Metaheurísticas

Departamento de Ciencias de la Computación e Inteligencia Artificial

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Práctica 2: Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

Álvaro Rodríguez Gallardo. 77034155W.

alvaro155w@correo.ugr.es

Grupo 2 (17.30-19.30)

Curso 2023-2024

Índice general

Descripción del problema

El problema afrontado, Aprendizaje de Pesos en Características, consiste en obtener un vector de pesos que permita ponderar aquellas características de una muestra, intentando optimizar el porcentaje de clasificación para futuras muestras con mismas características. Si se tiene que la muestra tiene $n \in \mathbb{N}$ características, entonces el vector $c_i = (c_1, ..., c_n)$,$0 \leq c_i \leq 1$, representa las características de la i-ésima componente del dato $X = (x_1, ..., x_m)$, con $m \in \mathbb{N}$ y $1 \leq i \leq m$. Además, cada componente tendrá asociada una clase, es decir, para un vector de datos como X, para la componente i, que se llamará "muestra i-ésima", se le asociará un vector de características y una clase. Fijado X anterior, se define para cierto conjunto A de clases

$$p : \mathbb{R} \to \mathbb{R}^n \times A$$

$$p(x_i) = (c_i, \text{clase}_i)$$

para $1 \leq i \leq m$ y $x_i \in X = (x_1, ..., x_m) \subset \mathbb{R}^m$.

Con lo anterior, se quiere encontrar un vector de pesos $W = (w_1, ..., w_n)$, con el peso j-ésimo ponderando a la característica j-ésima del vector de características. Para ello, se usará un clasificador k-NN, pero en el caso k= 1, con el que se considera el vecino más cercano (muestra con menor distancia euclídea).

Haciendo uso de 5-fold cross validation, se buscará tanto encontrar el vector de pesos correspondiente como evaluar la calidad del mismo (primero aprendizaje, luego validación). Puesto que el dataset se divide en cinco particiones, se separa una partición para validación y se usa el resto de entrenamiento (todas deben ser para validación en algún momento). A continuación, se usan varios estadísticos para evaluar la calidad mencionada, con la media de los cinco porcentajes de clasificación obtenidos con 5-fold cross validation.

Se mezclan dos estadísticos:

- Precisión:

$$\text{tasa\_clas} = 100 \cdot \frac{\text{Nº instancias bien clasificadas en T}}{\text{Nº instancias en T}}$$

- Simplicidad:

$$\text{tasa\_red} = 100 \cdot \frac{\text{Nº valores } w_j < 0{,}1}{\text{Nº características}}$$

donde T es la partición para validación. Se usará la siguiente función objetivo, $F(W) = \alpha \cdot \text{tasa\_clas}(W) + (1 - \alpha) \cdot \text{tasa\_red}(W)$, que se busca maximizar, con $\alpha = 0{,}75$, importante por ser el que pondera la importancia entre acierto y reducción de características de una solución, dando en este caso más importancia al acierto.

Se implementa el algoritmo 1-NN, junto a los algoritmos Greedy Relief y búsqueda local el primer mejor, sobre los datasets *breast-cancer*,*ecoli* y *parkinsons*, previa normalización de los mismos.

Aplicación de algoritmos empleados

2.1. Esquema y representación de las soluciones(práctica 1)

Se ha buscado representar de manera consistente todos y cada uno de los datos presentes en los datasets. Por ello, se declará la estructura *Muestra*, que representa una de las componentes de X:

```
struct Muestra {
    vector<double> caracteristicas;
    string clase;
}
```

con *caracteristicas* representando los valores de las características para una muestra y *clase*, que representa su clase. Nótese que, volviendo a la función p anterior, una instancia de *Muestra* es la tupla (c_i, clase) que le asocia a cierto $x_i \in X$.

Además, se almacenan los atributos, con el tipo correspondiente, con la siguiente estructura, buscando almacenar los datos dados.

```
struct Atributo {
    string nombre;
    string tipo;
}
```

donde, para cierto atributo o característica, se almacena el nombre y su tipo como cadenas de texto.

Para terminar de representar los datos recogidos en un archivo .arff, donde se almacenan las particiones de los datasets, se utiliza la siguiente estructura de datos (sería la representación de X):

```
struct Dataset {
        vector<Atributo> atributos;
        vector<Muestra> muestras;
}
```

con *atributos* hace referencia a la colección de nombre y tipo de las características recogidas, y *muestras* es una colección de muestras. Volviendo a la notación de la descripción del problema, una instancia inicializada de *Dataset* implica que *muestras* tendrá n muestras, correspondiendo a cada una (cada x_i) una clase y vector de características. Por último, se considera necesario hacer una representación consistente de las soluciones de los algoritmos (vectores de pesos). Se usa la siguiente estructura:

```
struct Pesos {
```

```
        vector<double> valores;
}
```

donde *valores* es el vector (de n componentes) de valores de los pesos, todos ellos normalizados a $[0, 1]$.

2.2. Esquema y representación de las soluciones(práctica 2)

Una población estár formada por una serie de cromosomas, que son los candidatos a soluciones. Se ha buscado una representación que use lo menos posible la función objetivo, por lo que al identificarse varias operaciones propias de un cromosoma, se ha representado de la siguiente manera.

```
class Cromosoma{
 private:
  Pesos pesos;
  double valor;
 public:
  Cromosoma(const Dataset& entrenamiento, int N_pesos);
  Cromosoma(const Pesos& pesos, double valor);
  bool esMejorQue(const Cromosoma& crom) const;
  Pesos getPesos() const;
  double getValor() const;
  void mutacion(int indice,const Dataset& entrenamiento);
}
```

donde las estructuras de datos usadas son un objeto *Pesos*, que tiene incluido el vector de pesos asociado (cuyo tamaño es el número de genes del cromosoma), y un valor decimal que representa el valor de fitness para ese vector de pesos. Los métodos asociados a la clase se han implementado conforme se han visto necesarios para el correcto desarrollo de la práctica, y se explicarán más adelante. La decisión de usar una clase para representar al cromosoma, y no un struct como en *Muestra*, ha sido la facilidad de comprensión en el funcionamiento de un cromosoma, que en la naturaleza tiene asociadas operaciones como la mutación.

Finalmente, para representar a una población de cromosomas, se usa la siguiente estructura:

```
struct Poblacion {
    vector<Cromosoma> cromosomas;
};
```

buscando representar de forma intuitiva y comprensible una población para cierto $t \in \mathbb{N}$. En general, una población tendrá $cromosomas.size() * NPesos$ genes en total.

2.3. Operaciones auxiliares(práctica 1)

Se mencionó anteriormente que se usará el algoritmo 1-NN para clasificar una muestra. Por ello, son necesarias funciones para el cálculo de distancias. Como los valores son numéricos, se escoge la métrica euclídea. Se implementan *distanciaEuclidea* y *distanciaEuclideaPonderada*, siendo la primera un caso especial de la segunda (todos los pesos valen 1,0):

Algorithm 1 Calcula la distancia Euclídea entre dos muestras

```
1: function DISTANCIAEUCLIDEA(Muestra m1, Muestra m2)
2:     sum ← 0,0
3:     for i ← 0 to m1.caracteristicas.size() − 1 do
4:         sum ← sum + (m1.caracteristicas[i] − m2.caracteristicas[i])^2
5:     end for
6:     return √sum
7: end function
```

El siguiente código generaliza la distancia euclídea a un vector de pesos cualquiera:

Algorithm 2 Calcula la distancia Euclídea ponderada entre dos muestras

```
1: function DISTANCIAEUCLIDEAPONDERADA(Muestra m1, Muestra m2, Pesos pesos)
2:     sum ← 0,0
3:     for i ← 0 to m1.caracteristicas.size() − 1 do
4:         if pesos.valores[i] > 0,1 then
5:             sum ← sum + pesos.valores[i] ∗ (m1.caracteristicas[i] −
   m2.caracteristicas[i])^2
6:         end if
7:     end for
8:     return √sum
9: end function
```

Se debe remarcar que, aunque se use la raíz cuadrada en el cálculo de la distancia, para el uso que se hace de la distancia euclídea se podría eliminar, pues la función raíz cuadrada sobre los números positivos es estrictamente creciente, y no se pierde información si se suprime.

Estas implementaciones se han realizado para simplificar la programación de la función que hará de clasificador 1-NN. Esta última función, dado un *Dataset* de entrenamiento y otro de validación. La implementación hará uso de la distancia euclídea ponderada, y si fuese necesario usar la distancia euclídea normal, se usarían todos los pesos con valor 1,0.

Algorithm 3 Algoritmo 1-NN

```
1: function ALGORITMO1NN(Muestra m, Dataset entrenamiento, Pesos pesos)
2:     index_min_distancia ← 0
3:     min_distancia ← DISTANCIAEUCLIDEAPONDERADA(m,
4:     entrenamiento.muestras[index_min_distancia], pesos)
5:     distancia ← min_distancia
6:     for i ← 1 to entrenamiento.muestras.size() − 1 do
7:         distancia ← DISTANCIAEUCLIDEAPONDERADA(m, entrenamiento.muestras[i], pesos)
8:         if distancia < min_distancia then
9:             index_min_distancia ← i
10:            min_distancia ← distancia
11:        end if
12:    end for
13:    return entrenamiento.muestras[index_min_distancia].clase
14: end function
```

Por último, se implementa una función auxiliar para simplificar la implementación de los algoritmos, *buscarMuestraMenorDistancia*, con el siguiente pseudocódigo:

Algorithm 4 Buscar Muestra a Menor Distancia

```
1: function BUSCARMUESTRAAMENORDISTANCIA(ds, m)
2:     index_buscado ← 0
3:     distancia_m_buscado ← DISTANCIAEUCLIDEA(ds.muestras[index_buscado], m)
4:     for i ← 1 to ds.muestras.size() − 1 do
5:         distancia_m_probable ← DISTANCIAEUCLIDEA(ds.muestras[i], m)
6:         if distancia_m_probable < distancia_m_buscado then
7:             index_buscado ← i
8:             distancia_m_buscado ← distancia_m_probable
9:         end if
10:    end for
11:    return index_buscado
12: end function
```

2.4. Operaciones auxiliares(práctica 2)

En primera instancia, al representarse a un cromosoma como clase, se le han asociado una serie de métodos, entre los que se encuentran dos constructores por parámetros para la construcción de un cromosoma:

Algorithm 5 Constructor de la clase Cromosoma dado un conjunto de datos

```
1: function CROMOSOMA(Dataset& entrenamiento, int N_pesos)
2:     distribution ← uniform_real_distribution<double>(0.0, 1.0)
3:     for i ← 0 to N_pesos − 1 do
4:         this → pesos.valores.push_back(Random::get(distribution))
5:     end for
6:     this → valor ← fitness(tasaClasificacionLeaveOneOut(entrenamiento, this → pesos), tasaReduccion(this → pesos))
7: end function
```

Algorithm 6 Constructor de la clase Cromosoma con inicialización de atributos

```
1: function CROMOSOMA(const Pesos& pesos, double valor)
2:     this → pesos ← pesos
3:     this → valor ← valor
4: end function
```

el primer constructor usado para la inicialización de cromosomas, y el segundo para facilitar los cálculos propios de cada algoritmo.

Otro método importante es comprobar si un cromosoma es mejor o no según el valor de la función objetivo. Se ha implementado el siguiente método.

Algorithm 7 Comprobar si un Cromosoma es Mejor que Otro

```
1: function ESMEJORQUE(const Cromosoma& crom) const
2:     if this → valor > crom.getValor() then
3:         return true
4:     end if
5:     return false
6: end function
```

que devuelve True si el cromosoma desde el que se llama el método es mejor que el pasado como argumento y False en otro caso.

Se obvian los métodos *Get* por ser de estructura muy simple. La última función que podría comentarse es la asociada a la mutación del cromosoma.

Algorithm 8 Método de Mutación de Cromosoma

```
1: function MUTACION(int indice, const Dataset& entrenamiento)
2:     normal ← normal_distribution<double>(0.0, std::sqrt(VARIANZA))
3:     this → pesos.valores[indice] += Random::get(normal)
4:     if this → pesos.valores[indice] > 1,0 then
5:         this → pesos.valores[indice] ← 1,0
6:     end if
7:     if this → pesos.valores[indice] < 0,0 then
8:         this → pesos.valores[indice] ← 0,0
9:     end if
10:    this → valor ← fitness(tasaClasificacionLeaveOneOut(entrenamiento, this →
       pesos), tasaReduccion(this → pesos))
11: end function
```

Cada vez que es llamado este método, se procederá a mutar el gen de la posición *indice* del cromosoma desde el que se llama al método, según una normal con media nula y varianza 0,3. Al acabar la mutación, se hacen las aproximaciones correspondientes para mantener las restricciones del problema y se vuelve a evaluar la función objetivo.

Para inicializar la población de forma aleatoria para cada algoritmo se implementa la siguiente función.

Algorithm 9 Inicialización de una Población de Cromosomas

```
1: function INICIALIZACION(Poblacion& poblacion, const Dataset& entrenamien-
   to)
2:     N_pesos ← entrenamiento.muestras[0].caracteristicas.size()
3:     for i ← 0 to N_CROMOSOMAS − 1 do
4:         crom ← Cromosoma(entrenamiento, N_pesos)
5:         poblacion.cromosomas.push_back(crom)
6:     end for
7: end function
```

Tanto para los algoritmos genéticos como para los meméticos, se ha buscado simplificar el código. Por lo tanto, se implementan las siguientes funciones.

Algorithm 10 Encuentra el Mejor Cromosoma en una Población

```
1: function MEJORCROMOSOMA(const Poblacion& poblacion)
2:     mejor ← poblacion.cromosomas[0]
3:     ind_mejor ← 0
4:     for i ← 1 to poblacion.cromosomas.size() − 1 do
5:         if poblacion.cromosomas[i].esMejorQue(mejor) then
6:             mejor ← poblacion.cromosomas[i]
7:             ind_mejor ← i
8:         end if
9:     end for
10:    return std::make_pair(mejor, ind_mejor)
11: end function
```

Algorithm 11 Encuentra el Peor Cromosoma en una Población

```
1: function PeorCromosoma(const Poblacion& poblacion)
2:     peor ← poblacion.cromosomas[0]
3:     ind_peor ← 0
4:     for i ← 0 to poblacion.cromosomas.size() − 1 do
5:         if peor.esMejorQue(poblacion.cromosomas[i]) then
6:             peor ← poblacion.cromosomas[i]
7:             ind_peor ← i
8:         end if
9:     end for
10:    return std::make_pair(peor, ind_peor)
11: end function
```

Algorithm 12 Comprueba si un Cromosoma Está Dentro de una Población

```
1: function EstaDentro(const Poblacion& poblacion, const Cromosoma&
   crom)
2:     for i ← 0 to poblacion.cromosomas.size() − 1 do
3:         if crom.getPesos().valores == poblacion.cromosomas[i].getPesos().valores
   then
4:             return true
5:         end if
6:     end for
7:     return false
8: end function
```

La última función será en la que se apoyará el reemplazo generacional para el uso de elitismo.

Finalmente, para obtener los cromosomas sobre los que se aplicará la búsqueda local en dos casos de algoritmos meméticos se usa la siguiente función.

Algorithm 13 Devolver Subconjunto de Cromosomas de una Población

```
1: function DevolverSubconjuntoCromosomas(const Poblacion& poblacion, int tam_subconjunto, bool elMejor)
2:     subconjunto ← new vector<int>()
3:     if elMejor then
4:         misCromos ← poblacion.cromosomas
5:         cromosomasConIndices ← new vector<pares(Cromosoma, int)>()
6:         for i ← 0 to N_CROMOSOMAS − 1 do
7:             cromosomasConIndices.push_back({misCromos[i], i})
8:         end for
9:         Sort cromosomasConIndices by getValor() of each Cromosoma in descending order
10:        for i ← 0 to tam_subconjunto − 1 do
11:            subconjunto.push_back(cromosomasConIndices[i].second)
12:        end for
13:    else
14:        rd ← std::random_device()
15:        gen ← std::mt19937(rd())
16:        distrib_bern ← std::bernoulli_distribution(PROB_MEMETICOS_BL)
17:        for i ← 0 to N_CROMOSOMAS − 1 do
18:            if distrib_bern(gen) then
19:                subconjunto.push_back(i)
20:            end if
21:        end for
22:    end if
23:    return subconjunto
24: end function
```

El método de selección mediante torneo se implementa con

Algorithm 14 Selección de Cromosoma por Torneo

```
1: function Torneo(const Poblacion& poblacion)
2:     K ← 3
3:     TAM_POBL ← poblacion.cromosomas.size()
4:     distrib_entera ← uniform_int_distribution<int>(0, TAM_POBL - 1)
5:     ganador ← poblacion.cromosomas[Random::get(distrib_entera)]
6:     for i ← 0 to K − 1 do
7:         candidato ← poblacion.cromosomas[Random::get(distrib_entera)]
8:         if candidato.esMejorQue(ganador) then
9:             ganador ← candidato
10:        end if
11:    end for
12:    return ganador
13: end function
```

El operador de cruce en BLX es

Algorithm 15 Cruce BLX de Dos Cromosomas

```
1: function CRUCE_BLX(const Cromosoma& c1, const Cromosoma& c2, const
   Dataset& entrenamiento)
2:     descendientes ← vector<Cromosoma>()
3:     p_1 ← c1.getPesos()
4:     p_2 ← c2.getPesos()
5:     h1, h2 ← new Pesos()
6:     for i ← 0 to p_1.valores.size() − 1 do
7:         c_min ← mín(p_1.valores[i], p_2.valores[i])
8:         c_max ← máx(p_1.valores[i], p_2.valores[i])
9:         I ← c_max − c_min
10:        minimo ← c_min − I · ALPHA_BLX
11:        maximo ← c_max + I · ALPHA_BLX
12:        distrib ← uniform_real_distribution<float(minimo, maximo)
13:        h1.valores.push_back(Random::get(distrib))
14:        h2.valores.push_back(Random::get(distrib))
15:        if h1.valores[i] > 1 then
16:            h1.valores[i] ← 1,0
17:        end if
18:        if h1.valores[i] < 0 then
19:            h1.valores[i] ← 0,0
20:        end if
21:        if h2.valores[i] > 1 then
22:            h2.valores[i] ← 1,0
23:        end if
24:        if h2.valores[i] < 0 then
25:            h2.valores[i] ← 0,0
26:        end if
27:    end for
28:    fitness1 ← fitness(tasaClasificacionLeaveOneOut(entrenamiento, h1), tasaReduccion(h1))
29:    fitness2 ← fitness(tasaClasificacionLeaveOneOut(entrenamiento, h2), tasaReduccion(h2))
30:    descendientes.push_back(Cromosoma(h1, fitness1))
31:    descendientes.push_back(Cromosoma(h2, fitness2))
32:    assert(descendientes.size() == 2)
33:    return descendientes
34: end function
```

El siguiente es el cruce aritmético

Algorithm 16 Cruce Aritmético de Dos Cromosomas

```
1: function CRUCE_ARIT(const Cromosoma& c1, const Cromosoma& c2, const
   Dataset& entrenamiento)
2:     v_mutar1, v_mutar2 ← new Pesos()
3:     p_1 ← c1.getPesos()
4:     p_2 ← c2.getPesos()
5:     distr ← uniform_real_distribution<double>(0.0, 1.0)
6:     alpha_arit ← Random::get(distr)
7:     for i ← 0 to p_1.valores.size() − 1 do
8:         v_mutar1.valores.push_back(alpha_arit · p_1.valores[i] + (1 −
   alpha_arit) · p_2.valores[i])
9:         v_mutar2.valores.push_back(alpha_arit · p_2.valores[i] + (1 −
   alpha_arit) · p_1.valores[i])
10:        if v_mutar1.valores[i] > 1 then
11:            v_mutar1.valores[i] ← 1,0
12:        end if
13:        if v_mutar1.valores[i] < 0 then
14:            v_mutar1.valores[i] ← 0,0
15:        end if
16:        if v_mutar2.valores[i] > 1 then
17:            v_mutar2.valores[i] ← 1,0
18:        end if
19:        if v_mutar2.valores[i] < 0 then
20:            v_mutar2.valores[i] ← 0,0
21:        end if
22:    end for
23:    fitness_mutar1 ← fitness(tasaClasificacionLeaveOneOut(entrenamiento,
24:    v_mutar1), tasaReduccion(v_mutar1))
25:    fitness_mutar2 ← fitness(tasaClasificacionLeaveOneOut(entrenamiento,
26:
27:    crom1 ← Cromosoma(v_mutar1, fitness_mutar1)
28:    crom2 ← Cromosoma(v_mutar2, fitness_mutar2)
29:    hijos ← vector<Cromosoma>()
30:    hijos.push_back(crom1)
31:    hijos.push_back(crom2)
32:    assert(hijos.size() == 2)
33:    return hijos
34: end function
```

Por último, para poder realizar el estudio de convergencia de algunos datasets, su usa la siguiente función:

Algorithm 17 Escribir Datos de Fitness para la Convergencia en un Archivo CSV

```
function EscribirFitnessParaConvergencia(num_part, fitness, n_generacion, algoritmo, nombre_dataset)
    filename ← "fitness_Partic_ + to_string(num_part) + "_algoritmo_ + algoritmo + "_dataset_ + nombre_dataset + ".csv
    file ← new ofstream()
    file.open(filename, ios::out|ios::app)
    if not file.is_open() then
        cerr ← .Error al abrir el archivo: - filename
        return
    end if
    file.seekp(0, ios::end)                                  ▷ Mover al final del archivo
    if file.tellp() == 0 then
        file ← "Generación,Fitness"    ▷ Escribir encabezados si el archivo está vacío
    end if
    file ← n_generacion + "," + fitness + "\n"  ▷ Escribir los datos de fitness
    file.close()
end function
```

2.5. Función objetivo y estadísticos

Tal y como se mencionó en la descripción del problema, se busca maximizar el valor de la función objetivo, llamada *fitness*, con $\alpha = 0{,}75$. Además, se usan estadísticos complementarios para calcular la reducción y la tasa de acierto según unos datos de entrenamiento, validación y vector de pesos. A continuación se muestra el pseudocódigo correspondiente.

Algorithm 18 Tasa de Clasificación

```
function tasaClasificacion(Dataset entrenamiento, Dataset test, Pesos pesos)
    n_bien_clasificada ← 0
    prediccion ←
    for i ← 0 to test.muestras.size() − 1 do
        prediccion ← ALGORITMO1NN(test.muestras[i], entrenamiento, pesos)
        if prediccion = test.muestras[i].clase then
            n_bien_clasificada ← n_bien_clasificada + 1
        end if
    end for
    return (100,0 × convertir a double(n_bien_clasificada))/ convertir a double(test.muestras.size())
end function
```

Algorithm 19 Tasa de Reducción

```
1: function TasaReduccion(Pesos pesos)
2:     UMBRAL ← 0,1
3:     n_menor_umbral ← 0
4:     for i ← 0 to pesos.valores.size() − 1 do
5:         if pesos.valores[i] < UMBRAL then
6:             n_menor_umbral ← n_menor_umbral + 1
7:         end if
8:     end for
9:     return (100,0 × n_menor_umbral)/(1,0 × pesos.valores.size())
10: end function
```

Algorithm 20 Calcula el valor de fitness

```
1: function Fitness(tasa_cla, tasa_red)
2:     return α × tasa_cla + (1,0 − α) × tasa_red
3: end function
```

Descripción de algoritmos implementados

3.1. Algoritmos de la práctica 1

A continuación se muestran los pseudocódigos de los algoritmos implementados para la resolución del problema: búsqueda local y greedy RELIEF. Se busca un vector de pesos que maximice la función objetivo.

3.1.1. Búsqueda local

Se escoge la estrategia de el primero mejor para obtener las distintas soluciones para el vector de pesos. Por iteración se muta una única componente del vector de pesos de forma aleatoria sin repetir hasta que no se hayan mutado todas las componentes del vector. Más específicamente, para n pesos (inicializados con valores de una distribución uniforme sobre $[0,1]$, $U[0,1]$), se escoge aleatoriamente un índice $0 \leq i \leq n-1$ con $i \notin H$, donde H es el conjunto de índices ya mutados (este conjunto, cuando no se ha empezado a iterar, o se han mutado todos los elementos del vector de pesos, es $H = \{\}$. Ahora, escogido un índice i, si $pesos.valores[i]$ es el peso correspondiente, entonces la mutación es $pesos.valores[i] + X$ con $X \sim \mathcal{N}(0,0{,}3)$ (media 0 varianza 0.3). En pseudocódigo se tiene

Algorithm 21 Mutación con Búsqueda Local

```
1: function MUTACIONBL(pesos, index, normal)
2:     pesos.valores[index] ← pesos.valores[index] + RANDOM::GET(normal)
3:     if pesos.valores[index] > 1,0 then
4:         pesos.valores[index] ← 1,0
5:     end if
6:     if pesos.valores[index] < 0,0 then
7:         pesos.valores[index] ← 0,0
8:     end if
9: end function
```

donde, tras mutar, se trunca a 1,0 si $pesos.valores[i] > 1{,}0$ o a 0,0 si $pesos.valores[i] < 0{,}0$.

La forma de tomar los índices de forma aleatoria sin repetir se muestra en la llamada a la siguiente función, donde para optimizar el tiempo solo se recorre una vez el vector de índices, haciendo siempre una mezcla aleatoria.

Con *Random::shuffle* se baraja aleatoriamente el vector de índices según la semilla introducida en la llamada al programa.

El pseudocódigo completo de la función de búsqueda local se muestra a continuación, desde la creación de la solución inicial hasta la mutación y elección de nuevas

Algorithm 22 Mezclar Índices

```
1: function MEZCLARINDICES(n_pesos, indices, inicializar)
2:     if inicializar then
3:         for i ← 0 to n_pesos − 1 do
4:             indices[i] ← i
5:         end for
6:     end if
7:     RANDOM::SHUFFLE(indices)
8: end function
```

soluciones, con criterios de parada $MAXIMO_{=}15000$ evaluaciones de la función objetivo o la generación de $20 \cdot n\_caracteristicas = LIMITE$ vecinos sin mejora:

Algorithm 23 Búsqueda Local para Optimización

```
1: function BusquedaLocal(const Dataset& entrenamiento)
2:     solucion_actual ← new Pesos()
3:     distribution ← uniform_real_distribution<double>(0.0, 1.0)
4:     normal ← normal_distribution<double>(0.0, std::sqrt(VARIANZA))
5:     for i ← 0 to entrenamiento.muestras[0].caracteristicas.size() − 1 do
6:         solucion_actual.valores.push_back(Random::get(distribution))
7:     end for
8:     tasa_cla_sol_actual ← tasaClasificacionLeaveOneOut(entrenamiento, solucion_actual)
9:     tasa_red_sol_actual ← tasaReduccion(solucion_actual)
10:    fitness_actual ← fitness(tasa_cla_sol_actual, tasa_red_sol_actual)
11:    n_vecinos_generados_sin_mejora ← 0
12:    num_caracteristicas ← entrenamiento.muestras[0].caracteristicas.size()
13:    indexes ← vector<int>(num_caracteristicas)
14:    hubo_mejora ← false
15:    num_iterac ← 0
16:    mezclarIndices(num_caracteristicas, indexes, true)
17:    while num_iterac < MAXIMO_ITERACIONES and
       n_vecinos_generados_sin_mejora < LIMITE · num_caracteristicas
       do
18:        hubo_mejora ← false
19:        for j ← 0 to indexes.size() − 1 and not hubo_mejora and
           num_iterac < MAXIMO_ITERACIONES do
20:            vecino ← solucion_actual
21:            mutacionBL(vecino, indexes[j], normal)
22:            tasa_cla_vecino ← tasaClasificacionLeaveOneOut(entrenamiento, vecino)
23:            tasa_red_vecino ← tasaReduccion(vecino)
24:            fitness_vecino ← fitness(tasa_cla_vecino, tasa_red_vecino)
25:            num_iterac ← num_iterac + 1
26:            if fitness_vecino > fitness_actual then
27:                fitness_actual ← fitness_vecino
28:                solucion_actual ← vecino
29:                hubo_mejora ← true
30:            else
31:                n_vecinos_generados_sin_mejora ←
                   n_vecinos_generados_sin_mejora + 1
32:            end if
33:        end for
34:        if hubo_mejora then
35:            n_vecinos_generados_sin_mejora ← 0
36:            mezclarIndices(num_caracteristicas, indexes, false)
37:        end if
38:    end while
39:    return std::make_pair(solucion_actual, fitness_actual)
40: end function
```

3.1.2. Greedy RELIEF

El algoritmo Greedy RELIEF es un algoritmo sencillo, clasificado como voraz. Dada una muestra, se busca un enemigo más cercano (muestra más cercana con clase distinta) y un amigo más cercano (muestra más cercana con misma clase distinta a la muestra). Con ellos, se aplica una fórmula con las distancias de las caracterís-

ticas para calcular la respectiva coordenada del vector de pesos (el vector de pesos inicialmente tiene todas sus componentes a 0,0). Tras este cálculo, se normalizan los valores del vector de pesos (pues se pueden obtener valores fuera del intervalo $[0, 1]$).

Se muestran los correspondientes pseudocódigos para la implementación del algoritmo:

Algorithm 24 Buscar Enemigo Más Cercano

```
1: function BUSCARENEMIGOMASCERCANO(entrenamiento, m)
2:     candidatosEnemigo ← new Dataset
3:     for i ← 0 to size(entrenamiento.muestras) − 1 do
4:         if entrenamiento.muestras[i].clase ≠ m.clase then
5:             candidatosEnemigo.muestras.push_back(entrenamiento.muestras[i])
6:         end if
7:     end for
8:     index_enemigo_buscado ←
9:     BUSCARMUESTRAAMENORDISTANCIA(candidatosEnemigo, m)
10:    return candidatosEnemigo.muestras[index_enemigo_buscado]
11: end function
```

Algorithm 25 Buscar Amigo Más Cercano

```
1: function BUSCARAMIGOMASCERCANO(entrenamiento, m)
2:     candidatosAmigo ← nuevo Dataset vacío
3:     amigo ← nueva Muestra vacía
4:     index_amigo_buscado ← −1
5:     for i ← 0 to entrenamiento.muestras.size() − 1 do
6:         if entrenamiento.muestras[i].clase = m.clase and m.caracteristicas ≠
   entrenamiento.muestras[i].caracteristicas then
7:             candidatosAmigo.muestras.push_back(entrenamiento.muestras[i])
8:         end if
9:     end for
10:    if candidatosAmigo.muestras.size() > 0 then
11:        index_amigo_buscado ← BUSCARMUESTRAAMENORDISTAN-
   CIA(candidatosAmigo, m)
12:        amigo ← candidatosAmigo.muestras[index_amigo_buscado]
13:    end if
14:    return amigo
15: end function
```

Se usa una función auxiliar, *buscarMuestraMenorDistancia*, para hacer más legible el código. Su implementación se mostró anteriormente.

En la búsqueda del amigo más cercano, puede ocurrir que en el conjunto de entrenamiento haya una muestra con una clase que no tengan otras muestras, lo que hace que no se pueda encontrar ese amigo (ocurre en el dataset *ecoli*). Para solventar el problema, en el siguiente código (del algoritmo Greedy RELIEF) se hace la comprobación para evitar un error de ejecución.

Algorithm 26 Algoritmo Greedy Relief

```
1: function GreedyRelief(entrenamiento)
2:     solucion_actual ← nuevo Pesos
3:     for i ← 0 to entrenamiento.muestras[0].caracteristicas.size() − 1 do
4:         solucion_actual.valores.push_back(0,0)
5:     end for
6:     for i ← 0 to entrenamiento.muestras.size() − 1 do
7:         enemigoMasCercano ← buscarEnemigoMasCercano(entrenamiento, entrenamiento.muestras[i])
8:         amigoMasCercano ← buscarAmigoMasCercano(entrenamiento, entrenamiento.muestras[i])
9:         if amigoMasCercano.caracteristicas.size() > 0 then
10:            for j ← 0 to amigoMasCercano.caracteristicas.size() − 1 do
11:                solucion_actual.valores[j] ← solucion_actual.valores[j] + |entrenamiento.muestras[i].caracteristicas[j]−enemigoMasCercano.caracteristicas[j]|− |entrenamiento.muestras[i].caracteristicas[j]−amigoMasCercano.caracteristicas[j]|
12:            end for
13:        end if
14:    end for
15:    w_max ← máximo elemento en solucion_actual.valores
16:    for j ← 0 to solucion_actual.valores.size() − 1 do
17:        if solucion_actual.valores[j] < 0,0 then
18:            solucion_actual.valores[j] ← 0,0
19:        else
20:            solucion_actual.valores[j] ← solucion_actual.valores[j]/w_max
21:        end if
22:    end for
23:    tasa_cla ← tasaClasificacionLeaveOneOut(entrenamiento, solucion_actual)
24:    tasa_red ← tasaReduccion(solucion_actual)
25:    valor_fitness ← fitness(tasa_cla, tasa_red)
26:    return make_pair(solucion_actual, valor_fitness)
27: end function
```

3.2. Algoritmos de la práctica 2

3.2.1. Algoritmos Genéticos Generacionales

A grandes rasgos, se aplica torneo para elegir a los padres sobre los que se aplicará cruce tantas veces como número de cromosomas haya en la población. A continuación, debido a que puede ser un gasto innceseario generar tantos números aleatorios como candidatos a cruce haya, se procede a calcular el número esperado de cruces (esperanza matemática en la distribución de cruces). Se aplicarán tantos cruces como indique tal valor. Acto seguido, se procede a realizar las mismas operaciones para el número de mutaciones esperadas de los hijos. Tanto el hijo a mutar como el gen que se mutará según una normal se eligen aleatoriamente según una distribución uniforme discreta. Al final, se aplica reemplazo generacional con uso de elitismo (si el mejor padre no está en la nueva población, este sustituirá al peor hijo si fuese mejor). El enfoque de reemplazo seguido se implementa en el siguiente pseudocódigo.

Algorithm 27 Reemplazo Generacional en Poblaciones

```
1: function ReemplazoGeneracional(const Poblacion& poblacion, const Poblacion& hijos)
2:     PoblReemplazo ← hijos
3:     mejorPadre ← mejorCromosoma(poblacion)
4:     peorHijo ← peorCromosoma(hijos)
5:     if !estaDentro(hijos, mejorPadre.first) and
6:         mejorPadre.first.esMejorQue(peorHijo.first) then
7:         PoblReemplazo.cromosomas[peorHijo.second] ← mejorPadre.first
8:     end if
9:     return PoblReemplazo
10: end function
```

Respecto al número esperado de cruces y de mutaciones, las fórmulas aplicadas son las siguientes:

- Número esperado de cruces: $\frac{P_c \cdot N}{2}$, donde P_c es la probabilidad de cruce.
- Número esperado de mutaciones: $P_m \cdot NCROMOSOMAS \cdot NPESOS$, donde P_m es la probabilidad de mutación por gen, por lo que es importante tener en cuenta que la distribución tenida en cuenta es por gen, no por cromosoma

Ambos valores se redondean al alza siempre que el valor dado no sea un entero.

Algoritmo Genético Generacional - BLX

La implementación del algoritmo genético generacional, en pseudocódigo, es la siguiente. Se apoya en las funciones auxiliares anteriormente expuestas, como la forma de cruce, donde para cada gen $i \in \{1, ..., NPESOS\}$ se toman dos hijos de forma aleatoria en $[\text{mín}(crom1[i-1], crom2[i-1]) - I \cdot \alpha, \text{máx}(crom1[i-1], crom2[i-1]) + I \cdot \alpha]$ para cada gen, con $\alpha = 0{,}3$, $I = \text{máx}(crom1[i-1], crom2[i-1]) - \text{mín}(crom1[i-1], crom2[i-1])$.

Algorithm 28 Algoritmo Genético Generacional con Cruce BLX (AGG_BLX)

```
1: function AGG_BLX(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     evaluaciones ← 0
4:     num_generaciones ← 1
5:     INICIALIZACION(poblacion, entrenamiento)
6:     evaluaciones += poblacion.cromosomas.size()
7:     num_cruces_esperanza_mat ← round(PROB_CRUCE_AGG · (poblacion.cromosomas.size()/2))
8:     if num_cruces_esperanza_mat < 1 then
9:         num_cruces_esperanza_mat ← 1
10:    end if
11:    numGenes ← poblacion.cromosomas.size() · poblacion.cromosomas[0].getPesos().valores.size()
12:    num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_GENETICOS · numGenes)
13:    if num_mutaciones_esp_mat < 1 then
14:        num_mutaciones_esp_mat ← 1
15:    end if
16:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
17:    getPesos().valores.size() − 1)
18:    distrib_cromos_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas.
19:    size() − 1)
20:    while evaluaciones < MAX_EVALUACIONES do
21:        intermedio ← new Poblacion()
22:        if nombre ≠ BCANCER then
23:            escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
24:            getValor(), num_generaciones, "AGG_BLX", nombre)
25:        end if
26:        for i ← 0 to N_CROMOSOMAS − 1 do
27:            intermedio.cromosomas.push_back(torneo(poblacion))
28:        end for
29:        RANDOM::SHUFFLE(intermedio.cromosomas)
30:        for i ← 0 to 2 · num_cruces_esperanza_mat − 1 by 2 do
31:            desc_BLX ← Cruce_BLX(intermedio.cromosomas[i], intermedio.cromosomas[i+
    1], entrenamiento)
32:            intermedio.cromosomas[i] ← desc_BLX[0]
33:            intermedio.cromosomas[i + 1] ← desc_BLX[1]
34:            evaluaciones += 2
35:        end for
36:        for i ← 0 to num_mutaciones_esp_mat − 1 do
37:            index_cromosoma_mutar ← Random::get(distrib_cromos_uniforme)
38:            index_gen_mutar ← Random::get(distrib_indices_uniforme)
39:            intermedio.cromosomas[index_cromosoma_mutar].mutacion(index_gen_mutar,
40:            entrenamiento)
41:            evaluaciones += 1
42:        end for
43:        poblacion ← reemplazoGeneracional(poblacion, intermedio)
44:        num_generaciones += 1
45:    end while
46:    mejorCrom ← mejorCromosoma(poblacion).first
47:    return std::make_pair(mejorCrom.getPesos(), mejorCrom.getValor())
48: end function
```

Algoritmo Genético Generacional - CA

El esquema de resolución en este algoritmo es similar al anterior, con la única diferencia de la forma de cruzar los padres elegidos por torneo, que devolverá dos hijos, generando cierto $\alpha \in [0, 1]$ aleatorio cada vez que se haga un cruce aritmético aleatorio. Esto es, cada gen del hijo se calcula de la manera $h1[i-1] = \alpha \cdot padre1[i-1] + (1-\alpha) \cdot padre2[i-1]$, $h2[i-1] = \alpha \cdot padre2[i-1] + (1-\alpha) \cdot padre1[i-1]$ $\forall i \in \{1, ..., NPESOS\}$.

Algorithm 29 Algoritmo Genético Generacional con Cruce Aritmético (AGG_Arit)

```
1: function AGG_ARIT(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     evaluaciones ← 0
4:     num_generaciones ← 1
5:     INICIALIZACION(poblacion, entrenamiento)
6:     evaluaciones += poblacion.cromosomas.size()
7:     num_cruces_esperanza_mat ← round(PROB_CRUCE_AGG · (poblacion.cromosomas.size()/2))
8:     numGenes ← poblacion.cromosomas.size() · poblacion.cromosomas[0].getPesos().valores.size()
9:     num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_GENETICOS·
10:    numGenes)
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distrib_cromos_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas.
14:    size() − 1)
15:    while evaluaciones < MAX_EVALUACIONES do
16:        intermedio ← new Poblacion()
17:        if nombre ≠ BCANCER then
18:            escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
19:            getValor(), num_generaciones, "AGG_CA", nombre)
20:        end if
21:        for i ← 0 to N_CROMOSOMAS − 1 do
22:            intermedio.cromosomas.push_back(torneo(poblacion))
23:        end for
24:        RANDOM::SHUFFLE(intermedio.cromosomas)
25:        for i ← 0 to 2 · num_cruces_esperanza_mat − 2 by 2 do
26:            desc_Arit ← Cruce_Arit(intermedio.cromosomas[i], intermedio.
27:            cromosomas[N_CROMOSOMAS − i − 1], entrenamiento)
28:            intermedio.cromosomas[i] ← desc_Arit[0]
29:            intermedio.cromosomas[i + 1] ← desc_Arit[1]
30:            evaluaciones += 2
31:        end for
32:        for i ← 0 to num_mutaciones_esp_mat − 1 do
33:            index_cromosoma_mutar ← Random::get(distrib_cromos_uniforme)
34:            index_gen_mutar ← Random::get(distrib_indices_uniforme)
35:            intermedio.cromosomas[index_cromosoma_mutar].mutacion(index_gen_mutar,
36:            entrenamiento)
37:            evaluaciones += 1
38:        end for
39:        poblacion ← reemplazoGeneracional(poblacion, intermedio)
40:        num_generaciones += 1
41:    end while
42:    mejorCrom ← mejorCromosoma(poblacion).first
43:    return std::make_pair(mejorCrom.getPesos(), mejorCrom.getValor())
44: end function
```

3.2.2. Algoritmo Genético Estacionario

La forma de proceder en el caso estacionario será similar al modelo generacional, con la diferencia de las veces que se aplica el torneo (en este caso, solo dos veces) y la forma de hacer el reemplazo, donde los dos hijos compiten con los dos peores cromosomas de la población, manteniendo los dos mejores.

Algorithm 30 Reemplazo Estacionario en Población

```
1: function REEMPLAZOESTACIONARIO(Poblacion& poblacion, const vector<Cromosoma>& hijos)
2:     assert(hijos.size() == 2)                          ▷ Asegura que sólo hay dos hijos
3:     hijo1 ← hijos[0]
4:     hijo2 ← hijos[1]
5:     peorCrom1 ← PEORCROMOSOMA(poblacion)               ▷ Encuentra el peor cromosoma
6:     peorCrom2 ← MAKE_PAIR(poblacion.cromosomas[0], 0)  ▷ Inicializa el segundo peor
7:     if peorCrom1.second == 0 then
8:         peorCrom2 ← MAKE_PAIR(poblacion.cromosomas[1], 1)  ▷ Evita el primer cromosoma si es el peor
9:     end if
10:    for i ← 0 to N_CROMOSOMAS − 1 do
11:        if i ≠ peorCrom1.second then
12:            if ESMEJORQUE(peorCrom2.first, poblacion.cromosomas[i]) then
13:                peorCrom2.first ← poblacion.cromosomas[i]
14:                peorCrom2.second ← i
15:            end if
16:        end if
17:    end for
18:    croms ← {peorCrom1.first, peorCrom2.first, hijo1, hijo2}  ▷ Crea un conjunto para clasificar
19:    SORT(croms, esMejorQue)                            ▷ Ordena por calidad
20:    SORT(croms, getValor)          ▷ Ordena por valor fitness de mayor a menor
21:    poblacion.cromosomas[peorCrom1.second] ← croms[0]
22:    poblacion.cromosomas[peorCrom2.second] ← croms[1]
23: end function
```

Algoritmo Genético Estacionario - BLX

Las características propias de este modelo es el indicado anteriormente, haciendo el cruce según el método BLX.

Algorithm 31 Algoritmo Genético Estacionario con Cruce BLX (AGE_BLX)

```
1: function AGE_BLX(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     if nombre ≠ BCANCER then
4:         escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
5:         getValor(), num_generaciones, "AGE_BLX", nombre)
6:     end if
7:     evaluaciones ← 0
8:     num_generaciones ← 1
9:     INICIALIZACION(poblacion, entrenamiento)
10:    evaluaciones += poblacion.cromosomas.size()
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distr_para_cruce ← uniform_int_distribution<int(0, 1)
14:    while evaluaciones < MAX_EVALUACIONES do
15:        intermedio ← new Poblacion()
16:        for i ← 0 to MAX_GANADORES − 1 do
17:            intermedio.cromosomas.push_back(torneo(poblacion))
18:        end for
19:        cruces ← Cruce_BLX(intermedio.cromosomas[0], intermedio.cromosomas[1],
20:        entrenamiento)
21:        evaluaciones += 2
22:        numGenes ← cruces.size() · cruces[0].getPesos().valores.size()
23:        num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_GENETIC
    numGenes)
24:        RANDOM::SHUFFLE(cruces)
25:        for i ← 0 to num_mutaciones_esp_mat − 1 do
26:            index_cruce ← Random::get(distr_para_cruce)
27:            index_mutar ← Random::get(distrib_indices_uniforme)
28:            cruces[index_cruce].mutacion(index_mutar, entrenamiento)
29:            evaluaciones += 1
30:        end for
31:        REEMPLAZOESTACIONARIO(poblacion, cruces)
32:        num_generaciones += 1
33:    end while
34:    mejorCrom ← mejorCromosoma(poblacion).first
35:    return std::make_pair(mejorCrom.getPesos(), mejorCrom.getValor())
36: end function
```

Algoritmo Genético Estacionario - CA

Mismas características que el algoritmo genético estacionario BLX pero usando el operador aritmético para el cruce.

Algorithm 32 Algoritmo Genético Estacionario con Cruce Aritmético (AGE_Arit)

```
1: function AGE_ARIT(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     if nombre ≠ BCANCER then
4:         escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
5:         getValor(), num_generaciones, "AGE_CA", nombre)
6:     end if
7:     evaluaciones ← 0
8:     num_generaciones ← 1
9:     INICIALIZACION(poblacion, entrenamiento)
10:    evaluaciones += poblacion.cromosomas.size()
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distr_para_cruce ← uniform_int_distribution<int(0, 1)
14:    while evaluaciones < MAX_EVALUACIONES do
15:        intermedio ← new Poblacion()
16:        for i ← 0 to MAX_GANADORES − 1 do
17:            intermedio.cromosomas.push_back(torneo(poblacion))
18:        end for
19:        cruces ← Cruce_Arit(intermedio.cromosomas[0], intermedio.cromosomas[1],
20:        entrenamiento)
21:        evaluaciones += 2
22:        numGenes ← cruces.size() · cruces[0].getPesos().valores.size()
23:        num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_GENETIC
   numGenes)
24:        RANDOM::SHUFFLE(cruces)
25:        for i ← 0 to num_mutaciones_esp_mat − 1 do
26:            index_cruce ← Random::get(distr_para_cruce)
27:            index_mutar ← Random::get(distrib_indices_uniforme)
28:            cruces[index_cruce].mutacion(index_mutar, entrenamiento)
29:            evaluaciones += 1
30:        end for
31:        REEMPLAZOESTACIONARIO(poblacion, cruces)
32:        num_generaciones += 1
33:    end while
34:    mejorCrom ← mejorCromosoma(poblacion).first
35:    return std::make_pair(mejorCrom.getPesos(), mejorCrom.getValor())
36: end function
```

3.2.3. Algoritmos Meméticos

Algoritmo Memético All

El primer algoritmo memético es la implementación del algoritmo genético que mejores resultados ha dado (se ha usado el algoritmo genético generacional de cruce aritmético pues, aunque los resultados sean muy similares, en el conjunto más grande da resultados algo mejores, pero se entiende que se podría usar cualquiera de los do, aunque si no se tiene en cuenta el dataset *Breast-Cancer*, habría sido más conveniente escoger $AGG - BLX$, pese a que a grandes rasgos no importaría debido a la poca diferencia entre los fitness). Se aplicará búsqueda local cada diez generaciones a todos los cromosomas. El pseudocódigo es (se usa una implementación de búsqueda

local análoga a la mostrada anteriormente y se escribe fitness para la covnergencia cada 15 generaciones para evitar escribir excesivamente en disco),

Algorithm 33 Algoritmo Memético con Aplicación de Búsqueda Local a Todos los Cromosomas (AM_All)

```
1: function AM_ALL(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     evaluaciones ← 0
4:     num_generaciones ← 1
5:     INICIALIZACION(poblacion, entrenamiento)
6:     evaluaciones += poblacion.cromosomas.size()
7:     num_cruces_esperanza_mat ← round(PROB_CRUCE_MEMETICOS·
   (poblacion.cromosomas.size()/2))
8:     numGenes ← poblacion.cromosomas.size() ·
   poblacion.cromosomas[0].getPesos().valores.size()
9:     num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_
10:    MEMETICOS · numGenes)
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distrib_cromos_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas.size
14:    −1)
15:    while evaluaciones < MAX_EVALUACIONES do
16:        intermedio ← new Poblacion()
17:        if nombre ≠ BCANCER and num_generaciones − 1 mód 15 == 0
   then
18:            escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
19:            getValor(), num_generaciones, "AM_All", nombre)
20:        end if
21:        for i ← 0 to N_CROMOSOMAS − 1 do
22:            intermedio.cromosomas.push_back(torneo(poblacion))
23:        end for
24:        RANDOM::SHUFFLE(intermedio.cromosomas)
25:        for i ← 0 to 2 · num_cruces_esperanza_mat − 2 by 2 do
26:            desc_Arit ← Cruce_Arit(intermedio.cromosomas[i], intermedio.
27:            cromosomas[N_CROMOSOMAS − i − 1], entrenamiento)
28:            intermedio.cromosomas[i] ← desc_Arit[0]
29:            intermedio.cromosomas[i + 1] ← desc_Arit[1]
30:            evaluaciones += 2
31:        end for
32:        for i ← 0 to num_mutaciones_esp_mat − 1 do
33:            index_cromosoma_mutar ← Random::get(distrib_cromos_uniforme)
34:            index_gen_mutar ← Random::get(distrib_indices_uniforme)
35:            intermedio.cromosomas[index_cromosoma_mutar].mutacion(index_gen_mutar,
36:            entrenamiento)
37:            evaluaciones += 1
38:        end for
39:        REEMPLAZOGENERACIONAL(poblacion, intermedio)
40:        num_generaciones += 1
41:        if num_generaciones mód N_GENERACIONES == 0 then
42:            for i ← 0 to N_CROMOSOMAS − 1 do
43:                evaluaciones += busquedaLocalPractica2(entrenamiento, poblacion.
44:                cromosomas[i])
45:            end for
46:        end if
47:    end while
48:    mejorCrom ← mejorCromosoma(poblacion).first
49:    return std::make_pair(mejorCrom.getPesos(), mejorCrom.getValor())
50: end function
```

Algoritmo Memético Random

De la misma manera, cada diez generaciones, se aplica búsqueda local a los cromosomas elegidos de forma aleatoria bajo cierta probabilidad. Esta probabilidad ha sido modelada según una distribución de Bernoulli, pues el problema es elegir o no elegir un cromosoma para aplicar búsqueda local. El pseudocódigo es

Algorithm 34 Algoritmo Memético Aleatorio con Aplicación de Búsqueda Local (AM_Rand)

```
1: function AM_Rand(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     evaluaciones ← 0
4:     num_generaciones ← 1
5:     Inicializacion(poblacion, entrenamiento)
6:     evaluaciones += poblacion.cromosomas.size()
7:     num_cruces_esperanza_mat ← round(PROB_CRUCE_MEMETICOS·
   (poblacion.cromosomas.size()/2))
8:     numGenes ← poblacion.cromosomas.size() ·
   poblacion.cromosomas[0].getPesos().valores.size()
9:     num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_
10:    MEMETICOS · numGenes)
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distrib_cromos_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas.
14:    size() − 1)
15:    while evaluaciones < MAX_EVALUACIONES do
16:        intermedio ← new Poblacion()
17:        if nombre ≠ BCANCER and num_generaciones − 1 mód 15 == 0
   then
18:            escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
19:            getValor(), num_generaciones, "AM_Rand", nombre)
20:        end if
21:        for i ← 0 to N_CROMOSOMAS − 1 do
22:            intermedio.cromosomas.push_back(torneo(poblacion))
23:        end for
24:        Random::shuffle(intermedio.cromosomas)
25:        for i ← 0 to 2 · num_cruces_esperanza_mat − 2 by 2 do
26:            desc_Arit ← Cruce_Arit(intermedio.cromosomas[i], intermedio.
27:            cromosomas[N_CROMOSOMAS − i − 1], entrenamiento)
28:            intermedio.cromosomas[i] ← desc_Arit[0]
29:            intermedio.cromosomas[i + 1] ← desc_Arit[1]
30:            evaluaciones += 2
31:        end for
32:        for i ← 0 to num_mutaciones_esp_mat − 1 do
33:            index_cromosoma_mutar ← Random::get(distrib_cromos_uniforme)
34:            index_gen_mutar ← Random::get(distrib_indices_uniforme)
35:            intermedio.cromosomas[index_cromosoma_mutar].mutacion(index_gen_mutar,
36:            entrenamiento)
37:            evaluaciones += 1
38:        end for
39:        ReemplazoGeneracional(poblacion, intermedio)
40:        num_generaciones += 1
41:        if num_generaciones mód N_GENERACIONES == 0 then
42:            posic_alterar ← devolverSubconjuntoCromosomas(poblacion)
43:            for i ← 0 to posic_alterar.size() − 1 do
44:                evaluaciones += busquedaLocalPractica2(entrenamiento, poblacion.
45:                cromosomas[posic_alterar[i]])
46:            end for
47:        end if
48:    end while
49:    mejorCrom ← mejorCromosoma(poblacion).first
```

Algoritmo Memético Best

Procedimiento análogo al caso anterior, con la diferencia de que cada diez generaciones se aplica búsqueda local a los $0{,}1 \cdot N$ mejores cromosomas presente, donde N es el número de cromosomas.

Algorithm 35 Algoritmo Memético con Aplicación de Búsqueda Local al Mejor Subconjunto (AM_Best)

```
1: function AM_Best(const Dataset& entrenamiento, string nombre, int n_part)
2:     poblacion ← new Poblacion()
3:     evaluaciones ← 0
4:     num_generaciones ← 1
5:     Inicializacion(poblacion, entrenamiento)
6:     evaluaciones += poblacion.cromosomas.size()
7:     num_cruces_esperanza_mat ← round(PROB_CRUCE_MEMETICOS·
   (poblacion.cromosomas.size()/2))
8:     numGenes ← poblacion.cromosomas.size() ·
   poblacion.cromosomas[0].getPesos().valores.size()
9:     num_mutaciones_esp_mat ← round(PROB_MUTAR_INDIVIDUO_
10:    MEMETICOS · numGenes)
11:    distrib_indices_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas[0].
12:    getPesos().valores.size() − 1)
13:    distrib_cromos_uniforme ← uniform_int_distribution<int(0, poblacion.cromosomas.
14:    size() − 1)
15:    while evaluaciones < MAX_EVALUACIONES do
16:        intermedio ← new Poblacion()
17:        if nombre ≠ BCANCER and num_generaciones − 1 mód 15 == 0
   then
18:            escribirFitnessParaConvergencia(n_part, mejorCromosoma(poblacion).first.
19:            getValor(), num_generaciones, "AM_Best", nombre)
20:        end if
21:        for i ← 0 to N_CROMOSOMAS − 1 do
22:            intermedio.cromosomas.push_back(torneo(poblacion))
23:        end for
24:        Random::shuffle(intermedio.cromosomas)
25:        for i ← 0 to 2 · num_cruces_esperanza_mat − 2 by 2 do
26:            desc_Arit ← Cruce_Arit(intermedio.cromosomas[i], intermedio.
27:            cromosomas[N_CROMOSOMAS − i − 1], entrenamiento)
28:            intermedio.cromosomas[i] ← desc_Arit[0]
29:            intermedio.cromosomas[i + 1] ← desc_Arit[1]
30:            evaluaciones += 2
31:        end for
32:        for i ← 0 to num_mutaciones_esp_mat − 1 do
33:            index_cromosoma_mutar ← Random::get(distrib_cromos_uniforme)
34:            index_gen_mutar ← Random::get(distrib_indices_uniforme)
35:            intermedio.cromosomas[index_cromosoma_mutar].mutacion(index_gen_mutar,
36:            entrenamiento)
37:            evaluaciones += 1
38:        end for
39:        ReemplazoGeneracional(poblacion, intermedio)
40:        num_generaciones += 1
41:        if num_generaciones mód N_GENERACIONES == 0 then
42:            posic_alterar ← devolverSubconjuntoCromosomas(poblacion, PROB_
43:            MEMETICOS_BL · N_CROMOSOMAS, true)
44:            for i ← 0 to posic_alterar.size() − 1 do
45:                evaluaciones += busquedaLocalPractica2(entrenamiento, poblacion.
46:                cromosomas[posic_alterar[i]])
47:            end for
48:        end if
49:    end while
```

Procedimiento para el desarrollo de la práctica

En primer lugar, la implementación de la práctica se ha realizado en C++, con el apoyo de librerías como *vector*,*utility*,*cmath*,..., además de usar *random.hpp*, sin hacer uso de ningún framework.

Por otro lado, se ha hecho uso de las siguientes carpetas y archivos (más adelante se mencionan y explican los archivos desarrollados para implementar la práctica):

- *BIN*: Se almacena el ejecutable y la carpet *BIN/DATA*, en la que se almacenan los archivos con los datos usados en la práctica.
- *FUENTES*: Todo el código desarrollado se almacena en esta carpeta:
  - *FUENTES/INCLUDE*: Se almacenan los archivos .h y .hpp con la declaración de las funciones y constantes a usar. Además, el archivo random.hpp también aparece aquí.
  - *FUENTES/SRC*: Aquí aparecen los archivos donde se implementan las funciones declaradas en los archivos .h y .hpp.
  - *FUENTES/CMakeLists.txt*: Archivo con las órdenes correspondientes para la creación del archivo Makefile, con el que se genera el programa. Para reducir el tiempo de ejecución, se usa el parámetro -O3 para optimizar el código compilado.

Además, los archivos realizados por el alumno para la práctica son:

- *aux.h/aux.cpp*: En estos archivos se declaran e implementan funciones para el preprocesamiento de datos (lectura y normalización), el cálculo de distancias, la declaración de constantes que se consideran generales, la declaración de las estructuras de datos y algunas funciones de depuración.
- *practica1.hpp/practica1.cpp*: Declaración e implementación de los algoritmos de la práctica. Aquí está la implementación del clasificador 1-NN, los algoritmos de búsqueda lineal y Greedy RELIEF y la función para mostrar los resultados.
- *practica2.hpp/practica2.cpp*: Declaración e implementación de los algoritmos de la práctica. Está la implementación de los algoritmos geneéticos generacionales, estaccionarios y meméticos, además de funciones auxiliares y una búsqueda local revisada para los algoritmos meméticos.
- *main.cpp*: Establece la semilla para trabajar con números aleatorios y se llama a la función para las soluciones indicando el algoritmo a ejecutar.

Los pasos a seguir para la creación y ejecución del programa de la **práctica 1** son los siguientes:

- **Paso 1**: Abrir la terminal, o ir, a la carpeta *software/FUENTES*,
- **Paso 2**: Ejecutar el comando çmake .".
- **Paso 3**: Ejecutar el comando "make". Esto creará el ejecutable en la carpeta *software/BIN*.
- **Paso 4**: Ir a la carpeta *software/BIN*.
- **Paso 5**: Ejecutar el comando "./practica1 <semilla>".

Los pasos a seguir para la creación y ejecución del programa de la **práctica 2** son los siguientes:

- **Paso 1**: Abrir la terminal, o ir, a la carpeta *software/FUENTES*,
- **Paso 2**: Ejecutar el comando çmake .".
- **Paso 3**: Ejecutar el comando "make". Esto creará el ejecutable en la carpeta *software/BIN*.
- **Paso 4**: Ir a la carpeta *software/BIN*.
- **Paso 5**: Ejecutar el comando "./practica2 <semilla>".

Experimentación y análisis de resultados

Se han realizado las mediciones de los estadísticos sobre tres conjuntos de datos: *breast-cancer*,*ecoli* y *parkinsons*. El dataset *breast-cancer* son datos para identificar la gravedad del cáncer de mama a partir de ciertas características, con 569 muestras, 31 características (incluida su clasificación) y 2 clases. El dataset *ecoli* contiene datos para identificar la posición de proteinas, con 366 muestras, 8 características (clase incluida) y clasificación en 8 clases. Por último, *parkinsons* almacena datos que identifican la presencia de la enfermedad de Parkinson según medidas obtenidas por la voz de pacientes, con 195 muestras, 23 características (clase incluida) y 2 posibles clases; este dataset está desbalanceado, con más enfermos que sanos.

5.1. Exposición de resultados

5.1.1. Práctica 1

Los resultados obtenidos se consiguen mediante el uso de la semilla 60. No se usan otras debido a la gran cantidad de muestras que hay en los conjuntos de datos, aunque para algoritmos probabilísticos como búsqueda local es recomendable la comparación de resultados entre varias semillas. Aparecen, según la técnica 5-fold cross validation, los resultados de los estadísticos para cada ejecución del algoritmo, mostrándose la tasa de clasificación para la partición usada para test, la tasa de reducción de la solución, el valor de fitness y el tiempo de ejecución en segundos (s).

Tabla 5,1: Resultados obtenidos por el algoritmo 1NN en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	80.00	0.00	60.00	9×10-5	97.50	0.00	73.12	7×10-5	97.39	0.00	73.04	8.8×10-4
Partición 2	72.85	0.00	54.64	9×10-5	87.50	0.00	65.62	7×10-5	96.52	0.00	72.39	1.03×10-3
Partición 3	82.35	0.00	61.76	1×10-4	97.50	0.00	73.12	7×10-5	92.17	0.00	69.13	1.42×10-3
Partición 4	83.82	0.00	62.86	9×10-5	100.00	0.00	75.00	7×10-5	98.26	0.00	73.69	7.8×10-4
Partición 5	85.00	0.00	63.75	7×10-5	91.42	0.00	68.57	6×10-5	93.57	0.00	70.18	7.2×10-4
Media	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4

Tabla 5,2: Resultados obtenidos por el algoritmo BL en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	42.85	68.57	0.44	100.00	77.27	94.31	0.92	93.04	76.66	88.94	10.15
Partición 2	54.28	71.42	58.57	0.62	92.50	72.72	87.55	1.12	94.78	86.66	92.75	21.00
Partición 3	77.94	57.14	72.74	0.56	90.00	77.27	86.81	0.65	93.91	86.66	92.10	14.85
Partición 4	79.41	57.14	73.84	0.56	92.50	86.36	90.96	0.85	94.78	80.00	91.08	11.37
Partición 5	86.66	42.85	75.71	1.12	91.42	68.18	85.61	1.12	94.49	83.33	91.70	11.66
Media	75.08	54.28	69.88	0.66	93.28	76.36	89.05	0.93	94.20	82.66	91.31	13.80

Tabla 5,3: Resultados obtenidos por el algoritmo Greedy RELIEF en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	28.57	65.00	3.16×10-3	97.50	0.00	73.12	1.07×10-3	97.39	3.33	73.87	1.014×10-2
Partición 2	74.28	28.57	62.85	3.07×10-3	90.00	0.00	67.50	1.17×10-3	97.39	3.33	73.87	1.275×10-2
Partición 3	79.41	28.57	66.70	3.13×10-3	95.00	0.00	71.25	0.11× 10-3	89.56	13.33	70.50	9.93×10-3
Partición 4	80.88	28.57	67.80	3.3×10-3	100.00	0.00	75.00	1.06×10-3	97.39	0.00	73.04	8.78×10-3
Partición 5	86.66	28.57	72.14	3.59×10-3	91.42	0.00	68.57	1.12×10-3	94.49	0.00	70.87	9.21×10-3
Media	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016×10-2

Además, en la siguiente tabla se recogen los estadísticos medios en la ejecución de cada algoritmo:

Tabla 5,4: Resultados globales en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
1-NN	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4
RELIEF	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016x10-2
BL	75.08	54.28	69.88	0.66	93.28	76.36	89.05	0.93	94.20	82.66	91.31	13.80

5.1.2. Práctica 2

Los resultados obtenidos se han conseguido usando la semilla 42, y debido a la gran cantidad de muestras existentes, solo se realiza una ejecución de cada algoritmo para cada partición (aunque en los algoritmos probabilísticos es recomendable hacer varias ejecuciones). Según la técnica de 5-fold cross-validation,los resultados de los estadísticos para cada ejecución del algoritmo, mostrándose la tasa de clasificación para la partición usada para test, la tasa de reducción de la solución, el valor de fitness y el tiempo de ejecución en segundos (s).

Tabla 5,1: Resultados obtenidos por el algoritmo 1NN en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	80.00	0.00	60.00	9×10-5	97.50	0.00	73.12	7×10-5	97.39	0.00	73.04	8.8×10-4
Partición 2	72.85	0.00	54.64	9×10-5	87.50	0.00	65.62	7×10-5	96.52	0.00	72.39	1.03×10-3
Partición 3	82.35	0.00	61.76	1×10-4	97.50	0.00	73.12	7×10-5	92.17	0.00	69.13	1.42×10-3
Partición 4	83.82	0.00	62.86	9×10-5	100.00	0.00	75.00	7×10-5	98.26	0.00	73.69	7.8×10-4
Partición 5	85.00	0.00	63.75	7×10-5	91.42	0.00	68.57	6×10-5	93.57	0.00	70.18	7.2×10-4
Media	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4

Tabla 5,2: Resultados obtenidos por el algoritmo BL en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	75.71	42.86	67.50	0.59	95.00	72.73	89.43	1.42	97.39	83.33	93.88	16.19
Partición 2	68.57	57.14	65.71	0.79	92.50	77.27	88.69	1.09	98.26	90.00	96.20	12.85
Partición 3	73.53	42.86	65.86	0.68	95.00	68.18	88.30	1.63	93.04	83.33	90.62	12.17
Partición 4	67.65	71.43	68.59	0.49	90.00	77.27	86.82	1.16	90.43	86.67	89.49	21.23
Partición 5	71.67	71.43	71.61	0.47	91.43	77.27	87.89	1.09	94.50	90.00	93.37	19.32
Media	71.43	57.14	67.86	0.60	92.79	74.55	88.23	1.27	94.73	86.67	92.71	16.35

Tabla 5,3: Resultados obtenidos por el algoritmo Greedy RELIEF en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	28.57	65.00	3.16×10-3	97.50	0.00	73.12	1.07×10-3	97.39	3.33	73.87	1.014×10-2
Partición 2	74.28	28.57	62.85	3.07×10-3	90.00	0.00	67.50	1.17×10-3	97.39	3.33	73.87	1.275×10-2
Partición 3	79.41	28.57	66.70	3.13×10-3	95.00	0.00	71.25	0.11× 10-3	89.56	13.33	70.50	9.93×10-3
Partición 4	80.88	28.57	67.80	3.3×10-3	100.00	0.00	75.00	1.06×10-3	97.39	0.00	73.04	8.78×10-3
Partición 5	86.66	28.57	72.14	3.59×10-3	91.42	0.00	68.57	1.12×10-3	94.49	0.00	70.87	9.21×10-3
Media	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016×10-2

Tabla 5,4: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	78.57	57.14	73.21	36.73	100.00	72.73	93.18	16.58	94.78	90.00	93.59	182.29
Partición 2	67.14	57.14	64.64	35.93	95.00	81.82	91.70	16.72	94.78	86.67	92.75	190.02
Partición 3	77.94	57.14	72.74	36.29	95.00	77.27	90.57	16.83	88.70	86.67	88.19	190.38
Partición 4	80.88	57.14	74.95	36.04	92.50	81.82	89.83	16.85	88.70	93.33	89.86	160.32
Partición 5	85.00	57.14	78.04	38.53	88.57	86.36	88.02	18.24	93.58	86.67	91.85	162.01
Media	77.91	57.14	72.72	36.70	94.21	80.00	90.66	17.04	92.11	88.67	91.25	177.00

Tabla 5,5: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	75.71	57.14	71.07	35.53	100.00	77.27	94.32	17.16	95.65	86.67	93.41	158.63
Partición 2	67.14	57.14	64.64	35.76	95.00	77.27	90.57	16.88	94.78	83.33	91.92	160.12
Partición 3	77.94	57.14	72.74	36.49	90.00	72.73	85.68	17.11	90.43	90.00	90.33	158.76
Partición 4	79.41	57.14	73.84	36.40	92.50	81.82	89.83	16.62	93.04	90.00	92.28	160.57
Partición 5	85.00	57.14	78.04	37.95	91.43	81.82	89.03	18.45	93.58	83.33	91.02	163.11
Media	77.04	57.14	72.07	36.42	93.79	78.18	89.88	17.25	93.50	86.67	91.79	160.24

Tabla 5,6: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	72.86	57.14	68.93	35.84	87.50	81.82	86.08	16.94	93.91	86.67	92.10	153.79
Partición 2	54.29	71.43	58.57	35.55	82.50	72.73	80.06	17.55	96.52	86.67	94.06	160.39
Partición 3	77.94	57.14	72.74	36.73	92.50	77.27	88.69	17.63	94.78	90.00	93.59	158.28
Partición 4	72.06	71.43	71.90	36.10	100.00	77.27	94.32	17.44	90.43	90.00	90.33	155.38
Partición 5	85.00	57.14	78.04	38.45	88.57	86.36	88.02	18.17	97.25	83.33	93.77	164.16
Media	72.43	62.86	70.04	36.53	90.21	79.09	87.43	17.55	94.58	87.33	92.77	158.40

Tabla 5,7: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	74.29	57.14	70.00	36.46	95.00	72.73	89.43	17.08	96.52	90.00	94.89	161.31
Partición 2	75.71	42.86	67.50	36.81	90.00	77.27	86.82	17.33	94.78	86.67	92.75	158.92
Partición 3	77.94	57.14	72.74	37.45	97.50	77.27	92.44	17.15	86.96	86.67	86.88	162.78
Partición 4	79.41	57.14	73.84	37.12	92.50	81.82	89.83	16.98	87.83	93.33	89.20	161.53
Partición 5	85.00	57.14	78.04	38.71	94.29	77.27	90.03	18.18	97.25	90.00	95.44	167.45
Media	78.47	54.29	72.42	37.31	93.86	77.27	89.71	17.34	92.67	89.33	91.83	162.40

Tabla 5,8: Resultados obtenidos por el algoritmo AM-All en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	57.14	72.14	36.45	97.50	81.82	93.58	26.58	96.52	90.00	94.89	332.09
Partición 2	71.43	57.14	67.86	36.83	92.50	81.82	89.83	26.28	98.26	90.00	96.20	333.34
Partición 3	77.94	57.14	72.74	37.18	92.50	77.27	88.69	26.80	89.57	86.67	88.84	340.00
Partición 4	80.88	57.14	74.95	37.15	92.50	81.82	89.83	26.65	92.17	90.00	91.63	337.92
Partición 5	85.00	57.14	78.04	39.06	88.57	77.27	85.75	27.95	96.33	90.00	94.75	341.01
Media	78.48	57.14	73.14	37.33	92.71	80.00	89.54	26.85	94.57	89.33	93.26	336.87

Tabla 5,9: Resultados obtenidos por el algoritmo AM-Rand en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	77.14	57.14	72.14	40.79	90.00	77.27	86.82	21.19	96.52	90.00	94.89	162.83
Partición 2	74.29	57.14	70.00	42.25	95.00	81.82	91.70	19.46	95.65	90.00	94.24	182.00
Partición 3	77.94	57.14	72.74	41.63	95.00	81.82	91.70	23.11	94.78	90.00	93.59	172.35
Partición 4	80.88	57.14	74.95	41.89	92.50	81.82	89.83	19.40	92.17	90.00	91.63	154.10
Partición 5	85.00	57.14	78.04	43.69	97.14	81.82	93.31	22.58	90.83	90.00	90.62	199.03
Media	79.05	57.14	73.57	42.05	93.93	80.91	90.67	21.15	93.99	90.00	92.99	174.06

Tabla 5,10: Resultados obtenidos por el algoritmo AM-Best en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
Partición 1	78.57	57.14	73.21	40.59	100.00	77.27	94.32	21.96	94.78	90.00	93.59	216.79
Partición 2	68.57	57.14	65.71	41.00	85.00	72.73	81.93	22.45	93.04	90.00	92.28	213.45
Partición 3	77.94	57.14	72.74	41.70	97.50	77.27	92.44	22.42	87.83	90.00	88.37	215.29
Partición 4	72.06	71.43	71.90	41.16	92.50	81.82	89.83	21.47	91.30	90.00	90.98	213.52
Partición 5	85.00	57.14	78.04	43.80	85.71	81.82	84.74	23.23	94.50	83.33	91.70	221.63
Media	76.43	60.00	72.32	41.65	92.14	78.18	88.65	22.31	92.29	88.67	91.38	216.14

Además, en la siguiente tabla se recogen los estadísticos medios en la ejecución de cada algoritmo.

Tabla 5,4: Resultados globales en el problema del APC

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T	*%_clas*	*%red*	*Fit,*	T
1-NN	80.80	0.00	60.60	9×10-5	94.78	0.00	71.08	7×10-5	95.58	0.00	71.68	9.6×10-4
RELIEF	79.67	28.57	66.90	3.25×10-3	94.78	0.00	71.08	1.1×10-3	95.24	4.00	72.43	1.016x10-2
BL	71.43	57.14	67.86	0.60	92.79	74.55	88.23	1.27	94.73	86.67	92.71	16.35
AGG-BLX	77.91	57.14	72.72	36.70	94.21	80.00	90.66	17.04	92.11	88.67	91.25	177.00
AGG-CA	77.04	57.14	72.07	36.42	93.79	78.18	89.88	17.25	93.50	86.67	91.79	160.24
AGE-BLX	72.43	62.86	70.04	36.53	90.21	79.09	87.43	17.55	94.58	87.33	92.77	158.40
AGE-CA	78.47	54.29	72.42	37.31	93.86	77.27	89.71	17.34	92.67	89.33	91.83	162.40
AM-All	78.48	57.14	73.14	37.33	92.71	80.00	89.54	26.85	94.57	89.33	93.26	336.87
AM-Rand	79.05	57.14	73.57	42.05	93.93	80.91	90.67	21.15	93.99	90.00	92.99	174.06
AM-Best	76.43	60.00	72.32	41.65	92.14	78.18	88.65	22.31	92.29	88.67	91.38	216.14

5.2. Estudio de convergencia

5.2.1. Práctica 2

Se hará un estudio de convergencia solo para dos conjuntos de datos, y únicamente para la partición uno, intuyendo que el resto se comportarán de manera similar. Las gráficas han sido generadas en *R*, habiendo escrito la evolución de fitness en un archivo .csv. El pseudocódigo usado en *R* para una partición de los algoritmos genéticos generacionales es el siguiente (para el resto solo cambian los nombres):

Algorithm 36 Análisis y Visualización de Fitness de Algoritmos Genéticos

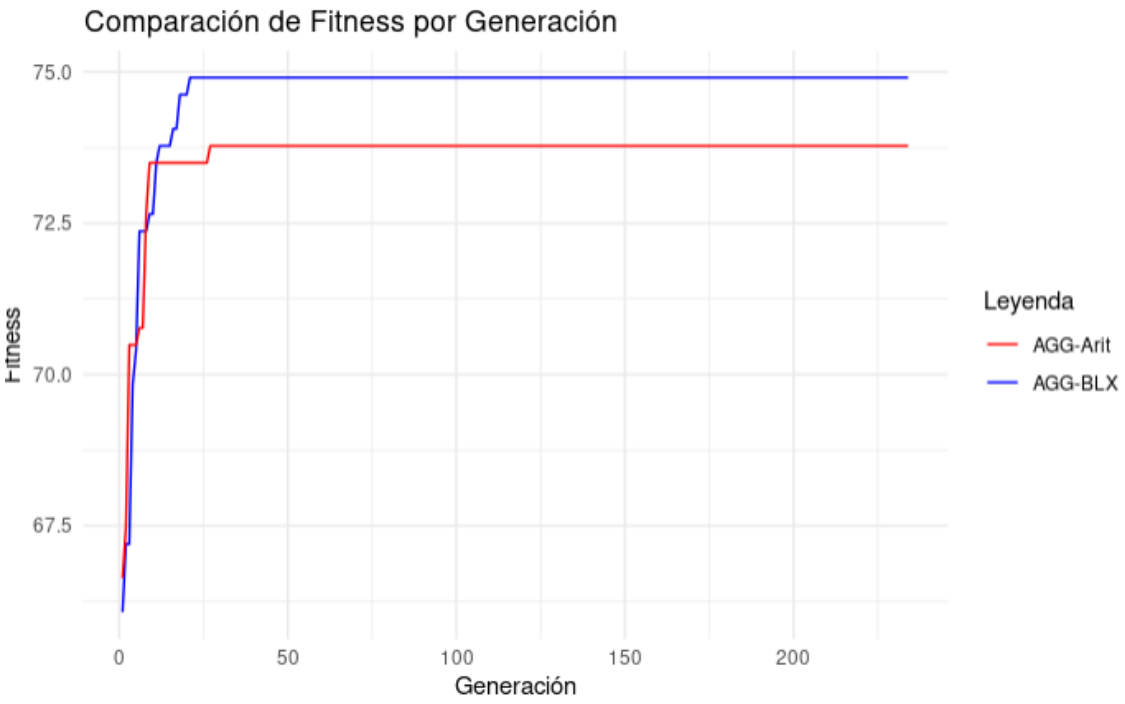
```
agg_blx ← leerArchivoCSV("fitness_Partic_1_algoritmo_AGG_BLX_dataset_ecoli_.csv")
ImprimirEstructura(agg_blx)
agg_arit ← leerArchivoCSV("fitness_Partic_1_algoritmo_AGG_Arit_dataset_ecoli_.csv")
ImprimirEstructura(agg_arit)
                                  ▷ Renombrar la columna 'Fitness' en cada dataframe
agg_blx ← transformar(agg_blx, fitness1 = agg_blx.Fitness)
agg_arit ← transformar(agg_arit, fitness2 = agg_arit.Fitness)
                                  ▷ Eliminar la columna 'Fitness' original si aún existe
agg_blx.Fitness ← NULL
agg_arit.Fitness ← NULL
df_unido ← fusionar(agg_blx, agg_arit, por = "Generacin", todo = TRUE)
CargarLibrería(ggplot2)
                                                              ▷ Crear el gráfico
CrearGráfico(df_unido)

procedure CrearGráfico(datos)
    ggplot(datos, aes(x = Generacin))+
      geom_line(aes(y = fitness1, colour = "AGG − BLX"))+
      geom_line(aes(y = fitness2, colour = "AGG − Arit"))+
      scale_colour_manual(values = {"AGG − BLX" = "blue","AGG − Arit" = "red"})+
      labs(title = "ComparacindeFitnessporGeneracin", x = "Generacin", y = "Fitness", colour = "Leyenda")+
      theme_minimal()
end procedure
```

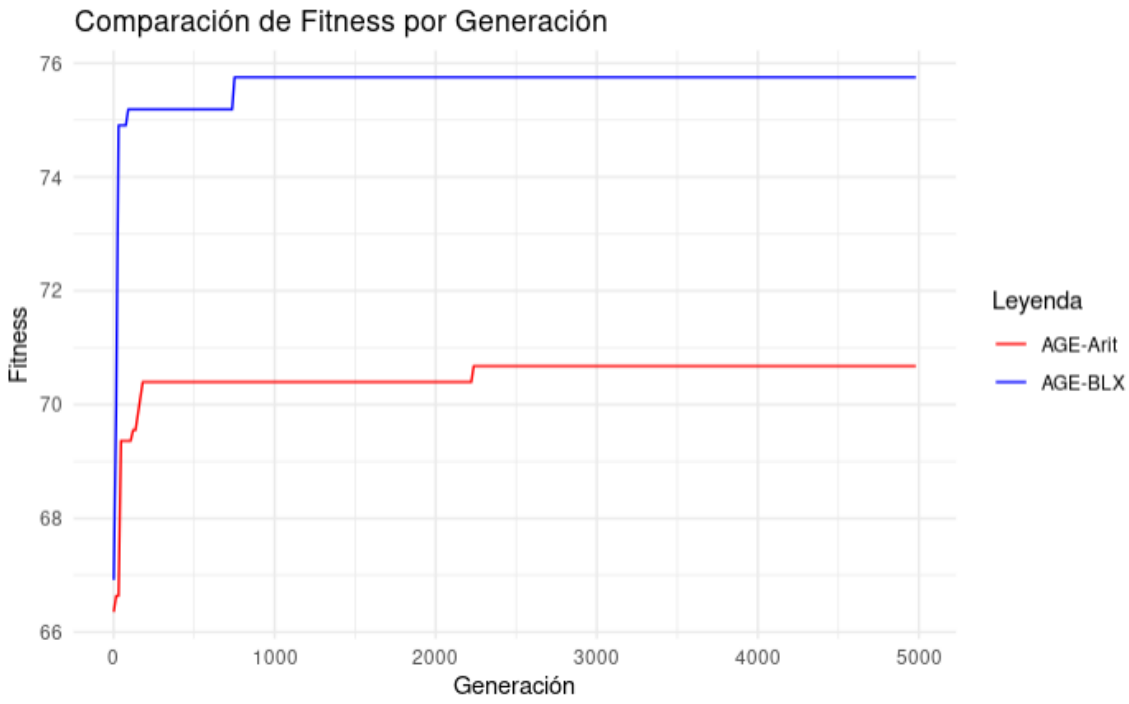
Para los algoritmos meméticos se aplica la técnica de *oversampling* para que se pueda realizar la unión de dataframes y se puede hacer la gráfica, intentando no añadir más ruido del necesario.

Dataset *Ecoli*

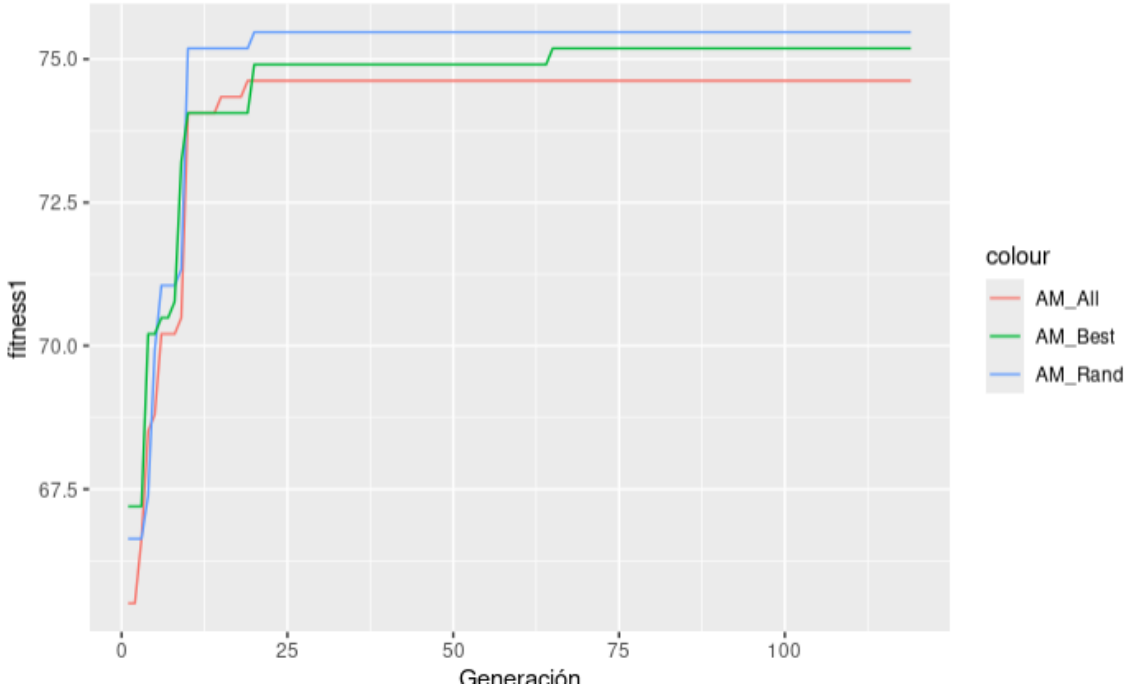
Convergencia de algoritmos genéticos generacionales:



Convergencia de algoritmos genéticos estacionarios:

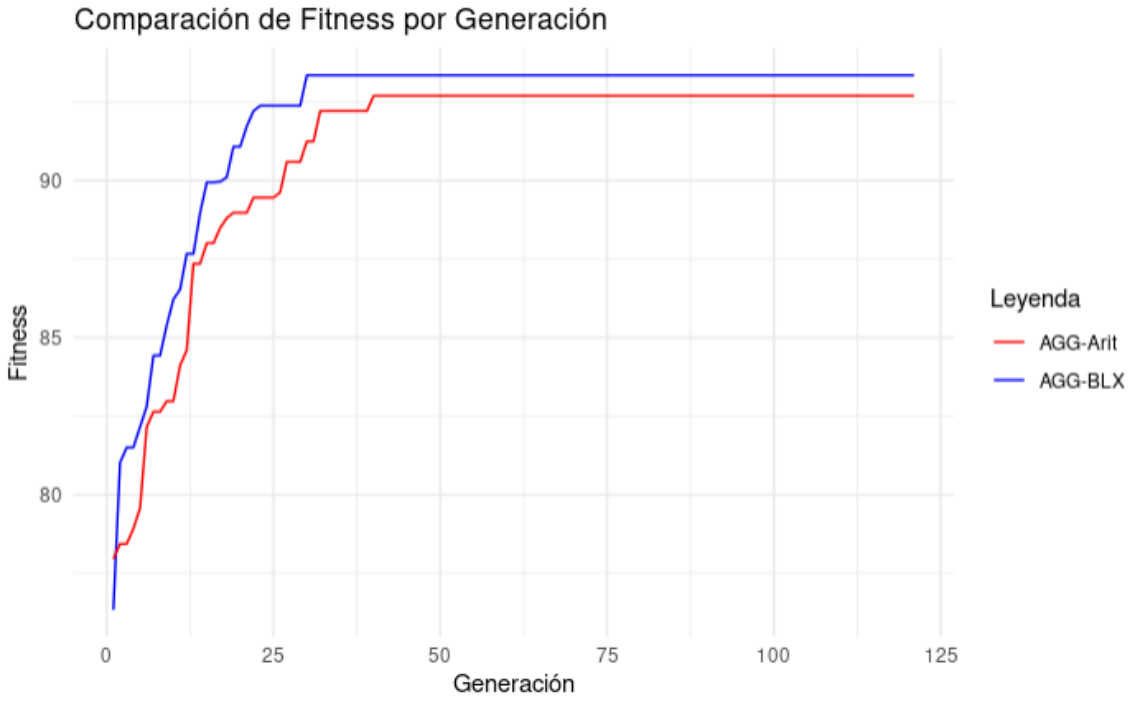


Convergencia de algoritmos meméticos:

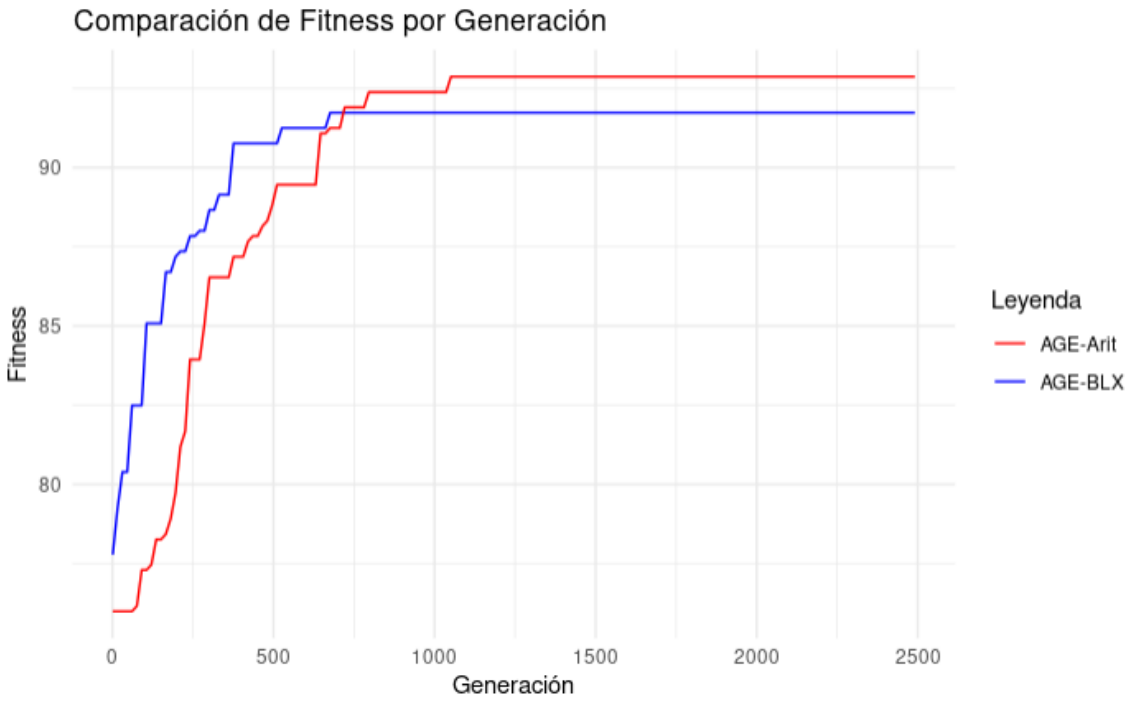


Dataset *Parkinsons*

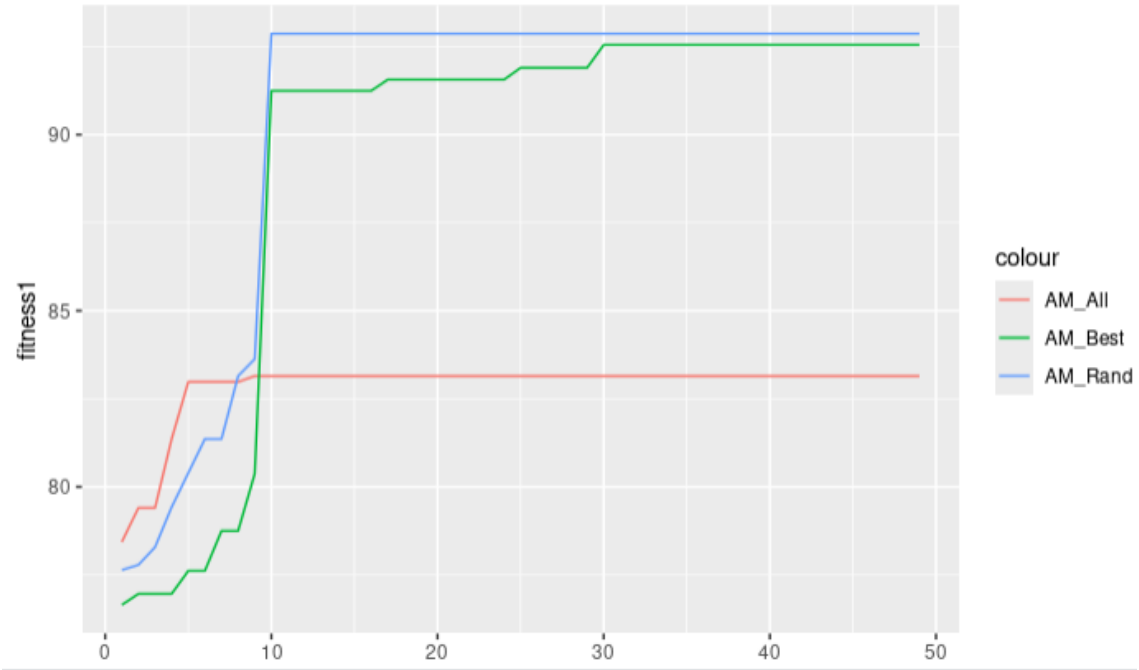
Convergencia de algoritmos genéticos generacionales:



Convergencia de algoritmos genéticos estacionarios:



Convergencia de algoritmos meméticos:



Comentarios sobre la convergencia

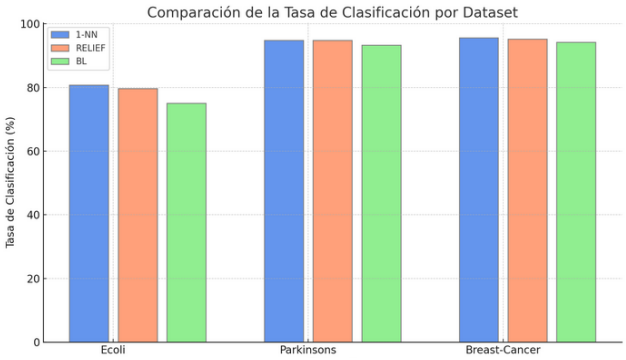
Se observa que los algoritmos que en general menos tardan en converger para la partición uno son los meméticos, seguidos de los genéticos generacionales y por último los genéticos estacionarios, que son los que en general más tardan en converger. Si se tuviese que elegir un algoritmo únicamente por su velocidad de convergencia, sería *AM_All*, aunque dado el tiempo de ejecución medio asociado, este algoritmo parece converger pero en realidad se queda oscilando. Sin embargo, el algoritmo *AM_Rand* converge casi en el mismo momento que *AM_Best*, dando ligeramente mejores resultados en fitness, por lo que se escogería debido a los resultados dados en la partición uno y su rápida convergencia.

5.3. Análisis de resultados

5.3.1. Práctica 1

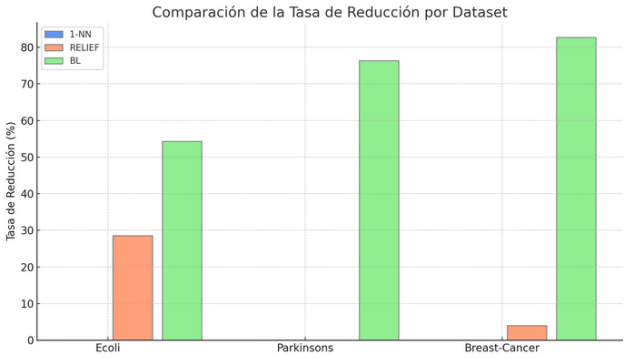
Se ha considerado el uso de gráficos de barras, ya que se estima recogen y muestran mejor los estadísticos medios para comparar entre algoritmos. Se hace la restricción a la media de cada estadístico.

- Tasa de clasificación:



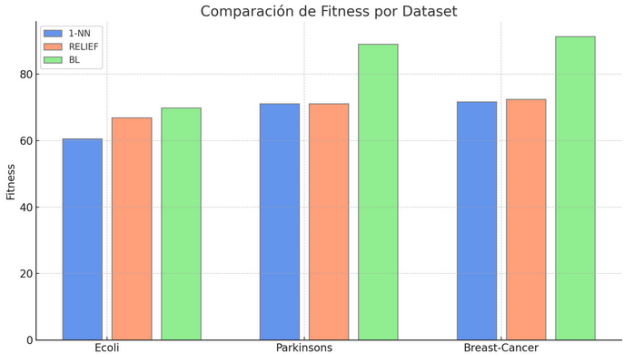
Los resultados de la tasa de clasificación en test media arroja resultados similares entre los tres algoritmos, siendo la diferencia más notable que el dataset *ecoli* da una tasa de clasificación menor al resto de datasets. Mirando cada algoritmo, 1-NN siempre es algo mejor, y esto se debe a que están todos los pesos a 1.0, siendo el resto valores entre 0.0 y 1.0.

- Tasa de reducción:



Tal y como se define la tasa de reducción, es normal que para 1-NN no haya, pues todos los pesos están con valor 1.0. Respecto al resto de algoritmos, se desprenden resultados esperables, teniendo en todos los conjuntos de datos menor tasa de reducción en el algoritmo greedy RELIEF que en la búsqueda local, puesto que 1-NN no descarta características y greedy RELIEF es menos agresivo. En general, búsqueda local elimina un mayor número de características irrelevantes en los conjuntos de datos usados, lo que lleva a pensar que muchas características tomadas en cuenta en las mediciones de las muestras no son de mucha importancia para la clasificación en el modelo, sobre todo en el dataset *breast-cancer* (aunque *parkinsons* no da una tasa de reducción mucho menor).

- Fitness:

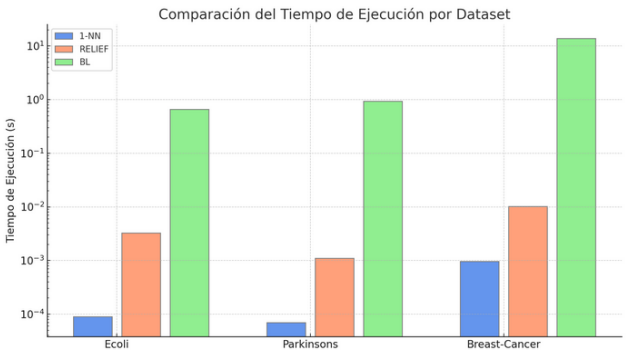


Usando $\alpha = 0{,}75$, dando mayor peso a las muestras bien clasificadas que a la reducción de características, como la expresión es $\alpha \cdot \text{Tasa\_clas}(W) + (1 - \alpha) \cdot \text{Tasa\_red}(W)$, los resultados mostrados son normales. Como α no es muy pequeño, se parte de que en general la tasa de clasificación es parecida, lo que hace que $\alpha \cdot \text{Tasa\_clas}(W)$ no varíe mucho entre algoritmos, teniendo búsqueda local algo menos. Por otro lado, si se suma $(1 - \alpha) \cdot \text{Tasa\_red}(W)$, dado que $\text{Tasa\_red}(W)$ es mucho más alto en búsqueda local, resulta en un fitness más alto que el resto de algoritmos. Además, en 1-NN la tasa de reducción es nula, y en consecuencia el valor de fitness coincide con el de la tasa de clasificación.

Se ha remarcado que si α no fuese muy pequeño porque en tal caso, como $\lim_{\alpha \to 0} \text{fitness}(W) = \text{Tasa\_red}(W)$, entonces fitness para 1-NN es nulo, y el de búsqueda local es mucho mayor al de greedy RELIEF (notándose mucho más en el dataset *parkinsons*, ya que volviendo a la gráfica de la tasa de reducción media, dicho dataset tiene tasa de reducción nula). Si $\alpha \to 1$, se tendría que 1-NN tiene más fitness (pasando al límite), aunque no con una diferencia tan notable como cuando $\alpha \to 0$. Esto da a entender que la elección de α no es trivial, y debe tomarse teniendo en cuenta cuánta importancia se le quiera dar a la cantidad de muestras bien clasificadas en test o a la cantidad de características que el algoritmo considera no relevantes.

A grandes rasgos, el algoritmo de búsqueda local obtiene mejor fitness medio que el resto de algoritmos por lo expuesto anteriormente, aunque en *ecoli* no es mucho mejor al algoritmo greedy RELIEF.

- Tiempo de ejecución (segundos):



Para una correcta comparación se ha pasado a una escala logarítmica, manteniendo las proporciones, ya que el tiempo de ejecución medio de búsqueda local en *breast-cancer* es mucho mayor al del resto de ejecuciones.

Se recuerda que *ecoli* tiene 366 muestas con 7 características (excluyendo la clase), *parkinsons* tiene 195 muestras con 22 características (excluyendo la

clase) y *breast-cancer* tiene 569 muestras con 30 características (excluyendo la clase). A primera vista, se observa que el dataset *breast-cancer* es mucho mayor que los otros dos, y entre *ecoli* y *parkinsons*, el primero tiene más muestras, pero el segundo más características. Esto puede traducirse en un objeto *Dataset* de más filas (más muestras) en *ecoli* y un objeto *Dataset* de más columnas (más características) en *parkinsons*. Por lo tanto, pensando en el objeto como una matriz (exluyendo la clase y el nombre y tipo de atributos), se tiene que *ecoli* tiene $366 \times 7 = 2562$ elementos y *parkinsons* tiene $195 \times 22 = 4290$ elementos, lo que lleva a pensar que *parkinsons* sea más grande que *ecoli*.

Como *breast-cancer* es el dataset más grande, se observa que es el que más tarda. Si se mira un dataset de los tres, se observa que el que más tarda es búsqueda local, y el que menos 1-NN. Esto es normal puesto que búsqueda local puede caer en óptimos locales, intentando mejorar hasta que se da la condición de parada en cuanto a iteraciones o vecinos generados sin mejora, que conlleva bastante más tiempo que simplemente observar si la clase de una muestra en test coincide con la clase de su vecino más cercano.

Por otra parte, el tiempo de greedy RELIEF se encuentra acotado entre los tiempos de 1-NN (inferiormente) y búsqueda local (superiormente). Tarda más que 1-NN ya que debe evaluar y seleccionar características, pero menos que búsqueda local ya que no tiene una condición de parada de tantas iteraciones y generaciones como búsqueda local, además de que búsqueda local es un algoritmo probabilístico, encontrando una solución mejor o no encontrándola en función de la mutación, mientras que greedy RELIEF no lo es.

5.3.2. Conclusiones de la práctica 1

Tras realizar el análisis de cada estadístico medio, se llega a la conclusión de que el algoritmo de búsqueda local ofrece mejores resultados en cuanto a que obtiene mayor valor en fitness. Sin embargo, cabe mencionar que tarda bastante más que el resto de algoritmos. Era de esperar también que 1-NN fuese el que menos tardase.

Si se tuviese que escoger un algoritmo con el fin de resolver un problema, la elección del mismo dependerá de los requisitos del mismo. Si el tiempo no es un inconveniente, o el dataset no es muy grande, se escogería el algoritmo de búsqueda local, vistos los resultados medios dados en fitness, además de la capacidad de reducción de características que no considera relevantes. Podría decirse que consigue obviar características no relevantes para una correcta clasificación (aprende características más robustas). Sin embargo, si el tiempo fuese crucial en el problema a resolver, habría que discutirse si escoger 1-NN o greedy RELIEF. Este último da unos resultados similares a 1-NN, aunque mejora un poco en el dataset *ecoli*. Si se observan los valores de las tablas (los de las gráficas han sido escalados logarítmicamente), se haría la elección del algoritmo greedy RELIEF, ya que aunque no mejora mucho el valor de fitness, y pese a tardar ligeramente más, en general reduce algunas características (aunque no ocurre en el dataset *parkinsons*.

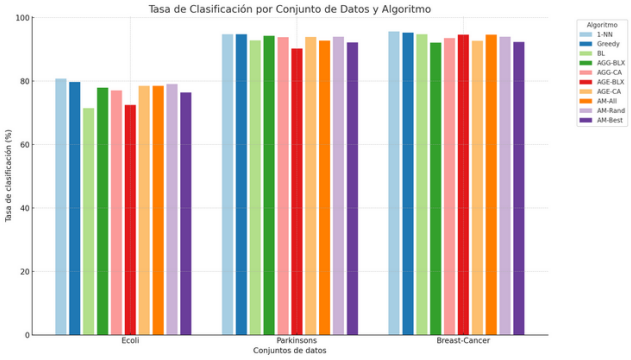
Una consideración crucial en nuestro análisis es el desequilibrio de clases presente en el dataset *parkinsons*. Este desequilibrio puede influir significativamente en la fiabilidad y el rendimiento de los modelos evaluados. En particular, puede llevar a una sobreestimación de la importancia de ciertas características, dado que las métricas de evaluación podrían estar sesgadas hacia la clase predominante. Este fenómeno se observa en la tasa de reducción nula reportada para el algoritmo Greedy Relief en este dataset, lo que sugiere que el algoritmo podría estar priorizando incorrectamente las características en función de su prevalencia en la clase mayoritaria en lugar de su verdadera relevancia para la clasificación. Para solucionar esto se podrían usar otras

métricas (más potentes con respecto el desbalanceo de clases) u obtener muestras tales que aumente la calidad de los datos, entre otros. Esto demuestra que la calidad y fiabilidad de un modelo dependen en gran medida de la calidad de los datos con los que se entrene.

5.3.3. Práctica 2

Al igual que en la práctica anterior, se considera el uso de gráficos de barras sobre los estadísticos medios por ser una buena representación de los datos obtenidos.

- Tasa de clasificación:



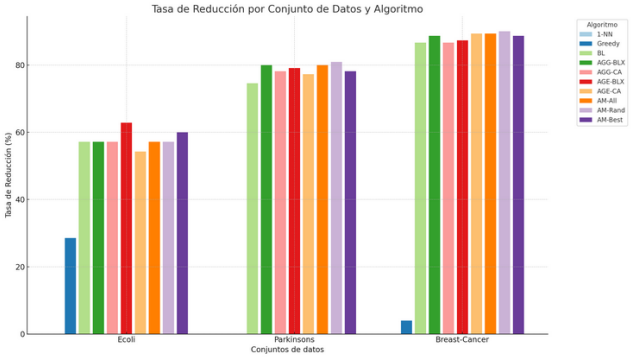
Realizando una comparación entre los algoritmos, en cuanto a la consistencia de algoritmos, *1-NN* y *Greedy* muestran consistencia entre los tres conjuntos de datos, con diversos niveles de efectividad. Con respecto a la variablidad, *AGG-BLX* y *AGE-CA* parecen tener una variabilidad más notable en su rendimiento en función del dataset, dando a entender que tienen un nivel alto de sensibilidad a las características particulares de cada conjunto.

Se observa que los resultados son menores en *Ecoli*, lo cual es entendible debido a las pocas características que presenta el conjunto, o a la naturaleza de las mismas.

Los algoritmos evolutivos y meméticos muestran resultados variados, por lo que un algoritmo más complejo no es necesariamente equivalente a resolver de mejor manera el problema (esto se discutirá más adelante, en el apartado relacionado a fitness).

En general, si se tuviese que elegir un algoritmo en función a este estadístico **únicamente**, se escogería **1-NN**.

- Tasa de reducción:

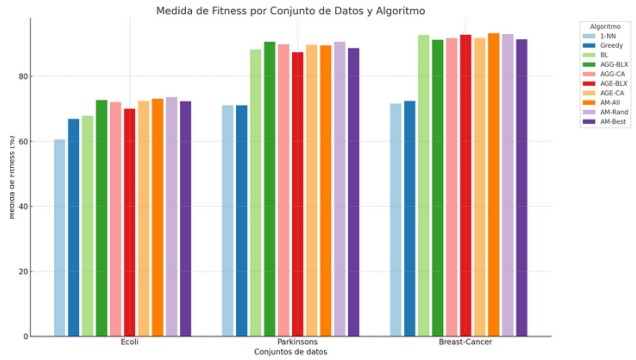


En el caso de la tasa de reducción, los resultados de algoritmos como *1-NN* o *Greedy* son bastante peores, dando a entender que no reducen muchas características en función del conjunto de datos, y el modelo obtenido no es simple. Comparando los resultados en función del conjunto de datos, *Breast-Cancer* parece inducir a mayor variabilidad en lo que respecta a reducción, lo cual era deseable por la gran cantidad de características que presenta. Sin embargo, el dataset *Ecoli* reduce bastante menos, por lo que se pdoría decir que es más consistente que el resto de conjuntos. Se debe comentar que mayor reducción no implica mejor modelo, sino simplicidad del mismo, por lo que no es un estadístico sobre el que se deba elegir modelo.

Los algoritmos evolutivos y meméticos consiguen reducir bastante características, en comparación a la reducción en algoritmos de la práctica anterior. Sumado a lo discutido en la tasa de clasificación, da a entender que pueden ofrecer mejores de fitness.

Si se tuviese que elegir un modelo **exclusivamente** por el valor de la tasa de reducción buscando un modelo **simple**, se escogería *AGG-BLX* o *AGE-BLX*, intuyendo que el cruce *BLX* implica reducir más.

- Fitness:



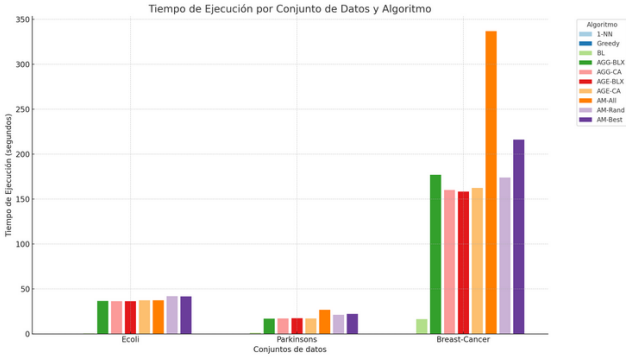
La discusión de la situación en función del valor $\alpha \in [0, 1]$ se realizó en la discusión de la anterior práctica.

A grandes rasgos, se observa que en el conjunto de datos *Breast-Cancer* se obtiene mejor fitness, indicando que los algoritmos manejan en general bien las características y sus relaciones del mismo. Sin embargo, en el resto de datasets, sobre todo en el caso de *Ecoli*, al haber un menor valor de fitness, parece que los algoritmos presentan consistencia en los mismos, siendo útiles cuando las catacterísticas del problema no cambian drásticamente.

Con respecto a los algoritmos de la práctica 2, los meméticos arrojan resultados levemente mejores que los genéticos y la búsqueda local. Sin embargo, se afirma que los algoritmos geneéticos en general también dan buenos resultados. Además, los algoritmos meméticos ajustan dinámicamente su comportamiento en respuesta a cambios, lo que puede explicar los resultados con respecto a los algoritmos genéticos.

Si se tuviese que elegir un algoritmo **únicamente** en función de fitness, se escogería *AM-Rand*, que corresponde con el denominado como **AM-(10,0.1)**.

- Tiempo (segundos):



Se observa que el dataset *Breast-Cancer* presenta un tiempo de ejecución bastante mayor que el resto de conjuntos. Esto era esperable por tener muchas más muestras que el resto y presentar mayor número de características. Obviando los algoritmos *1-NN* y *Greedy*, que presentan menor tiempo de ejecución, los algoritmos genéticos parecen mantener un tiempo de ejecución similar, incluso con algunos algoritmos meméticos. Mención especial merece el algoritmo *AM-All*, que corresponde a *AM-(10,1)*, que tarda bastante más en comparación al resto, lo que podría derivarse de tener que aplicar, cada 10 generaciones, búsqueda local a todos los cromosomas de la población (mientras que en el resto se aplica bajo cierta probabilidad pequeña o un subconjunto de mejores soluciones).

Si se tuviese que escoger un algoritmo **exclusivamente** por este estadístico medio, de los algoritmo probabilísticos se escogería **búsqueda local**, aunque para la elección definitiva de algoritmo dependerá de otros factores como fitness.

5.3.4. Conclusiones de la práctica 2

Brevemente, si el tiempo fuese crucial, de todos los algoritmos se escogería el algoritmo *Greedy Relief*, por presentar buen balance entre fitness y tiempo de ejecución. Teniendo en cuenta que solo se pueden escoger los algoritmos de la práctica 2, se escogería el algoritmo *AGE-BLX*, pues entre todos los algoritmos, aunque presenta en general el mismo valor fitness que el resto, tarda algo menos que el resto de algoritmos. Por otro lado, si el tiempo no importase, buscando maximizar la función objetivo se escogería el algoritmo *AM-All*, también llamado *AM-(10,1)*, pues arroja valores ligeramente mayores que el resto (aunque en cuanto al tiempo, tarda bastante más).

Frente a un problema real, donde se busca balancear resultados con tiempo de ejecución, se haría la elección del algoritmo **AM-Rand**, también llamado **AM-(10,0.1)**, ya que a grandes rasgos arroja los mismos resultados que el resto de algoritmos y tarda en media menos que el resto (aunque más que algunos genéticos, el balance entre tiempo de ejecución y resultados es mejor). Lo anterior si se tuviese que elegir un algoritmo de la práctica 2. Con una visión global de todos los algoritmos, aunque el fitness sea algo menor en *búsqueda local*, tarda bastante menos que los algoritmos genéticos y meméticos, por lo que también podría considerarse para la resolución de un problema.

También es importante hacer un análisis de los resultados en función del desequilibrio presente en datasets como *parkinsons*, que influye significativamente en la fiabilidad y rendimiento de los modelos entrenados, llegando a dar un posible sobreajuste y dar un modelo sesgado en función de la clase mayoritaria. Por ello, habría que cuestionar si las caracteríticas eliminadas en el dataset *parkinsons* en función del algoritmo no ha hecho que el modelo se ajuste a las clases mayoritarias.