UNIVERSIDAD DE GRANADA

Metaheurísticas Problema del Aprendizaje de Pesos en Características

Práctica 2: Técnicas de Búsqueda basadas en Poblaciones
Algoritmos Genéticos y Meméticos

Miguel Ángel Fernández Alonso
mfernandez1@correo.ugr.es
76065289-B
Grupo 3 - Jueves 17:30h

Grado en Ingeniería Informática. Tercero.
Curso 2023-2024

Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

I. Descripción del Problema

El problema del APC consisnte en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos.

La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así se puede formular como:

$MaximizarF(W) = \alpha tasaclas(W) + (1-\alpha) tasared(W)$

Partimos de un vector de datos $T = t_1, ..., t_n$ donde cada dato contiene un conjunto de características o *traits* $x_1, ..., x_n$.

El problema se reduce en calcular un vector de pesos

$$W = w_1, ..., w_n$$

donde el peso

$$w_i \in [0, 1]$$

pondera la característica

$$x_i$$

A este valor lo denominaremos vector de entrenamiento o aprendizaje y nos permitirá clasificar otro conjunto de datos desconocido al que llamaremos vector de test o validación.

Para aumentar la fiabilidad del proceso usaremos el método *5-fold cross validation* que consiste en crear cinco particiones distintas de datos repartidos equitativamente según su clase. Las particiones las dividiremos de tal forma que dediquemos una a *test* y cuatro a *train*.

Para clasificar los datos, implementamos el algoritmo **k-NN** en su versión con *k = 1*. Este algoritmo asigna a cada dato la clase su vecino más cercano calculado a partir de la *distancia euclídea*.

Calcularemos el vector de pesos con los diferentes algoritmos y usaremos nuestro clasificador **k-NN** para valorar el rendimiento de los resultados. Además es necesario calcular los diferentes porcentajes para cada conjunto de datos valorados. [1] [2]

Definiremos:

$tasaclas = 100 \frac{ninstanciasbienclasificadasenT}{ninstanciasenT}$

$tasred = 100 \frac{nvaloresw_i<0{,}2}{ncaracteristicas}$

Para proceder a los cálculos debemos partir de unos datos normalizados. En la primera prácitica, implementamos un algoritmo **Greedy Relief** y uno de **Búsqueda Local** y en la segunda hemos desarrollado cuatro algoritmos **Genéticos** y tres **Meméticos**. Los algoritmos son ejecutados sobre un conjunto de datos que contienen los archivos *ecoli, parkinsons, breastcancer* todos ellos en el formato *.csv*.

II. Descripción de los Algoritmos empleados en el Problema

Nuestro esquema de representación de la solución, será un vector de pesos W que nos servirá para valorar la bondad de los datos en clasificaciones futuras. El vector será del tipo $W = w_1, ..., w_n$ donde cada

$$w_i \in [0, 1]$$

y cada w_i gradúa el peso asociado a cada característica y pondera su importancia.

Nuestro algoritmo **k-NN** medirá la bondad de los pesos calculados mediante cada algoritmo, recibiendo como parámetros un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculado por el algoritmo seleccionado. El clasificador devolverá un struct *Resultados* que contiene las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo que ha tardado en ejecutarse.

La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. A mayor tasa de clasificación, mejor será el vector de pesos W generado por nuestro algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto al total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

Algorithm 1 KNN_LOO

```
Require: train, test, w
Ensure: tasa_clas, tasa_red
aciertos ← 0
num_w_menor ← 0
for i = 0 to tamaño_test do
    pos ← nearestNeighbour_LOO(train, test[i], w)
    if train[pos].clase = test[i].clase then
        aciertos ← aciertos + 1
    end if
end for
for i = 0 to tamaño_w do
    if w[i] <0.2 then
        num_w_menor ← num_w_menor + 1
    end if
end for
tasa_clas ← 100.0 * (aciertos / tamaño_test)
tasa_red ← 100.0 * (num_w_menor / tamaño_w)
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que estemos valorando. Además, ha sido programado aplicando leave-one-out, ya que cuando se utiliza en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto, porque estamos aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo dato que estamos comparando.

Finalmente utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de características teniendo en cuenta los pesos cuyo valor supere 0.2.

Además, he añadido una versión de la función **k-NN** que utiliza una modalidad del nearestNeighbour sin el criterio de

Algorithm 2 nearestNeighbour_LOO

Require: train, actual, w
Ensure: pos
1: mejor_distancia ← n_traits_primera_partición_de_train + 1
2: **for** i = 0 to tamaño_train - 1 **do**
{leave-one-out}**if** actual ≠ train[i] **then** {.t accede al vector de características}
3: distancia_actual ← euclideanDistance(train[i].t, actual.t, w)
5: **if** distancia_actual < mejor_distancia **then**
6: mejor_distancia ← distancia_actual
7: pos ← i
8: **end if**
9: **end if**
10: **end for**

Algorithm 3 euclideanDistance

Require: v1, v2, w
Ensure: dist
1: dist ← 0
2: **for** i = 0 to tamaño_v1 **do**
3: **if** w[i] ≥ 0.2 **then**
4: dist ← dist + w[i] * (v2[i] - v1[i]) * (v2[i] - v1[i])
5: **end if**
6: **end for**

Leave-One-Out. Esta versión del algoritmo será usada cuando apliquemos el **k-NN** sobre dos conjuntos de datos diferentes.

Algorithm 4 nearestNeighbour

Require: train, actual, w
Ensure: pos
1: mejor_distancia ← n_traits_primera_partición_de_train + 1
2: **for** i = 0 to tamaño_train - 1 **do**
3: distancia_actual ← euclideanDistance(train[i].t, actual.t, w)
4: **if** distancia_actual < mejor_distancia **then**
5: mejor_distancia ← distancia_actual
6: pos ← i
7: **end if**
8: **end for**

En los algoritmos genéticos y meméticos he creado un struct *Cromosoma* que contiene la información básica de cada elemento con el que vamos a trabajar (la lista de características y su valor de evaluación).

Listing 1. Definición de la estructura Cromosoma

```
struct Cromosoma {
    vector<double> w;
    double pts;
};
```

Para seleccionar el conjunto de cromosomas padres de la población, usaremos la función *binaryTournament*, la cual genera dos números aleatorios, dentro de los índices de cada población de cromosomas y se queda con el que tenga una mejor puntuación de evaluación de los dos.

Algorithm 5 binaryTournament

Require: poblacion, generador
Ensure: max
1: max ← 0
2: distribucion ← random_int(0, poblacion.size()-1)
3: num1 ← distribucion(generador)
4: num2 ← distribucion(generador)
5: **while** num1 = num2 **do**
6: num2 ← distribucion(generador)
7: **end while**
8: **if** poblacion[num1].pts >poblacion[num2].pts **then**
9: max ← num1
10: **else**
11: max ← num2
12: **end if**
13: **return** max

Aplicando el torneo binario, obtendremos los padres de cada población que serán cruzados entre ellos para obtener los hijos de la nueva población. Usaremos dos operadores de cruce: *cruceBLX-alhpa* y *cruceAritmetico*.

El cruce **BLX-alpha** con alpha = 0.3, genera una pareja de descendientes a partir de dos padres, asignando a cada componente w[i] de cada hijo un valor aleatorio dentro del rango: $(w_{min}(w_{max}w_{min})alpha, (w_{max} + ((w_{max}w_{min})alpha)$ y truncando su valor entre 0 y 1.

Algorithm 6 cruceBLX

Require: padre1, padre2, generador
Ensure: pair(hijo1, hijo2)
1: crear hijo1
2: crear hijo2
3: **for** i = 0 to tamaño_caracteristicas_padre1 **do**
4: max ← maximo(padre1.w[i], padre2.w[i])
5: min ← minimo(padre1.w[i], padre2.w[i])
6: diff ← max - min
7: distribucion ← random(min - diff * ALPHA_AGG, max + diff * ALPHA_AGG)
8: aux ← distribucion(generador)
9: truncar aux
10: hijo1.w[i] ← aux
11: aux ← distribucion(generador)
12: truncar aux
13: hijo2.w[i] ← aux
14: **end for**
15: **return** pair(hijo1, hijo2)

El cruce **Aritmético**, genera solo un descendiente que tiene como valor de cada componente w[i] del vector de características, la media aritmética del valor de w[i] de cada padre.

Para realizar las mutaciones oportunas en los algoritmos genéticos y meméticos, realizaremos en cada generación el número de mutaciones esperadas sobre la población de hijos.

Algorithm 7 cruceArit

Require: padre1, padre2
Ensure: hijo
1: crear hijo
2: **for** i = 0 to tamaño_caracteristicas_padre1 **do**
3: hijo.w[i] ← (padre1.w[i] + padre2.w[i]) / 2.0
4: **end for**
5: **return** hijo

Elegiremos un cromosoma y un gen aleatorios de la población y le aplicaremos una mutación sumándole un valor de una distribución normal con alpha = 0.3 y finalmente truncando su valor.

Algorithm 8 Bucle de mutación

Require: num_mutaciones, generador, hijos, tamaño_caracteristicas
1: **for** i = 0 to num_mutaciones **do**
2: valor_mut ← random_mutaciones(generador)
3: cromosoma_mutar ← valor_mut / tamaño_caracteristicas
4: gen_mutar ← valor_mut % tamaño_caracteristicas
5: aux ← hijos[cromosoma_mutar].w[gen_mutar] + normal(generador)
6: truncar aux
7: hijos[cromosoma_mutar].w[gen_mutar] ← aux
8: **end for**

La búsqueda local implementada en los algoritmos meméticos es similar a la utilizada en la práctica 1, salvo el criterio de parada e inicialización.

Además he creado algunas funciones auxiliares para obtener el mejor y peor cromosoma dada una población, así como una para obtener una lista de los índices de los cromosomas ordenados de mejor a peor valor de evaluación.

III. Pseudocódigo de los Algoritmos

III-A. Algoritmo Greedy (Relief)

El algoritmo se basa en incrementar el peso de aquellas características que mejor separan a ejemplos que son enemigos entre sí y reducir el valor del peso en aquellas características que separan ejemplos que son amigos entre sí. El pseudocódigo del algoritmo **Greedy (Relief)** es el siguiente:

La función *nearestFriendEnemy* calcula el amigo y el enemigo más cercano a un dato dado en función de la distancia euclídea. Un dato se considera enemigo si tiene la clase distinta y amigo si tiene la misma clase del dato actual. La función emplea leave-one-out para evitar comparar distancias con el dato actual.

III-B. Búsqueda Local

La búsqueda local implementa una búsqueda de primero el mejor. El vector index nos indica en qué orden se van a modificar las componentes. Modificando así en cada paso una componente aleatoria que no hayamos modificado antes. El vector de pesos W se generará de manera aleatoria con valores

Algorithm 9 BL_MEM

Require: train, cromosoma, num_eval
1: w ← cromosoma.w
2: index ← 0,...,w.size()
3: mezcla los valores de index
4: // Reutiliza la evaluación del cromosoma
5: agr_ant ← cromosoma.pts;
6: // Condición de parada
7: **while** neighbour <2 * tamaño_w **do**
8: aux ← index[iter % tamaño_w]
9: w_mut ← w
10: w_mut[aux] += normal(generador)
11: truncar el vector de pesos mutado
12: // Evalúa el vector de pesos mutado
13: agr_new ← agregado(resultados KNN_LOO)
14: iter++
15: num_eval++
16: **if** agr_new >agr_ant **then**
17: w ← w_mut
18: agr_ant ← agr_new
19: **end if**
20: neighbour++
21: **if** iter % tamaño_w == 0 **then**
22: mezcla los valores de index
23: **end if**
24: **end while**
25: cromosoma.pts ← agr_ant

Algorithm 10 getBestCromosoma

Require: poblacion
Ensure: max
1: max ← 0
2: pts_max ← 0.0
3: **for** i = 0 to tamaño_población - 1 **do**
4: **if** poblacion[i].pts >pts_max **then**
5: max ← i
6: pts_max ← poblacion[i].pts
7: **end if**
8: **end for**
9: **return** max

Algorithm 11 getWorstCromosoma

Require: poblacion
Ensure: min
1: min ← 0
2: pts_min ← 100.0
3: **for** i = 0 to tamaño_población **do**
4: **if** poblacion[i].pts <pts_min **then**
5: min ← i
6: pts_min ← poblacion[i].pts
7: **end if**
8: **end for**
9: **return** min

Algorithm 12 getListBestCromosoma

Require: poblacion
Ensure: index
crear index(tamaño poblacion)
for i = 0 to tamaño poblacion **do**
  index[i] ← i
end for
ordenar index de mayor a menor poblacion[i].pts
return index

Algorithm 13 Relief

Require: train, w
Ensure: w
w ← 0,...,0
for i = 0 to tamaño train **do**
  nearestFriendEnemy(train, train[i], friend, enemy)
  for j = 0 to tamaño w **do**
    w[j] ← w[j] + |train[i].t[j] - train[i].t[enemy]| - |train[i].t[j] - train[i].t[friend]|
  end for
end for
w_max ← máximo del vector de pesos w
for i = 0 to tamaño w **do**
  if w[i] <0 **then**
    w[i] ← 0
  else
    w[i] ← w[i] / w_max
  end if
end for
return w

Algorithm 14 nearestFriendEnemy

Require: train, actual, friend, enemy
for i = 0 to tamaño train **do**
  if actual ≠ train[i] **then**
    distancia_actual ← euclideanDistance(train[i].t, actual.t)
    if train[i].category = actual.category **then**
      if distancia_actual < mejor_distancia_a **then**
        mejor_distancia_a ← distancia_actual
        friend ← i
      end if
    else
      if distancia_actual < mejor_distancia_e **then**
        mejor_distancia_e ← distancia_actual
        enemy ← i
      end if
    end if
  end if
end for

entre [0, 1] utilizando una distribución uniforme real. Para poder generar soluciones nuevas, deberemos modificar/mutar el vector W añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza, pero este método puede proporcionar soluciones negativas, por lo que debemos de truncar los valores negativos a 0. El pseudocódigo del algoritmo **Búsqueda Local** es el siguiente:

Algorithm 15 BL

Require: train, w
w ← distribucion_uniforme(0,1)
index ← 0,...,w.size()
mezcla los valores de index
// clasifica el vector de pesos w con KNN y calcula su agregado como tasa de evaluación
antiguo ← KNN(train, train, w)
agr$_a nt$← agregado(antiguo.clas, antiguo.red)
iter ← 0
neighbour ← 0
while iter <MAX_ITER and neighbour <tamaño w * MAX_NEIGHBOUR **do**
  aux ← index[iter % tamaño w]
  w_mut ← w
  w_mut[aux] += normal(generator)
  truncar el vector de pesos mutado
  // clasifica el vector de pesos mutado y calcula su agregado
  agr_new ← agregado(resultados KNN_LOO)
  iter++
  if agr_new >agr_ant **then**
    w ← w_mut
    agr_ant ← agr_new
    neighbour ← 0
  else
    neighbour++
  end if
  if iter % tamaño w = 0 **then**
    mezcla los valores de index
  end if
end while

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso y utiliza este resultado para evaluar la bondad de la solución obtenida. En este caso usamos alpha=0.5.

Algorithm 16 agregado

Require: t_clas, t_red, alpha
return alpha * t_clas + (1.0 - alpha) * t_red

III-C. Algoritmos Genéticos Generacionales

Partimos de una población de tamaño 30. Usaremos tres variables para indicar el mejor de la población anterior y el mejor y peor de la nueva población. Haremos las mutaciones esperadas (2 por generación) sobre el vector de hijos y

evaluaremos con el **k-NN** los cromosomas que hayan mutado o hayan sido generados nuevamente.

Aplicaremos *Elitismo* cuando el mejor cromosoma de la población anterior tenga un valor de evaluación superior al mejor de la nueva población. En ese caso, cambiaremos el mejor de la población anterior por el peor de la nueva.

III-C1. Algoritmo Genético Generacional cruce BLX-alpha:

III-C2. Algoritmo Genético Generacional cruce Aritmético: Para el cruce Aritmético, como el operador de cruce; en este caso, solo devuelve un único descendiente, crearemos el doble de padres para realizar el doble de cruces y obtener hijos diferentes y finalmente añadiremos a la población nueva los padres restantes para completar (TAM-PBL-num-cruces).

III-D. Algoritmos Genéticos Estacionarios

Partimos de una población de tamaño 30. En esta versión solo se generan dos hijos eligiendo en cada caso el cruce que corresponda. Usaremos tres variables para indicar el peor de la población anterior y el mejor y peor de la nueva población. Haremos las mutaciones esperadas (2 por generación) sobre el vector de hijos y evaluaremos con el **k-NN** los cromosomas que hayan mutado o hayan sido generados nuevamente. No se aplica Elitismo puesto que los hijos generados compiten con los dos peores de la población nueva para ser sustituidos.

III-D1. Algoritmo Genético Estacionario cruce BLX-alpha:

III-D2. Algoritmo Genético Estacionario cruce Aritmético: De forma similar a su versión Generacional, en esta versión seleccionaremos el doble de padres para obtener los dos hijos de la nueva población.

III-E. Algoritmos Meméticos

Para la implementación, usaremos una hibridación del Algoritmo Genético Generacional con el cruce que mejores resultados ha aportado (BLX-alpha en nuestro caso), con la Búsqueda Local. * La población se reduce a 10 individuos. * Usaremos una versión del Búsqueda Local que incluye una condición de parada hasta $2 \cdot$ numcaracteristicas vecinos y que elimina la inicialización del vector de pesos, ya que usaremos el que pasemos por parámetro en cada caso. * El resto de indicaciones se mantiene como en el AGG-BLX.

III-E1. Algoritmo Memético BLX (10, 1.0): Cada 10 generaciones, se aplica la Búsqueda Local sobre todos los cromosomas de la población.

III-E2. Algoritmo Memético BLX (10, 0.1): Cada 10 generaciones, se aplica la Búsqueda Local sobre un subconjunto de cromosomas de la población, seleccionado aleatoriamente con probabilidad 0.1 para cada cromosoma.

III-E3. Algoritmo Memético BLX (10, 0.1) mejor: Cada 10 generaciones, se aplica la Búsqueda Local sobre los $0.1 \cdot$ tampoblacion mejores cromosomas de la población actual.

IV. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar los datos obtenidos para cada algoritmo es siempre el mismo y se realiza en la función

Algorithm 17 AGG_BLX

Require: train, w

poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
// número de evaluaciones a 0
t=0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i=0 to TAM_PBL-1 **do**
  inicializar poblacion[i].w con distribucion uniforme
  // Evaluar cada cromosoma con el KNN_LOO
  poblacion[i].pts = agregado(resultados KNN_LOO)
  if poblacion[i].pts >poblacion[mejor].pts **then**
    mejor = i
  end if
  t++
end for
// MAX_ITER=15000
while t <MAX_ITER **do**
  for i=0 to TAM_PBL-1 **do**
    select = binaryTournament(poblacion, generador)
    padres[i] = poblacion[select]
  end for
  for i=0 to num_cruces-1, incrementando de 2 en 2 **do**
    hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
    poblacion_intermedia[i] = hijosBLX.first
    poblacion_intermedia[i+1] = hijosBLX.second
  end for
  for i=0 to num_mutaciones-1 **do**
    proceso de mutacion sobre poblacion_intermedia
  end for
  for i=num_cruces to TAM_PBL-1 **do**
    poblacion_intermedia[i] = padres[i]
  end for
  for i=0 to num_cruces-1 **do**
    poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
    t++
  end for
  mejorNueva = getBestCromosoma(poblacion_intermedia)
  peorNueva = getWorstCromosoma(poblacion_intermedia)
  if poblacion[mejor].pts >poblacion_intermedia[mejorNueva].pts **then**
    poblacion_intermedia[peorNueva] = poblacion[mejor]
    mejor = peorNueva
  else
    mejor = mejorNueva
  end if
  poblacion = poblacion_intermedia
end while
w = poblacion[mejor].w

Algorithm 18 AGG_CA

```
Require: train, w
// vector<Cromosoma>con TAM_PBL=30
poblacion(TAM_PBL), padres(num_cruces+TAM_PBL), poblacion_intermedia(TAM_PBL)
// número de evaluaciones a 0
t = 0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i = 0 to TAM_PBL - 1 do
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts >poblacion[mejor].pts then
        mejor = i
    end if
    t++
end for
// MAX_ITER=15000
while t <MAX_ITER do
    for i = 0 to num_cruces + TAM_PBL - 1 do
        select = binaryTournament(poblacion, generador)
        padres[i] = poblacion[select]
    end for
    for i = 0 to num_cruces*2 - 1 step 2 do
        hijoCA = cruceArit(padres[i], padres[i+1], generador)
        poblacion_intermedia[i/2] = hijoCA
    end for
    for i = 0 to num_mutaciones - 1 do
    end for
    for i = num_cruces*2 to num_cruces + TAM_PBL - 1 do
        poblacion_intermedia[i - num_cruces] = padres[i]
    end for
    for i = 0 to num_cruces - 1 do
        poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
        t++
    end for
    mejorNueva = getBestCromosoma(poblacion_intermedia)
    peorNueva = getWorstCromosoma(poblacion_intermedia)
    if poblacion[mejor].pts >poblacion_intermedia[mejorNueva].pts then
        poblacion_intermedia[peorNueva] = poblacion[mejor]
        mejor = peorNueva
    else
        mejor = mejorNueva
    end if
    // Sustituimos la población
    poblacion = poblacion_intermedia
end while
w = poblacion[mejor].w
```

Algorithm 19 AGE_BLX

```
Require: train, w
// vector<Cromosoma>con TAM_PBL=30
poblacion(TAM_PBL), poblacion_intermedia(2)
// número de evaluaciones a 0
t = 0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = 2 * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i = 0 to TAM_PBL - 1 do
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts <poblacion[peor].pts then
        peor = i
    end if
    t++
end for
// MAX_ITER=15000
while t <MAX_ITER do
    select = binaryTournament(poblacion, generador)
    p1 = poblacion[select]
    select = binaryTournament(poblacion, generador)
    p2 = poblacion[select]
    hijosBLX = cruceBLX(p1, p2, generador)
    poblacion_intermedia[0] = hijosBLX.first
    poblacion_intermedia[1] = hijosBLX.second
    for i = 0 to num_mutaciones - 1 do
        proceso de mutacion sobre poblacion_intermedia
    end for
    for i = 0 to 1 do
        poblacion_intermedia[i].pts = agregado(resultados KNN_LOO)
        t++
    end for
    if poblacion_intermedia[0].pts >poblacion_intermedia[1].pts then
        mejorNueva = 0
        peorNueva = 1
    else
        mejorNueva = 1
        peorNueva = 0
    end if
    if poblacion_intermedia[mejorNueva].pts >poblacion[peor].pts then
        poblacion[peor] = poblacion_intermedia[mejorNueva]
        peor = getWorstCromosoma(poblacion)
        if poblacion_intermedia[peorNueva].pts >poblacion[peor].pts then
            poblacion[peor] = poblacion_intermedia[peorNueva]
            peor = getWorstCromosoma(poblacion)
        end if
    end if
end while
w = poblacion[getBestCromosoma(poblacion)].w
```

Algorithm 20 AGE_CA

Require: train, w

// vector<Cromosoma>con TAM_PBL=30
poblacion(TAM_PBL), poblacion_intermedia(2)
// número de evaluaciones a 0
t = 0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = 2 * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i = 0 to TAM_PBL - 1 **do**
  poblacion[i].pts = agregado(resultados KNN_LOO)
  if poblacion[i].pts <poblacion[peor].pts **then**
    peor = i
  end if
  t++
end for
// MAX_ITER=15000
while t <MAX_ITER **do**
  select = binaryTournament(poblacion, generador)
  p1 = poblacion[select]
  select = binaryTournament(poblacion, generador)
  p2 = poblacion[select]
  poblacionintermedia[0] = cruceArit(p1, p2)
  select = binaryTournament(poblacion, generador)
  p1 = poblacion[select]
  select = binaryTournament(poblacion, generador)
  p2 = poblacion[select]
  poblacionintermedia[1] = cruceArit(p1, p2)
  for i = 0 to num_mutaciones - 1 **do**
    proceso de mutacion sobre poblacion_intermedia
  end for
  for i = 0 to 1 **do**
    poblacionintermedia[i].pts = agregado(resultados KNN_LOO)
    t++
  end for
  if poblacionintermedia[0].pts >poblacionintermedia[1].pts **then**
    mejorNueva = 0
    peorNueva = 1
  else
    mejorNueva = 1
    peorNueva = 0
  end if
  if poblacionintermedia[mejorNueva].pts >poblacion[peor].pts **then**
    poblacion[peor] = poblacionintermedia[mejorNueva]
    peor = getWorstCromosoma(poblacion)
    if poblacionintermedia[peorNueva].pts >poblacion[peor].pts **then**
      poblacion[peor] = poblacionintermedia[peorNueva]
      peor = getWorstCromosoma(poblacion)
    end if
  end if
end while
w = poblacion[getBestCromosoma(poblacion)].w

Algorithm 21 AM_BLX_10_1_0

Require: train, w

// vector<Cromosoma>con TAM_PBL=10
poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
// número de evaluaciones a 0
t = 0, generacion = 0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i = 0 to TAM_PBL - 1 **do**
  inicializar poblacion[i].w con distribucion uniforme
  // Evaluar cada cromosoma con el KNN_LOO
  poblacion[i].pts = agregado(resultados KNN_LOO)
  if poblacion[i].pts >poblacion[mejor].pts **then**
    mejor = i
  end if
  t++
end for
// MAX_ITER=15000
while t <MAX_ITER **do**
  generacion++
  for i = 0 to TAM_PBL - 1 **do**
    select = binaryTournament(poblacion, generador)
    padres[i] = poblacion[select]
  end for
  for i = 0 to num_cruces - 1 step 2 **do**
    hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
    poblacionintermedia[i] = hijosBLX.first
  poblacion$_intermedia[i+1] = hijosBLX.second$
  end for
  for i = 0 to num_mutaciones - 1 **do**
    proceso de mutacion sobre poblacion_intermedia
  end for
  // Introducimos los últimos padres en la población
  for i = num_cruces to TAM_PBL - 1 **do**
    poblacionintermedia[i] = padres[i]
  end for
  if generacion % 10 == 0 **then**
    for i = 0 to TAM_PBL - 1 **do**
    BL$_MEM(train, poblacionintermedia[i], t)$
    end for
  end if
  mejorNueva = getBestCromosoma(poblacion$_intermedia)peorNue$
  $getWorstCromosoma(poblacion_intermedia)$
if poblacion[mejor].pts >poblacionintermedia[mejorNueva].pts **then**
poblacion$_intermedia[peorNueva]$ = $poblacion[mejor]mejor = peorNueva$
else
  mejor = mejorNueva
end if
poblacion = poblacionintermedia
end while
w = poblacion[mejor].w

Algorithm 22 AM_BLX_10_0_1

```
Require: train, w
poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
t = 0, generacion = 0
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_bl_cromosoma = ceil(0.1*TAM_PBL)
num_mutaciones = ceil(PB_MUT * genesporgeneracion)
distribucion = randommutaciones(0, genesporgeneracion-1)
for i = 0 to TAM_PBL - 1 do
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts >poblacion[mejor].pts then
        mejor = i
    end if
    t++
end for
// MAX_ITER=15000
while t <MAX_ITER do
    generacion++
    for i = 0 to TAM_PBL - 1 do
        select = binaryTournament(poblacion, generador)
        padres[i] = poblacion[select]
    end for
    for i = 0 to num_cruces - 1 step 2 do
        hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
        poblacionintermedia[i] = hijosBLX.first
        poblacionintermedia[i+1] = hijosBLX.second
    end for
    for i = 0 to num_mutaciones - 1 do
    end for
    for i = num_cruces to TAM_PBL - 1 do
        poblacionintermedia[i] = padres[i]
    end for
    for i = 0 to num_cruces - 1 do
        poblacionintermedia[i].pts = agregado(resultados KNN_LOO)
        t++
    end for
    if generacion % 10 == 0 then
        for int i = 0 to num_bl_cromosoma - 1 do
            select = random_BL(generador)
            BL_MEM(train, poblacionintermedia[select], t)
        end for
    end if
    mejorNueva = getBestCromosoma(poblacionintermedia)
    peorNueva = getWorstCromosoma(poblacionintermedia)
    if poblacion[mejor].pts >poblacionintermedia[mejorNueva].pts then
        poblacionintermedia[peorNueva] = poblacion[mejor]
        mejor = peorNueva
    else
        mejor = mejorNueva
    end if
    poblacion = poblacionintermedia
end while
w = poblacion[mejor].w
```

ejecutar para cada algoritmo programado. Primero obtenemos los pesos usando el algoritmo a comparar y seguidamente clasificamos con el **k-NN** los datos de train y test sobre ese vector de pesos obtenido. Por último devolvemos en un struct Resultados las tasas de clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

V. Procedimiento del Desarrollo de la Práctica y Manual de Usuario

La práctica ha sido programada en el lenguaje C++. Para el desarrollo de la práctica, primero he necesitado leer los archivos que nos proporcionan los datos, para ello, he optado por pasar los archivos .arff a .csv, un formato que al menos para mí, es más manipulable. Seguidamente y con ayuda de la función read_csv, guardo los datos en memoria en un vector de estructuras que he denominado *FicheroCSV*. Cada *FicheroCSV* contiene la información de cada línea del fichero csv original incluyendo un vector de los datos y un string que indica la

Algorithm 23 AM_BLX_10_0_1_mej

```
Require: train, w
// vector<Cromosoma>con TAM_PBL=10
poblacion(TAM_PBL), padres(TAM_PBL), poblacion_intermedia(TAM_PBL)
// número de evaluaciones a 0
t = 0, generacion = 0
// PB_MUT=0.7
num_cruces = floor(PB_CRUCE * (TAM_PBL/2))*2
genes_por_generacion = num_cruces * tamaño características
num_bl_cromosoma = ceil(0.1*TAM_PBL)
num_mutaciones = ceil(PB_MUT * genes_por_generacion)
distribucion = random_mutaciones(0, genes_por_generacion-1)
for i = 0 to TAM_PBL - 1 do
    inicializar poblacion[i].w con distribucion uniforme
    // Evaluar cada cromosoma con el KNN_LOO
    poblacion[i].pts = agregado(resultados KNN_LOO)
    if poblacion[i].pts >poblacion[mejor].pts then
        mejor = i
    end if
    // Incremento el número de evaluaciones
    t++
end for
// MAX_ITER=15000
while t <MAX_ITER do
    generacion++
    // Seleccionamos los padres con el torneo binario
    for i = 0 to TAM_PBL - 1 do
        select = binaryTournament(poblacion, generador)
        padres[i] = poblacion[select]
    end for
    // Hacemos el cruce BLX para obtener los hijos
    for i = 0 to num_cruces - 1 step 2 do
        hijosBLX = cruceBLX(padres[i], padres[i+1], generador)
        poblacionintermedia[i] = hijosBLX.first
        poblacionintermedia[i+1] = hijosBLX.second
    end for
    // Hacemos el número de mutaciones esperadas
    for i = 0 to num_mutaciones - 1 do
        proceso de mutacion sobre poblacion_intermedia
    end for
    // Introducimos los últimos padres en la población
    for i = num_cruces to TAM_PBL - 1 do
poblacion_intermedia[i] = padres[i]
    end for
    // Evaluamos los nuevos hijos
    for i = 0 to num_cruces - 1 do
        poblacionintermedia[i].pts = agregado(resultados KNN_LOO)
        t++
    end for
    lista = getListBestCromosoma(poblacionintermedia)
    // Aplicamos Local Search al mejor cromosoma de la población
    if generacion % 10 == 0 then
        for int i = 0 to num_bl_cromosoma - 1 do
            select = lista[i]
            BL_MEM(train, poblacionintermedia[select], t)
        end for
    end if
    // Calculamos el mejor y el peor Cromosoma de la nueva población
    mejorNueva = getBestCromosoma(poblacionintermedia)
    peorNueva = getWorstCromosoma(poblacionintermedia)
    // Elitismo
    if poblacion[mejor].pts >poblacionintermedia[mejorNueva].pts then
        poblacionintermedia[peorNueva] = poblacion[mejor]
        mejor = peorNueva
    else
        mejor = mejorNueva
    end if
    // Sustituimos la población
    poblacion = poblacionintermedia
end while
w = poblacion[mejor].w
```

Algorithm 24 ejecutarAlgoritmo

Require: particion, i

test ← toma la partición i
train ← toma la suma de datos de las particiones $\neq i$
w ← Algoritmo(train, w)
resultados ← KNN(train, test, w)

clase de los mismos. Por tanto, consideraremos cada conjunto de datos como un *vector<FicheroCSV>*.

Seguidamente, y para aplicar la *5-fold cross validation*, creo un vector con 5 particiones que contienen punteros (para reducir el coste de memoria) a estructuras *FicheroCSV* repartidas de forma equitativa. Los datos se introducen en la partición ya normalizados.

Una vez que tenemos el vector<vector<FicheroCSV*, ya podemos repartir los datos en train y test y proceder como hemos descrito anteriormente con los algoritmos programados.

Para la primera práctica, desarrollé una versión del **k-NN** sin leave-one-out, seguidamente programé el algoritmo **Greedy (Relief)** y finalmente la Búsqueda Local con la cual tuve que modificar parte del código del **k-NN** para que aplicase el leave-one-out.

Para la segunda práctica, primero programé el código para los cruces **BLX** y **Aritmético** y más tarde desarrollé los 4 algoritmos Genéticos. Por últimos, cambié el código del **Búsqueda Local** para adaptarlo a las restricciones que se imponen en el guión y poder usarlo en los 3 algoritmos Meméticos que se nos piden.

En el archivo *LEEME.txt* se encuentra explicado el proceso para replicar las ejecuciones del programa. Así que me limitaré a resumirlo:

Existe un make que genera el ejecutable del programa. El programa ejecutable ha sido compilado con g++ y optimizado con -O2 para reducir los tiempos de ejecución. ./bin/p2 ./data/archivo.csv seed

Ejecuta los diez algoritmos programados en ambas prácticas sobre el conjunto de datos contenido en el archivo.csv (con archivo = Ecoli, Parkinsons, Breast_cancer) usando como semilla el número seed pasado como parámetro en el **Búsqueda Local**, **Algoritmos Genéticos** y **Meméticos**.

VI. Experimentos y Análisis de Resultados

Para la obtención de los resultados de esta práctica, hemos utilizado los siguientes conjuntos de datos:

Ecoli: es una base de datos para identificar la posición de proteínas tras obtener métricas mediante una serie de técnicas distintas. Consta de 366 ejemplos. Consta de 8 atributos (clase incluida). Consta de 8 clases (citoplasma, membrana interna, periplasma, ...). Atributos: Método de McGeoch's, Método de Heijne's...

Parkinsons: contiene datos que se utilizan para distinguir entre la presencia y la ausencia de la enfermedad de Parkinson en una serie de pacientes a partir de medidas biomédicas de la voz. Consta de 195 ejemplos. Consta de 23 atributos (clase incluida). Consta de 2 clases (1 Sano, 2 Enfermo), La distribución de ejemplos está desbalanceada (147 enfermos, 48 sanos). Atributos: Distintas métricas sonoras (como periodo pitch, Ratio harmónico-ruido).

Breast-Cancer: es una base de datos para identificar la gravedad de cancer de máma a partir de distintos atributos. Consta de 569 ejemplos. Consta de 31 atributos (clase incluida). Consta de 2 clase (B Benigno, M Maligno). Atributos: Métricas usando 3 imágenes con distinta orientación, cada una con radio, textura, perímetro, área, ... 10 por imágenes, obteniendo 30 en total.

El algoritmo de Búsqueda Local y los Genéticos y Meméticos dependen de un parámetro que especifica la semilla para le generación de números aleatorios. Voy a utilizar para el análisis de estos resultados SEED=42.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza donde varianza = 0.3. Como criterio de parada en la **Búsqueda Local**, se va a usar el número de evaluaciones que se han realizado así como el número vecinos por característica explorados. En nuestro caso los valores serán 15000 y $20 \cdot$ *tam_vector_pesos* respectivamente. Las tablas se generan automáticamente con la ejecución del programa y contienen los resultados para cada partición, así como la media de los resultados para cada algoritmo. A continuación, se muestran las tablas para cada fichero y cada algoritmo utilizado:

Teniendo en cuenta solo los algoritmos de la primera práctica, podemos ver que los mejores resultados son obtenidos por el algoritmo **1-NN** para el conjunto de datos *Ecoli* igualando casi al algoritmo **Relief**. Mientras que el **Búsqueda Local** es el que obtiene en los tres casos la mejor tasa de reducción, lo que indica que se necesitan menos atributos para clasificar los datos con respecto al resto de algoritmos.

Nos quedaremos por tanto con el **Búsqueda Local** para hacer las comparaciones con los algoritmos evolutivos.

Respecto a la segunda práctica, las tasas de clasificación de los diferentes algoritmos son similares en media. Siendo mejores los algoritmos meméticos consistentemente.

Si observamos la tasa de reducción, en este caso, podemos ver que el **Búsqueda Local** obtiene bastante buenos resultados de media, solo es superado por los **Genéticos Estacionarios** y los **Meméticos** salvo excepciones (**AGG_CA** lo iguala).

Esto es debido a que en la **Búsqueda Local** empezamos muy pronto a descartar características para poder clasificar los datos, en los algoritmos genéticos hay que esperar varias mutaciones para que al final se acabe realizando un proceso similar al de la **Búsqueda Local**. Es por ello que obtiene en varias ocasiones; resultados mejores, sobre todo en conjunto de datos más pequeños, en los algoritmos evolutivos no explotan todo su potencial.

Visualizando los resultados de *Parkinsons* (el que menor cantidad de datos presenta) podemos ver que el algoritmo **AGG-CA** obtiene la mejortasa de clasificación, mientras que no por ello presenta una mejor tasa de reducción. Sobre *Ecoli* ya no presenta los mejores resultados de clasificación.

En *Ecoli* los algoritmos evolutivos (**AGE-CA** y **AM-(10,0.1)** superan a **1-NN** y **RELIEF** en precisión a partir de tasas de al rededor del 63.88 % y superiores, lo cual sugiere que para $x \approx 64\,\%$, los algoritmos evolutivos empiezan a ser más competitivos.

En *Parkinsons* **AGG-CA** y **AGE-CA** muestran una superioridad a partir del 82.79 % en precisión, mientras que **RELIEF** se mantiene cercano. Para $x \approx 80\,\%$, los algoritmos evolutivos generalmente superan a los métodos tradicionales.

En *Breast-Cancer* **AGE-BLX** y **AM-(10,0.1-mej)** muestran tasas de clasificación competitivas, aunque **1-NN** y **RELIEF** también son altos. Para $x \approx 92\,\%$, los algoritmos evolutivos como **AGE-BLX** empiezan a tener una ventaja leve.

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Agr,*	T	*%_clas*	*%red*	*Agr,*	T	*%_clas*	*%red*	*Agr,*	T
1-NN	75,93	0,00	37,96	1,20E-04	77,86	0,00	38,93	1,00E-04	92,79	0,00	46,39	1,00E-04
RELIEF	73,26	43,00	58,13	1,00E-04	80,21	29,22	54,72	1,00E-04	93,51	28,77	61,14	2,00E-04
BL	62,10	74,00	67,82	5,76E-03	79,50	80,00	79,75	1,63E-02	90,86	83,48	87,17	2,97E-01
AGG-BLX	62,01	76,00	69,00	2,86E-01	80,71	86,96	83,84	2,54E-01	90,86	88,00	89,43	2,54E+00
AGG-CA	63,88	74,00	68,94	3,03E-01	83,29	78,78	81,03	2,55E-01	91,55	67,74	79,64	2,80E+00
AGE-BLX	63,65	76,00	69,83	2,93E-01	78,14	88,00	83,07	2,70E-01	92,81	90,32	91,56	2,55E+00
AGE-CA	65,05	76,00	70,53	2,86E-01	82,79	87,83	85,31	2,56E-01	91,05	87,74	89,40	2,57E+00
AM-(10,1.0)	64,14	77,50	70,82	2,90E-01	77,14	89,57	83,35	2,61E-01	90,15	91,48	90,82	2,65E+00
AM-(10,0.1)	66,60	75,00	70,80	2,90E-01	77,43	88,87	83,15	2,53E-01	91,90	90,19	91,05	2,54E+00
AM-(10,0.1mej)	61,67	76,50	69,09	2,91E-01	82,79	87,30	85,05	2,45E-01	91,94	89,86	90,81	2,49E+00

Figura 1. Tabla datos comparativa

Estos valores indican que, en general, los algoritmos evolutivos empiezan a destacar en precisión de clasificación en intervalos específicos dependiendo del conjunto de datos.

Podemos ver los resultados totales obtenidos para cada conjunto de datos:

	Ecoli			
	1-NN			
	%_clas	*%red*	*Agr,*	T
Partición 1	78,57	0,00	39,29	2,00E-04
Partición 2	67,14	0,00	33,57	1,00E-04
Partición 3	76,59	0,00	38,30	1,00E-04
Partición 4	80,66	0,00	40,33	1,00E-04
Partición 5	76,67	0,00	38,33	1,00E-04
Media	75,93	0,00	37,96	1,20E-04
	Greedy-Relief			
Partición 1	74,29	40,00	57,14	1,00E-04
Partición 2	57,14	45,00	51,07	1,00E-04
Partición 3	77,91	47,50	62,71	1,00E-04
Partición 4	83,63	52,50	68,06	1,00E-04
Partición 5	73,33	30,00	51,67	1,00E-04
Media	73,26	43,00	58,13	1,00E-04
	BL			
Partición 1	61,43	72,50	66,96	6,40E-03
Partición 2	60,00	72,50	66,25	6,80E-03
Partición 3	60,22	70,00	65,11	5,40E-03
Partición 4	63,87	77,50	70,34	5,60E-03
Partición 5	65,00	77,50	70,42	4,60E-03
Media	62,10	74,00	67,82	5,76E-03

Figura 2. Algoritmos k-NN, Relief y Local Search Ecoli.csv

Vemos que en general los algoritmos evolutivos muestran tiempos de ejecución que son manejables y relativamente similares entre sí, con los algoritmos Meméticos presentando una ligera ventaja en algunos casos específicos. Esto sugiere que los algoritmos Meméticos no solo son efectivos en términos de precisión de clasificación y reducción de características, sino que también son eficientes en términos de tiempo y ejecución.

Como hemos visto antes, para conjuntos de datos pequeños, como *Parkinsons*, no compensa utilizan algoritmos evolutivos, ya que el tiempo de ejecución comparado con el de **BL** es bastante mayor. Mientras que para el conjunto *BreastCancer* el

	AGG_BLX			
Partición 1	51,43	75,00	63,21	3,10E-01
Partición 2	55,71	77,50	66,61	3,17E-01
Partición 3	70,55	72,50	71,52	2,89E-01
Partición 4	69,01	77,50	73,26	2,95E-01
Partición 5	63,33	77,50	70,42	2,21E-01
Media	62,01	76,00	69,00	2,86E-01
	AGG_CA			
Partición 1	55,71	75,00	65,36	3,21E-01
Partición 2	50,00	75,00	62,50	3,50E-01
Partición 3	76,59	72,50	74,55	3,05E-01
Partición 4	72,09	72,50	72,29	2,95E-01
Partición 5	65,00	75,00	70,00	2,44E-01
Media	63,88	74,00	68,94	3,03E-01
	AGE_BLX			
Partición 1	60,00	75,00	67,50	3,11E-01
Partición 2	57,14	77,50	67,32	3,18E-01
Partición 3	68,90	72,50	70,70	3,19E-01
Partición 4	70,55	75,00	72,77	2,95E-01
Partición 5	61,67	80,00	70,83	2,19E-01
Media	63,65	76,00	69,83	2,93E-01
	AGE_CA			
Partición 1	58,57	72,50	65,54	3,04E-01
Partición 2	55,71	82,50	69,11	3,04E-01
Partición 3	72,09	75,00	73,54	3,01E-01
Partición 4	70,55	75,00	72,77	2,91E-01
Partición 5	68,33	75,00	71,67	2,30E-01
Media	65,05	76,00	70,53	2,86E-01

Figura 3. Algoritmos Geneticos en Ecoli.csv

algoritmo **Memético (10,0.1)mejor** es el que obtiene mejores resultados.

Referencias

[1] M. Mitchell, "Genetic algorithms: An overview." in *Complex.*, vol. 1, no. 1. Citeseer, 1995, pp. 31–39.

[2] B. Melián, J. A. M. Pérez, and J. M. M. Vega, "Metaheurísticas: Una visión global," *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, vol. 7, no. 19, p. 0, 2003.

	AM_BLX(10,1,0)			
Partición 1	58,57	75,00	66,79	3,11E-01
Partición 2	57,14	85,00	71,07	3,14E-01
Partición 3	66,37	75,00	70,69	3,02E-01
Partición 4	73,63	75,00	74,31	2,93E-01
Partición 5	65,00	77,50	71,25	2,30E-01
Media	64,14	77,50	70,82	2,90E-01
	AM_BLX(10,0,1)			
Partición 1	60,00	72,50	66,25	3,16E-01
Partición 2	60,00	80,00	70,00	3,14E-01
Partición 3	64,62	72,50	68,56	3,13E-01
Partición 4	75,05	75,00	75,03	2,90E-01
Partición 5	73,33	75,00	74,17	2,19E-01
Media	66,60	75,00	70,80	2,90E-01
	AM_BLX(10,0,1)_mej			
Partición 1	57,14	75,00	66,07	3,19E-01
Partición 2	54,29	82,50	68,39	3,00E-01
Partición 3	61,76	72,50	67,13	3,09E-01
Partición 4	75,16	75,00	75,08	2,96E-01
Partición 5	60,00	77,50	68,75	2,29E-01
Media	61,67	76,50	69,09	2,91E-01

Figura 4. Algoritmos Memeticos en Ecoli.csv

	Parkinsons			
	1-NN			
	%_clas	*%red*	*Agr,*	T
Partición 1	72,50	0,00	36,25	1,00E-04
Partición 2	77,50	0,00	38,75	1,00E-04
Partición 3	80,00	0,00	40,00	1,00E-04
Partición 4	85,00	0,00	42,50	1,00E-04
Partición 5	74,29	0,00	37,14	1,00E-04
Media	77,86	0,00	38,93	1,00E-04
	Greedy-Relief			
Partición 1	77,50	17,39	47,45	1,00E-04
Partición 2	72,50	25,22	48,86	1,00E-04
Partición 3	85,00	26,96	55,98	1,00E-04
Partición 4	77,50	36,52	57,01	1,00E-04
Partición 5	88,57	40,00	64,29	1,00E-04
Media	80,21	29,22	54,72	1,00E-04
	BL			
Partición 1	77,50	79,13	78,32	1,88E-02
Partición 2	75,00	78,26	76,63	1,82E-02
Partición 3	82,50	78,26	80,38	1,46E-02
Partición 4	82,50	80,87	81,68	1,32E-02
Partición 5	80,00	83,48	81,74	1,66E-02
Media	79,50	80,00	79,75	1,63E-02

Figura 5. Algoritmos k-NN, Relief y LS Parkinsons.csv

	AGG_BLX			
Partición 1	82,50	89,57	86,03	2,70E-01
Partición 2	60,00	85,22	72,61	2,79E-01
Partición 3	87,50	85,22	86,36	2,60E-01
Partición 4	85,00	85,22	85,11	2,57E-01
Partición 5	88,57	89,57	89,07	2,03E-01
Media	80,71	86,96	83,84	2,54E-01
	AGG_CA			
Partición 1	80,00	80,87	80,43	2,67E-01
Partición 2	80,00	72,17	76,09	2,71E-01
Partición 3	87,50	80,87	84,18	2,63E-01
Partición 4	77,50	76,52	77,01	2,75E-01
Partición 5	91,43	83,48	87,45	1,97E-01
Media	83,29	78,78	81,03	2,55E-01
	AGE_BLX			
Partición 1	75,00	91,30	83,15	3,03E-01
Partición 2	82,50	87,83	85,16	2,68E-01
Partición 3	77,50	85,22	81,36	2,76E-01
Partición 4	70,00	86,96	78,48	3,02E-01
Partición 5	85,71	88,70	87,21	1,98E-01
Media	78,14	88,00	83,07	2,70E-01
	AGE_CA			
Partición 1	77,50	89,57	83,53	2,74E-01
Partición 2	82,50	87,83	85,16	2,68E-01
Partición 3	80,00	86,96	83,48	2,67E-01
Partición 4	82,50	81,74	82,12	2,57E-01
Partición 5	91,43	93,04	92,24	2,15E-01
Media	82,79	87,83	85,31	2,56E-01

Figura 6. Algoritmos Geneticos Parkinsons.csv

	AM_BLX(10,1,0)			
Partición 1	75,00	88,70	81,85	2,66E-01
Partición 2	77,50	86,96	82,23	2,85E-01
Partición 3	85,00	88,70	86,85	2,76E-01
Partición 4	62,50	89,57	76,03	2,69E-01
Partición 5	85,71	93,91	89,81	2,11E-01
Media	77,14	89,57	83,35	2,61E-01
	AM_BLX(10,0,1)			
Partición 1	75,00	90,43	82,72	2,80E-01
Partición 2	72,50	86,96	79,73	2,54E-01
Partición 3	82,50	89,57	86,03	2,58E-01
Partición 4	80,00	86,96	83,48	2,67E-01
Partición 5	77,14	90,43	83,79	2,04E-01
Media	77,43	88,87	83,15	2,53E-01
	AM_BLX(10,0,1)_mej			
Partición 1	70,00	86,96	78,48	2,54E-01
Partición 2	80,00	86,09	83,04	2,45E-01
Partición 3	90,00	86,09	88,04	2,77E-01
Partición 4	82,50	86,09	84,29	2,44E-01
Partición 5	91,43	91,30	91,37	2,02E-01
Media	82,79	87,30	85,05	2,45E-01

Figura 7. Algoritmos Memeticos Parkinsons.csv

	Breast-Cancer			
	1-NN			
	%_clas	*%red*	*Agr,*	T
Partición 1	97,39	0,00	48,70	1,00E-04
Partición 2	96,52	0,00	48,26	1,00E-04
Partición 3	82,61	0,00	41,30	1,00E-04
Partición 4	95,65	0,00	47,83	1,00E-04
Partición 5	91,77	0,00	45,89	1,00E-04
Media	92,79	0,00	46,39	1,00E-04
	Greedy-Relief			
Partición 1	96,52	31,61	64,07	1,00E-04
Partición 2	97,39	22,58	59,99	1,00E-04
Partición 3	84,35	26,45	55,40	2,00E-04
Partición 4	95,65	31,61	63,63	4,00E-04
Partición 5	93,64	31,61	62,62	2,00E-04
Media	93,51	28,77	61,14	2,00E-04
	BL			
Partición 1	93,04	79,35	86,20	2,41E-01
Partición 2	95,65	83,23	89,44	2,86E-01
Partición 3	84,35	82,58	83,46	3,24E-01
Partición 4	90,43	85,81	88,12	3,05E-01
Partición 5	90,82	86,45	88,64	3,27E-01
Media	90,86	83,48	87,17	2,97E-01

Figura 8. Algoritmos k-NN, Relief y LS Breast-Cancer.csv

	AGG_BLX			
Partición 1	93,04	88,39	90,72	2,5236
Partición 2	96,52	88,39	92,45	2,62E+00
Partición 3	82,61	87,74	85,18	2,65E+00
Partición 4	91,30	90,32	90,81	2,59E+00
Partición 5	90,82	85,16	87,99	2,31E+00
Media	90,86	88,00	89,43	2,54E+00
	AGG_CA			
Partición 1	96,52	61,29	78,91	2,82E+00
Partición 2	94,78	74,84	84,81	2,78E+00
Partición 3	80,87	67,10	73,98	2,80E+00
Partición 4	94,78	61,94	78,36	2,98E+00
Partición 5	90,78	73,55	82,16	2,61E+00
Media	91,55	67,74	79,64	2,80E+00
	AGE_BLX			
Partición 1	93,04	91,61	92,33	2,4662
Partición 2	96,52	91,61	94,07	2,61E+00
Partición 3	86,96	87,10	87,03	2,88E+00
Partición 4	94,78	89,68	92,23	2,47E+00
Partición 5	92,73	91,61	92,17	2,33E+00
Media	92,81	90,32	91,56	2,55E+00
	AGE_CA			
Partición 1	91,30	89,03	90,17	2,48E+00
Partición 2	93,04	91,61	92,33	2,45E+00
Partición 3	84,35	81,94	83,14	2,79E+00
Partición 4	95,65	87,74	91,70	2,73E+00
Partición 5	90,91	88,39	89,65	2,41E+00
Media	91,05	87,74	89,40	2,57E+00

Figura 9. Algoritmos Geneticos Breast-Cancer.csv

	AM_BLX(10,1,0)			
Partición 1	91,30	90,97	91,14	2,59E+00
Partición 2	92,17	91,61	91,89	2,7144
Partición 3	83,48	90,32	86,90	2,71E+00
Partición 4	95,65	91,61	93,63	2,78E+00
Partición 5	88,14	92,90	90,52	2,45E+00
Media	90,15	91,48	90,82	2,65E+00
	AM_BLX(10,0,1)			
Partición 1	93,91	90,97	92,44	2,55E+00
Partición 2	94,78	91,61	93,20	2,50E+00
Partición 3	86,09	87,10	86,59	2,71E+00
Partición 4	93,91	90,32	92,12	2,51E+00
Partición 5	90,82	90,97	90,90	2,41E+00
Media	91,90	90,19	91,05	2,54E+00
	AM_BLX(10,0,1)_mej			
Partición 1	94,78	89,03	91,91	2,44E+00
Partición 2	90,43	90,32	90,38	2,41E+00
Partición 3	86,96	88,39	87,67	2,62E+00
Partición 4	94,78	90,32	92,55	2,63E+00
Partición 5	92,73	90,32	91,52	2,33E+00
Media	91,94	89,86	90,81	2,49E+00

Figura 10. Algoritmos Memeticos Breast-Cancer.csv