Esto se debe principalmente a que en estos algoritmos siempre realizamos las 15000 ejecuciones, mientras que en la búsqueda local terminábamos mucho antes al quedarnos atascados en un mínimo local. En caso de que eliminásemos la condición de salir si la solución no mejora en la búsqueda local, los tiempos son similares, por lo que la mayor parte del tiempo de ejecución se debe al calculo de la agregación de la solución. Pero a cambio, los resultados son mejores, por lo que si damos mayor prioridad a la calidad de la solución que al tiempo de ejecución, todos los algoritmos estudiados en esta práctica suponen una mejora.