Asignatura (2018-2019)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

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

Simón López Vico, 26504148Y
simondelosbros@correo.ugr.es
Grupo de prácticas: Miércoles

6 de mayo de 2019

Índice

1. Breve descripción del problema.

En esta práctica nos encargaremos de resolver el problema del Aprendizaje de Pesos en Características, que consistirá en ajustar un vector de pesos $\{\omega_1, \ldots, \omega_n\}$ donde cada una de sus componentes determinará el valor de la característica i-ésima del conjunto de datos. Si $\omega_i = 1$ significará que el atributo i-ésimo es muy importante, mientras que si $\omega_i = 0$ la característica i-ésima no será relevante; concretamente, cuando $\omega_i < 0{,}2$ descartaremos dicho atributo.

Utilizaremos distintos algoritmos para ajustar nuestro vector de pesos. Una vez optimizado el vector, usaremos el clasificador K-NN (con K=1) para clasificar según el vecino más cercano usando una distancia ponderada según los pesos:

```
function Distancia(vector a, vector b, vector pesos)
    Si tamanio(a) != tamanio(b)
        Devuelve -1

    Para i=0 hasta tamanio(a)
        Si pesos[i]>=0.2
            Distancia += ( pesos[i]*(a[i]-b[i]) )^2

    Devuelve Distancia
```

Véase que no estamos calculando la raíz para computar la distancia euclídea; prescindimos de esto porque los resultados serán los mismos calculando y sin calcular la raíz, y además la operación `sqrt(x)` tiene un alto tiempo de computo.

El código del clasificador K-NN será el siguiente:

```
function one_NN(e_prima,pesos, indice) //indice leave-one-out
    c_min=train[0].etiqueta
    dist_min=Distancia(train[0], e_prima, pesos)

    Para i=1 hasta tamanio(train)
        Si i != indice
            d=Distancia(train[i], e_prima, pesos)
            Si d<d_min
                c_min=train[i].etiqueta
                d_min=d

    Devuelve c_min
```

La variable `indice` indicada en la cabecera de la función es utilizada para la técnica *leave-one-out*, la cual es necesaria usar porque si intentásemos clasificar un ejemplo del conjunto de entrenamiento directamente con el clasificador 1-NN, el ejemplo más cercano sería siempre él mismo, con lo que se obtendría un 100 % de acierto. Para solucionar esto, clasificaremos cada elemento del conjunto de entrenamiento buscando el ejemplo más cercano sin considerar a él mismo.

Para ajustar el vector de pesos W sobre el clasificador trataremos de maximizar la función $F(W) = \alpha * tasa_{clas}(W) + (1-\alpha) * tasa_{red}(W)$, donde $tasa_{clas}$ determinará el porcentaje de instancias bien clasificadas sobre el total de instancias, y $tasa_{red}$ el porcentaje de atributos con un peso asociado menor a 0.2 sobre el total de atributos.

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

2.1. Esquema de representación del problema y formateo de los ficheros de datos.

El esquema de representación asociado a nuestro problemas es el vector de pesos w, el cuál tomará valores en el intervalo $[0, 1]$ y tendrá una longitud de N elementos.

Por otra parte, se representará cada cromosoma con un `pair<vector<double>, double>`, donde el primer elemento será el vector de pesos y el segundo el valor de la función de evaluación sobre dicho vector, aplicando la técnica *leave-one-out*.

La resolución de este problema APC ha sido implementada en C++. Para el almacenamiento de los datos recogidos en los distintos ficheros `arff` se ha usado un vector llamado `data` en el que cada componente será un `pair <vector<double>, int >`; el primer elemento del `pair` recogerá los atributos de una instancia del conjunto total de datos, mientras que el segundo será un `int` con la etiqueta de la instancia. El vector `data` será separado en 5 particiones disjuntas más adelante para realizar el *5-fold validation*.

Por otra parte, los ficheros `arff` han sido ligeramente modificados para facilitar la lectura de éstos:

- **colposcopy.arff**: ninguna modificación.
- **ionosphere.arff**: cambio de las clases b y g por 0 y 1.
- **texture.arff**: inicialmente cada uno de los atributos de cada instancia se encuentra separado por una coma y un espacio; se modifica para solamente se separen mediante una coma.
  Clases iniciales: {2, 3, 4, 9, 10, 7, 6, 8, 12, 13, 14}.
  Clases modificadas: {0, 1, 2, 6, 7, 4, 3, 5, 8, 9, 10}.

2.2. Lectura y preprocesado.

Comenzaremos leyendo el fichero de datos pasado al programa con la función `read_data`, la cual leerá un archivo en formato `arff` y guardará todos sus valores en el vector `data`;

además, almacenará el total de atributos y de clases del dataset leído.

A continuación normalizaremos los datos para que se encuentren en el intervalo $[0, 1]$. Para ello usaremos la siguiente fórmula:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j},$$

donde Max_j y Min_j serán el máximo y el mínimo de cada uno de los atributos. Hay que tener en cuenta que si un atributo toma el mismo valor para todas las instancias, la diferencia entre el máximo y el mínimo será 0 por lo que no podremos realizar la división; en este caso, $x_j^N = 0$.

Una vez normalizados todos los datos, realizaremos las particiones necesarias para el *K-fold validation*, en nuestro caso K=5. Para ello usaremos el siguiente código:

```
function k folds()
    Mezclar(data)

    Para i=0 hasta n_clases
        Para j=0 hasta tamanio(data)
            Si data.clase == i
                particiones[n_introd%n_part].push_back(data[j]);
```

De esta manera, el número de clases estará balanceada respecto a cada partición, es decir, en cada partición habrá el mismo porcentaje (aproximadamente) de elementos con una clase determinada.

Notemos que la normalización se debe de aplicar siempre antes de realizar las particiones de elementos, pues si se hiciera al contrario cada partición estaría normalizada por un máximo y mínimo distinto.

Finalmente generaremos los conjuntos *train* y *test* con los que vamos a entrenar y validar nuestro clasificador.

3. Algoritmos genéticos.

Usaremos dos esquemas de evolución distintos, un esquema generacional con elitismo (AGG) y un esquema estacionario (AGE). Los dos se basarán en:

- Generar una población inicial de 30 cromosomas mediante una distribución uniforme $U(0, 1)$.
- Seleccionar los individuos de la población que van a cruzarse. Aquí reside la diferencia entre AGG y AGE.

- Cruzar estos individuos generando dos hijos por cada cruce.
- Mutar los nuevos cromosomas con una determinada probabilidad.
- Reemplazar la población con esta nueva generación. Este paso también cambia entre AGG y AGE.

3.1. Operadores de Selección, Cruce y Mutaciones.

Como operador de selección se usará el torneo binario, consistente en elegir aleatoriamente dos individuos de la población y seleccionar el mejor de ellos. Se han realizado dos funciones diferentes para cada esquema de evolución, aunque muy parecidas.

Para el algoritmo generacional AGG debemos seleccionar una población de padres del mismo tamaño que la población genética; para evitar que algún elemento de la población sea padre dos veces, se usa un vector que almacenará las posiciones de los cromosomas que todavía no han sido padres. El código es el siguiente:

```
function binary_tournament(&ind) //Usamos un vector de indices
                                 //para no repetir los padres
    competidor1=Randint(0, ind.size()-1)
    competidor2;

    Hacer
        competidor2=Randint(0, ind.size()-1)
    Mientras competidor1==competidor2 //Asi los dos competidores
                                      //no seran el mismo

    Si cromosomas[ind[competidor1]].valor >
       cromosomas[ind[competidor2]].valor
        ind.Borra(Posicion(competidor1))
        Devuelve competidor1
    Sino
        ind.Borra(Posicion(competidor2))
        Devuelve competidor2
```

Por otra parte, el algoritmo estacionario AGE solo utilizará dos padres de la población, por lo que no es necesario llevar un recuento de quién ha sido padre en cada generación y nos olvidamos del vector de índices para ahorrar tiempo en la ejecución de la función:

```
function binary_tournament()
    competidor1=Randint(0, cromosomas.size()-1)
    competidor2;

    Hacer
        competidor2=Randint(0, cromosomas.size()-1)
    Mientras competidor1==competidor2 //Asi los dos competidores
                                      //no seran el mismo
```

```
    Si cromosomas[competidor1].valor >
       cromosomas[competidor2].valor
        Devuelve competidor1
    Sino
        Devuelve competidor2
```

Hablemos ahora de los operadores de cruce. Se han utilizado dos operadores diferentes, BLX-α y *Arithmetic-Crossover* (AC). El primero consistirá en generar dos hijos donde cada componente de su vector de pesos será un valor aleatorio tomado en en el intervalo $[C_{min} - I * \alpha, C_{max} + I * \alpha]$ ($\alpha = 0{,}3$), donde $C_{max} = max\{Padre_{1i}, Padre_{2i}\}$, $C_{min} = min\{Padre_{1i}, Padre_{2i}\}$ e $I = C_{max} - C_{min}$. Si el valor generado es menor que 0 o mayor que 1, truncaremos a 0 o 1 respectivamente. El pseudocódigo será:

```
function BLX_alpha(c1, c2, alpha=0.3) //c1 y c2 indices padres
    hijos.resize(2) //Espacio para dos hijos

    Para i=0 hasta n_atrib
        c_max=max(cromosomas[c1][i], cromosomas[c2][i])
        c_min=min(cromosomas[c1][i], cromosomas[c2][i])
        I=c_max-c_min

        add1=Randfloat( c_min - I*alpha, c_max + I*alpha )
        add2=Randfloat( c_min - I*alpha, c_max + I*alpha )

        Si add1>1.0          hijos[0].aniade(1.0)
        Sino, si add1<0.0    hijos[0].aniade(0.0)
        Sino                 hijos[0].aniade(add1)

        Si add2>1.0          hijos[1].aniade(1.0)
        Sino, si add2<0.0    hijos[1].aniade(0.0)
        Sino                 hijos[1].aniade(add2)

    hijos[0].valor=f_evaluacion(hijos[0])
    hijos[1].valor=f_evaluacion(hijos[1])

    Devuelve hijos
```

El operador de cruce *Arithmetic-Crossover* se encargará de generar dos hijos mediante una media ponderada de los valores del vector de pesos de los padres, es decir, $Hijo_{1i} = \alpha * Padre_{1i} + (1-\alpha) * Padre_{2i}$ e $Hijo_{2i} = (1-\alpha) * Padre_{1i} + \alpha * Padre_{2i}$, donde $\alpha \in [0, 1]$ será generado aleatoriamente para cada i.

```
function arithmetic_crossover(c1, c2) //c1 y c2 indices padres
    hijos.resize(2) //Espacio para dos hijos
    alpha=0;

    Para i=0 hasta n_atrib
```

```
        alpha=Randfloat(0,1)
        hijos[0].aniade( alpha*cromosomas[c1][i]+
                  (1.0-alpha)*cromosomas[c2][i] )
        hijos[1].aniade( (1.0-alpha)*cromosomas[c1][i]+
                  alpha*cromosomas[c2][i] )

    hijos[0].valor=f_evaluacion(hijos[0])
    hijos[1].valor=f_evaluacion(hijos[1])

    Devuelve hijos
```

Finalmente, las mutaciones de los hijos se realizarán con la función de la práctica anterior:

```
function mutacion(pesos, index)
    pesos[i]+=distribution_normal()

    Si pesos[i]>1
        pesos[i]=1
    Si pesos[i]<0
        pesos[i]=0
```

3.2. Esquemas de evolución: AGG.

El algoritmo genético generacional con elitismo consistirá en seleccionar todos los individuos de una población, hacer parejas y cruzarlos (o no) con una probabilidad de 0,7. Tras el cruce, se realizarán mutaciones con probabilidad 0,001 y se reemplazará con la nueva generación toda la población anterior salvo el mejor de esta, que se reemplaza por el peor de la nueva generación. Esto se realizará hasta que se llame 15000 veces a la función de evaluación.

Para evitar generar más números aleatorios de los necesarios, se fijan previamente el número esperado de cruces y número esperado de mutaciones, en nuestro caso $Cruces_{esperado} = \frac{PobTotal}{2} * 0{,}7 \approx 11$ y $Mut_{esperado} = TamHijos * nAtrib * 0{,}001$ (variable para cada dataset).

Tras generar todos los hijos, la nueva población constará de los 22 nuevos individuos junto a los 8 de la población anterior que no se han reproducido; finalmente cambiamos el peor individuo de la nueva generación por el mejor de la anterior.

```
function AGG(tipo_cruce) // cruce 0=BLXalpha; cruce 1=AC
    generacion=0
    prob_cruce=0.7, prob_mut=0.001
    tam_pob=30, n_parejas=tam_pob/2
    evals=0

    generate_cromosomas(tam_pob)
    evals+=tam_pob
```

```
reordenar() //Ordena de mayor a menor los cromosomas

Mientras evals<15000
    indices=generar_indices(tam_pob) //Padres no usados

    //Seleccion
    Para i=0 hasta n_parejas*prob_cruce
        padre1=binary_tournament(indices)
        padre2=binary_tournament(indices)

        Si tipo_cruce==0
            hijos=BLX_alpha(padre1,padre2)
        Sino, si tipo_cruce==1
            hijos=arithmetic_crossover(padre1,padre2)

        evals+=2;

    //Mutacion
    Para i=0 hasta tam(hijos)*n_atrib*prob_mut
        crom_mut=Randint(0, tam(hijos)-1);
        gen_mut=Randint(0, tam(hijos)-1);
        mutacion(hijos[crom_mut], gen_mut)
        hijos[crom_mut].valor=f_evaluacion(hijos[crom_mut])
        evals++

    // Aniadimos los que no han sido padres
    Para i=0 hasta tam(indices)
        hijos.aniade(cromosomas[indices[i]])

    minimo=hijos[0].valor, ind_min=0
    Para i=1 hasta tam(hijos)
        Si hijos[i].valor < minimo
            minimo=hijos[i].valor
            ind_min=i

    hijos[ind_min]=cromosomas[0]
    cromosomas=hijos
    generacion++
    reordenar()
Devuelve cromosomas[0]
```

3.3. Esquema de evolución: AGE.

Este esquema de evolución cruzará únicamente dos padres seleccionados mediante torneo binario cada generación. Los hijos obtenidos del cruce mutarán con una probabilidad de $P_{mut} = 0{,}001 * n\_atrib$, y tras ello competirán con el resto de población, sustituyendo a los dos peores de la población actual en caso de ser mejores que ellos. Tras 15000 llamadas a la función de evaluación, se devolverá el vector de pesos del mejor individuo de la

población.

Para realizar mutaciones, se generará un valor aleatorio para cada hijo; si éste es menor que P_{mut}, se generará otro aleatorio entero entre 0 y `n_atrib-1` para determinar el gen que muta.

Para simplificar las operaciones de competición, se añaden los dos hijos a la población actual, se reordenan los cromosomas según su valor de mayor a menor y se redimensiona el vector de cromosomas a `tam_pob`.

```
function AGE(tipo_cruce) // cruce 0=BLXalpha; cruce 1=AC
   generacion=0
   pm_gen=0.001, pm_cromosoma=pm_gen*n_atrib
   tam_pob=30, evals=0

   generate_cromosomas(tam_pob)
   evals+=tam_pob
   reordenar() //Ordena de mayor a menor los cromosomas

   Mientras evals<15000
      padre1=binary_tournament()
      padre2=binary_tournament()

      Si tipo_cruce==0
         hijos=BLX_alpha(padre1,padre2)
      Sino, si tipo_cruce==1
         hijos=arithmetic_crossover(padre1,padre2)

      evals+=2;

      Si Rand() < pm_cromosoma
         mutacion(hijos[0], Randint(0, n_atrib-1))
         hijos[0].valor=f_evaluacion(hijos[0])
         evals++

      Si Rand() < pm_cromosoma
         mutacion(hijos[1], Randint(0, n_atrib-1))
         hijos[1].valor=f_evaluacion(hijos[1])
         evals++

      cromosomas.aniade(hijos)
      reordenar()
      cromosomas.resize(tam_pob)

      generacion++

   Devuelve cromosomas[0]
```

4. Algoritmos Meméticos.

Los algoritmos meméticos se basan en aplicar un algoritmo genético sobre una población de cromosomas y, cada cierto tiempo, realizar búsqueda local sobre un subconjunto (o el conjunto total) de la población.

Como veremos más tarde, el algoritmo generacional que mejores resultados nos ha dado ha sido el que utiliza *arithmetic-crossover*, por lo que usaremos este para ajustar los pesos mediante los algoritmos meméticos presentados.

4.1. Algoritmo de Búsqueda Local.

El algoritmo de búsqueda local utilizado será el mismo del que hablamos en la práctica 1, aunque tendrá ligeras modificaciones para controlar el número máximo de vecinos evaluados (en nuestro caso $2*n\_atrib$), el número de llamadas a la función de evaluación y el vector de pesos sobre el que se ejecutará la búsqueda. El código es el siguiente:

```
function busqueda_local(alpha, max_eval, &evals, pesos)
   n_generados=0, n_eval=0
   max_generados=20*n_atrib

   Si pesos.esta_vacio()   pesos=generar_solucion_inicial()

   indices=generar_indices(n_atrib)
   int i=0;
   valor_eval=0;

   Mientras n_generados < max_generados && n_eval < max_eval
      Si i==0     baraja(indices)

      tmp=pesos
      mutacion(tmp, indices[i])
      n_generados++

      new_valor=f_evaluacion(tmp)
      n_eval++

      Si new_valor>valor_eval
         pesos=tmp
         valor_eval=new_valor
         n_generados=0

      i++
      Si i==n_atrib  i=0

   evals+=n_eval  //Actualizamos el numero de evaluaciones
   Devuelve pesos
```

4.2. Algoritmo Memético.

Estudiaremos tres variantes de algoritmos meméticos:

- AM-(10,1.0): Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población.
- AM-(10,0.1): Cada 10 generaciones, se aplica la BL sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con probabilidad p LS igual a 0.1 para cada cromosoma.
- AM-(10,0.1mej): Cada 10 generaciones, aplicar la BL sobre los 0.1·N mejores cromosomas de la población actual (N es el tamaño de ésta).

Para ello, pasaremos a la función los valores de cada cuantas generaciones se aplica la búsqueda local, el porcentaje de población usada y una variable `bool mejor` que determinará si se escogen los mejores o no. Si la variable `mejor` está a `false`, usaremos un vector `usados` que contendrá los índices de los cromosomas a los que ya se le ha aplicado BL, para no aplicarla de nuevo sobre ellos. Si el porcentaje de la población sobre el que aplicar BL es 1.0, pondremos `mejor=true`, pues no será necesario usar el vector `usados`.

El código es el mismo que el de AGG (aunque solo aplica *arithmetic-crossover*), así que obviaremos ciertas partes en el pseudocódigo insertado en la memoria:

```
function AM(n_gen_bl, porcentaje_pob, mejor)
   // VARIABLES
   tam_pob=10
   [...]
   Mientras evals<15000
      Si generacion % n_gen_bl==0
         Si mejor
            Para i=0 hasta tam_pob*porcentaje_pob
               cromosomas[i]=
                  busqueda_local(0.5,2*n_atrib,evals,cromosomas[i])
               cromosomas[i].valor=f_evaluacion(cromosomas[i])
               evals++
         Sino
            Para i=0 hasta tam_pob*porcentaje_pob
               Hacer
                  j=Randint(0, tam_pob-1)
               Mientras(esta_en_usados(j))

               usados.aniade(j)
               cromosomas[j]=
                  busqueda_local(0.5,2*n_atrib,evals,cromosomas[j])
               cromosomas[j].valor=f_evaluacion(cromosomas[j])
               evals++
         reordenar()
```

```
        // SELECCION
        [...]
        // CRUCE
        [...]
        // MUTACION
        [...]
        // REEMPLAZO
        [...]
        reordenar()
        generacion++

    Devuelve cromosomas[0]
```

5. Algoritmo de Comparación: RELIEF.

Este algoritmo se encargará de ir modificando los pesos de acuerdo a la distancia de cada elemento a su amigo (elemento con la misma etiqueta) y enemigo (elemento con distinta etiqueta) más cercano. Si la distancia al amigo más cercano es mayor que al enemigo más cercano, los pesos disminuirán, y en caso contrario aumentarán.

El algoritmo comenzará inicializando el vector de pesos a 0 y recorrerá todos los elementos del conjunto de entrenamiento, buscando el enemigo y el amigo más cercano y modificando el valor de los pesos según éstos. Tras ello, normalizará los pesos coordenada a coordenada, dividiéndolos entre el máximo de todos ellos (si el valor es mayor que 1) o igualando a 0 (si el valor es menor que 0).

Las funciones para buscar los enemigos y amigos más cercanos devolverán el índice donde se encuentran éstos. El pseudocódigo para enemigos más cercanos será:

```
function close_enemy_index(e)
    indice_min=-1
    dist_min=(Valor muy grande)

    Para i=0 hasta tamanio(train)
        Si train[i].clase != e.clase
            dist_tmp=distancia_euclidea(e, train[i])
            Si d_tmp<d_min
                indice_min=i;
                dist_min=dist_tmp
    Devuelve indice_min
```

El pseudocódigo para amigos más cercanos será:

```
function close_friend_index(e, index)
    indice_min=-1
    dist_min=(Valor muy grande)
```

```
    Para i=0 hasta tamanio(train)
        Si train[i].clase == e.clase && index != i
            dist_tmp=distancia_euclidea(e, train[i])
            Si d_tmp<d_min
                indice_min=i;
                dist_min=dist_tmp
    Devuelve indice_min
```

Finalmente, el pseudocódigo del algoritmo RELIEF será:

```
function relief()
    pesos={0}
    Para i=0 hasta tamanio(train)
        enemy_ind=close_enemy_index(train[i])
        friend_ind=close_friend_index(train[i], i)
        pesos += |train[i]-train[enemy_ind]| -
                 |train[i]-train[friend_ind]|

    peso_max=maximo(pesos)

    Para j=0 hasta n_atrib
        Si pesos[j]<0
            pesos[j]=0
        Si pesos[j]>1
            pesos[j]=pesos[j]/peso_max

    Devuelve pesos
```

6. Manual de usuario.

Para replicar los resultados de mis algoritmos y el funcionamiento de mi código, habría que empezar por formatear los archivos de datos tal y como se han comentado en la subsección **Esquema de representación del problema y formateo de los ficheros de datos.** Tras ello, bastaría con compilar el código con la instrucción `make` y ejecutar el archivo `ejecutar.sh`, el cual generará un fichero `solucion.txt` con los resultados de la ejecución de los algoritmos sobre los 3 ficheros de datos. Es posible que hubiera que dar permisos de ejecución al archivo `ejecutar.sh`.

El archivo `solucion.txt` será del siguiente estilo:

```
---------------------- Datos usados: datos -------------------------

=========================================================================
=========================================================================
```

```
                                PARTICION N
=========================================================================
=========================================================================

Algoritmo:          Tasa clas: VALOR                Tasa red: VALOR
*************** F evaluacion: VALOR (tiempo: VALOR) ******************
```

7. Experimentos y análisis de resultados.

La semilla empleada para la realización de los experimentos está definida al comienzo del archivo `main.cpp`, tras los `#include`, con el código `#define SEED 10000`.

Las tablas con los resultados de la primera práctica (1-NN, BL y RELIEF) son las siguientes:

Tabla 5.1: Resultados obtenidos por el algoritmo KNN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	67,7966	0	33,8983	0,003576	85,9155	0	42,95775	0,006898	93,6364	0	46,8182	0,011131
Partición 2	78,9474	0	39,4737	0,003404	82,8571	0	41,42855	0,004034	94,5455	0	47,27275	0,011008
Partición 3	82,4561	0	41,22805	0,003391	85,7143	0	42,85715	0,004527	90,9091	0	45,45455	0,010584
Partición 4	70,1754	0	35,0877	0,003435	90	0	45	0,004011	90,9091	0	45,45455	0,010899
Partición 5	70,1754	0	35,0877	0,003516	87,1429	0	43,57145	0,004022	91,8182	0	45,9091	0,010889
Media	73,91018	0	36,95509	0,0034644	86,32596	0	43,16298	0,0046984	92,36366	0	46,18183	0,0109022

Tabla 5.2: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	83,0508	77,4194	80,2351	10,7524	94,3662	67,6471	81,00665	5,21633	90,9091	77,5	84,20455	22,9095
Partición 2	87,7193	70,9677	79,3435	9,3236	90	82,3529	86,17645	6,77597	95,4545	87,5	91,47725	46,213
Partición 3	87,7193	53,2258	70,47255	11,6677	94,2857	85,2941	89,7899	8,11535	92,7273	85	88,86365	48,1748
Partición 4	78,9474	59,6774	69,3124	6,64878	98,5714	85,2941	91,93275	7,76152	96,3636	72,5	84,4318	20,6004
Partición 5	80,7018	54,8387	67,77025	5,92223	92,8571	85,2941	89,0756	6,55822	90	67,5	78,75	17,1701
Media	83,62772	63,2258	73,42676	8,862942	94,01608	81,17646	87,59627	6,885478	93,0909	78	85,54545	31,01356

Tabla 5.3: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	69,4915	27,4194	48,45545	0,012593	84,507	2,94118	43,72409	0,015008	94,5455	2,5	48,52275	0,03811
Partición 2	78,9474	20,9677	49,95755	0,012947	82,8571	2,94118	42,89914	0,015028	93,6364	15	54,3182	0,049454
Partición 3	84,2105	16,129	50,16975	0,012867	85,7143	2,94118	44,32774	0,015125	90	2,5	46,25	0,066729
Partición 4	66,6667	19,3548	43,01075	0,012814	91,4286	2,94118	47,18489	0,015878	95,4545	15	55,22725	0,044007
Partición 5	80,7018	56,4516	68,5767	0,012803	88,5714	2,94118	45,75629	0,014961	93,6364	15	54,3182	0,039821
Media	76,00358	28,0645	52,03404	0,0128048	86,61568	2,94118	44,77843	0,0152	93,45456	10	51,72728	0,0476242

Los resultados de los distintos esquemas genéticos (AGG y AGE) con los distintos operadores de cruce (BLX-α y *arithmetic-crossover*) son los siguientes:

Tabla 5.6: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	83,0508	66,129	74,5899	47,566	95,7746	67,6471	81,71085	58,4237	91,8182	77,5	84,6591	151,4
Partición 2	84,2105	75,8065	80,0085	48,0089	81,4286	91,1765	86,30255	63,1967	90,9091	87,5	89,20455	149,845
Partición 3	91,2281	82,2581	86,7431	48,5613	88,5714	88,2353	88,40335	62,5412	88,1818	85	86,5909	153,118
Partición 4	75,4386	83,871	79,6548	49,4135	88,5714	88,2353	88,40335	71,0791	87,2727	87,5	87,38635	156,517
Partición 5	92,9825	82,2581	87,6203	47,5228	88,5714	91,1765	89,87395	61,9708	88,1818	85	86,5909	150,46
Media	85,3821	78,06454	81,72332	48,2145	88,58348	85,29414	86,93881	63,4423	89,27272	84,5	86,88636	152,268

Tabla 5.7: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	84,7458	72,5806	78,6632	46,9385	95,7746	70,5882	83,1814	60,6561	90,9091	80	85,45455	155,824
Partición 2	84,2105	75,8065	80,0085	49,6257	81,4286	91,1765	86,30255	61,8266	90,9091	87,5	89,20455	147,926
Partición 3	91,2281	82,2581	86,7431	48,5586	88,5714	88,2353	88,40335	61,3368	88,1818	85	86,5909	153,084
Partición 4	75,4386	83,871	79,6548	47,1146	88,5714	88,2353	88,40335	70,3298	87,2727	87,5	87,38635	156,393
Partición 5	94,7368	82,2581	88,49745	46,6254	88,5714	91,1765	89,87395	61,3814	88,1818	85	86,5909	148,533
Media	86,07196	79,35486	82,71341	47,77256	88,58348	85,88236	87,23292	63,10614	89,0909	85	87,04545	152,352

Tabla 5.8: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	86,4407	74,1935	80,3171	46,4569	94,3662	76,4706	85,4184	74,68	90,9091	82,5	86,70455	181,974
Partición 2	84,2105	75,8065	80,0085	50,9944	81,4286	91,1765	86,30255	68,6765	90,9091	87,5	89,20455	194,362
Partición 3	91,2281	82,2581	86,7431	51,1451	88,5714	88,2353	88,40335	70,9027	88,1818	85	86,5909	191,466
Partición 4	75,4386	83,871	79,6548	44,7123	88,5714	88,2353	88,40335	71,9709	87,2727	87,5	87,38635	185,207
Partición 5	94,7368	82,2581	88,49745	48,165	88,5714	91,1765	89,87395	57,6902	88,1818	85	86,5909	184,849
Media	86,41094	79,67744	83,04419	48,29474	88,3018	87,05884	87,68032	68,78406	89,0909	85,5	87,29545	187,5716

Tabla 5.9: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	86,4407	74,1935	80,3171	47,6467	94,3662	76,4706	85,4184	68,9439	92,7273	87,5	90,11365	189,432
Partición 2	84,2105	75,8065	80,0085	50,5087	81,4286	91,1765	86,30255	59,5182	90,9091	87,5	89,20455	179,436
Partición 3	91,2281	82,2581	86,7431	51,547	88,5714	88,2353	88,40335	70,6268	88,1818	85	86,5909	178,363
Partición 4	75,4386	83,871	79,6548	46,0295	88,5714	88,2353	88,40335	71,2303	87,2727	87,5	87,38635	175,379
Partición 5	94,7368	82,2581	88,49745	45,3299	88,5714	91,1765	89,87395	57,6817	88,1818	85	86,5909	178,508
Media	86,41094	79,67744	83,04419	48,21236	88,3018	87,05884	87,68032	65,60018	89,45454	86,5	87,97727	180,2236

Los resultados de los algoritmos meméticos (con cruce AC) son los siguientes:

Tabla 5.10: Resultados obtenidos por el algoritmo AM(10, 1.0) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	91,5254	74,1935	82,85945	58,9286	90,1408	85,2941	87,71745	70,4481	97,2727	87,5	92,38635	208,264
Partición 2	87,7193	79,0323	83,3758	58,6819	97,1429	88,2353	92,6891	75,6371	98,1818	85	91,5909	183,475
Partición 3	91,2281	82,2581	86,7431	57,9897	94,2857	88,2353	91,2605	68,6447	96,3636	87,5	91,9318	183,165
Partición 4	91,2281	83,871	87,54955	56,9676	95,7143	85,2941	90,5042	69,8789	95,4545	85	90,22725	181,799
Partición 5	94,7368	83,871	89,3039	61,9598	94,2857	91,1765	92,7311	68,9719	94,5455	82,5	88,52275	182,621
Media	91,28754	80,64518	85,96636	58,90552	94,31388	87,64706	90,98047	70,71614	96,36362	85,5	90,93181	187,8648

Tabla 5.11: Resultados obtenidos por el algoritmo AM(10, 0.1) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	91,5254	75,8065	83,66595	56,7729	90,1408	88,2353	89,18805	70,4883	97,2727	87,5	92,38635	189,151
Partición 2	92,9825	79,0323	86,0074	55,6938	97,1429	88,2353	92,6891	69,0959	98,1818	85	91,5909	180,694
Partición 3	91,2281	83,871	87,54955	54,4219	94,2857	88,2353	91,2605	76,3275	96,3636	87,5	91,9318	177,865
Partición 4	91,2281	83,871	87,54955	54,0777	97,1429	85,2941	91,2185	69,6998	95,4545	85	90,22725	176,789
Partición 5	94,7368	83,871	89,3039	54,5259	94,2857	91,1765	92,7311	69,6127	97,2727	85	91,13635	194,622
Media	92,34018	81,29036	86,81527	55,09844	94,5996	88,2353	91,41745	71,04484	96,90906	86	91,45453	183,8242

Tabla 5.12: Resultados obtenidos por el algoritmo AM(10, 0.1mej) en el problema del APC

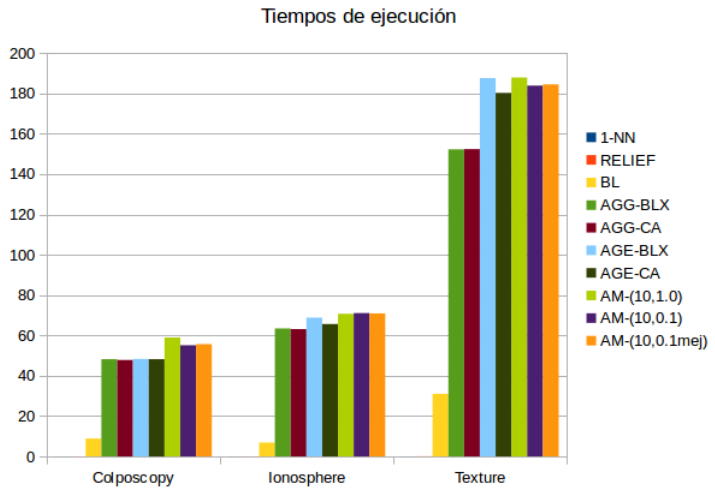
	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	91,5254	75,8065	83,66595	56,8173	94,3662	91,1765	92,77135	71,6202	97,2727	87,5	92,38635	185,092
Partición 2	91,2281	82,2581	86,7431	59,4811	97,1429	88,2353	92,6891	69,9274	98,1818	85	91,5909	182,925
Partición 3	91,2281	83,871	87,54955	54,2764	94,2857	88,2353	91,2605	73,9256	96,3636	87,5	91,9318	183,864
Partición 4	91,2281	83,871	87,54955	53,6442	92,8571	91,1765	92,0168	70,14	95,4545	85	90,22725	185,588
Partición 5	94,7368	85,4839	90,11035	54,049	92,8571	94,1176	93,48735	69,0093	97,2727	85	91,13635	184,766
Media	91,9893	82,2581	87,1237	55,6536	94,3018	90,58824	92,44502	70,9245	96,90906	86	91,45453	184,447

Finalmente, la tabla con todos los resultados (y con una entrada más correspondiente a los experimentos adicionales, AM(1,0.1)) será la siguiente:

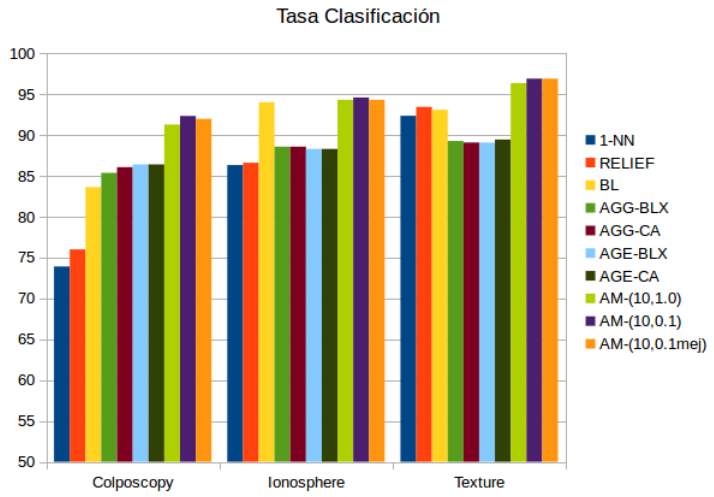
Tabla 5.14: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	73,91018	0	36,95509	0,0034644	86,32596	0	43,16298	0,0046984	92,36366	0	46,18183	0,0109022
RELIEF	76,00358	28,0645	52,03404	0,0128048	86,61568	2,94118	44,77843	0,0152	93,45456	10	51,72728	0,0476242
BL	83,62772	63,2258	73,42676	8,862942	94,01608	81,17646	87,59627	6,885478	93,0909	78	85,54545	31,01356
AGG-BLX	85,3821	78,06454	81,72332	48,2145	88,58348	85,29414	86,93881	63,4423	89,27272	84,5	86,88636	152,268
AGG-CA	86,07196	79,35	82,71	47,77256	88,58348	85,88	87,23	63,10614	89,0909	85,00	87,05	152,352
AGE-BLX	86,41094	79,68	83,04	48,29474	88,3018	87,06	87,68	68,78406	89,0909	85,50	87,30	187,5716
AGE-CA	86,41094	79,68	83,04	48,21236	88,3018	87,06	87,68	65,60018	89,45454	86,50	87,98	180,2236
AM-(10,1.0)	91,28754	80,65	85,97	58,90552	94,31388	87,65	90,98	70,71614	96,36362	85,50	90,93	187,8648
AM-(10,0.1)	92,34018	81,29	86,82	55,09844	94,5996	88,24	91,42	71,04484	96,90906	86,00	91,45	183,8242
AM-(10,0.1mej)	91,9893	82,26	87,12	55,6536	94,3018	90,59	92,45	70,9245	96,90906	86,00	91,45	184,447
AM-(1,0.1mej)	89,9316	77,10	83,51	53,97886	98,00402	87,65	92,83	67,61296	95,45454	84,00	89,73	167,8652

Comencemos comparando solamente los algoritmos concernientes a la práctica, pues los tres algoritmos de la primera práctica obtienen valores menores a los de ésta. Los tiempos no difieren demasiado respecto a cada algoritmo aplicado sobre un mismo dataset (aunque sí difieren mucho con 1-NN, RELIEF y BL); esto es debido a que el criterio de parada de todos los algoritmos ejecutados consiste en lo mismo, llamar 15000 veces a la función de evaluación. Por tanto, la diferencia de tiempos entre ellos residirá en los cruces usados o, en el caso de los meméticos, las ejecuciones de búsqueda local. Además, como era de esperar, los algoritmos meméticos obtienen mejores resultados que los algoritmos genéticos, pues la búsqueda local optimiza la mejora de la población.



Veamos con una gráfica generada sobre las tablas de Excel una comparación entre los valores de tasa de clasificación de cada uno de los algoritmos.



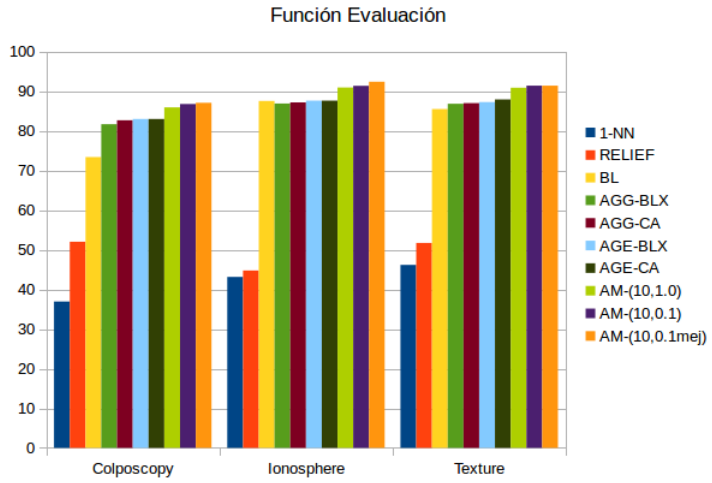
Podemos ver que, entre los distintos algoritmos genéticos, se tiende a obtener el mismo valor de tasa de clasificación (aunque en el dataset `colcoscopy` la tasa de acierto es un tanto menor para AGG-BLX), por tanto no podemos discernir por un claro ganador, ya que además los tiempos de ejecución son prácticamente los mismos. Además, podríamos optar por prescindir de los algoritmos genéticos en el conjunto de datos `ionosphere` y escoger búsqueda local, ya que la tasa de acierto de BL sobre este dataset es mayor que la de los algoritmos genéticos y el tiempo de ejecución de BL es mucho menos a AGG y AGE.

Respecto a los algoritmos meméticos, obtienen mayor acierto que los algoritmos genéticos, y la opción AM(10, 0.1), consistente en aplicar búsqueda local cada 10 generaciones a un 10 % de la población escogida aleatoriamente, gana (por poco) a los otros dos tipos de AM ejecutados.

Veamos ahora la tasa de reducción:

La tasa de reducción se mantiene más o menos al mismo nivel en todos los algoritmos ejecutados en esta práctica, tendiendo a crecer en el orden de ejecución de estos, es decir, el cruce BLX-α obtiene peores resultados que *arithmetic-crossover*, los algoritmos generacionales tienden a ser peores que los estacionarios, y los algoritmos meméticos son mejores que los genéticos (todo esto hablando de tasa de reducción).

Finalmente, comentemos los datos de la función de evaluación:



Simplemente con los valores de la tasa de clasificación y de reducción comentados anteriormente ya deberíamos de haber adivinado la forma de esta gráfica final, pues el valor de tasa de clasificación y de reducción de los algoritmos meméticos supera siempre a los genéticos. Al igual que en la tasa de reducción, CA es mejor que BLX, AGE mejor que

AGG y los algoritmos meméticos mejores que los genéticos, y todos estos mejores que 1-NN, RELIEF y BL (salvo en `ionosphere`, que la BL supera a los AG).

Si nos fijamos en los algoritmos meméticos, la mejor opción a ejecutar consistiría en que cada 10 generaciones se aplique búsqueda local al 10 % mejor de nuestra población. Estos mejores resultados son debidos al elitismo, pues en AM(10,0.1) aplicamos la BL sobre un individuo aleatorio de la población, el cuál puede resultar ser el que peor valor tenga, y no se asegura que la búsqueda local mejore considerablemente su función de evaluación. Por otra parte, en AM(10,1.0) estamos aplicando búsqueda local a todos los individuos de la población, lo cual puede ser una buena idea al comienzo de la población, pero a lo largo de las generaciones los valores de todos los cromosomas tienden a ser el mismo, por lo que al aplicar BL sobre todos ellos estamos repitiendo la misma búsqueda local sobre todos los elementos, lo cual empeora los resultados, pues estamos aumentando el número de evaluaciones sobre la función de evaluación innecesariamente, que es un valor finito e igual para todos los algoritmos meméticos independientemente de como realizar la búsqueda local sobre la población.

7.1. Experimentos adicionales.

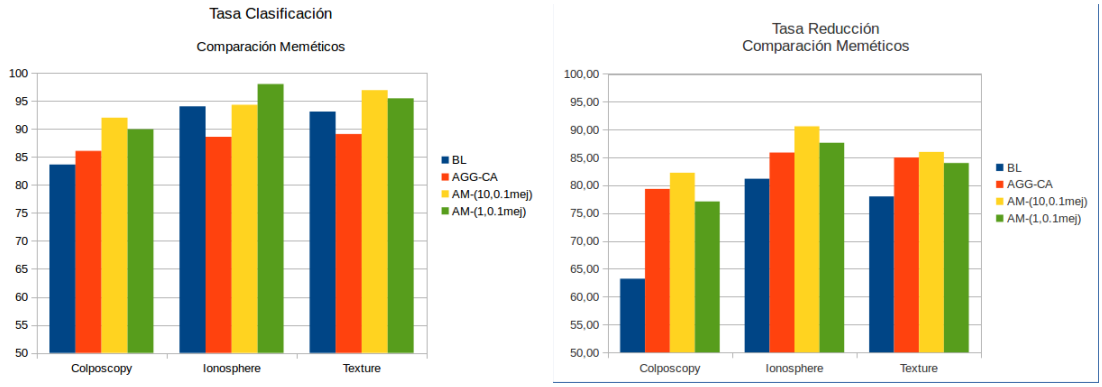
¿Qué resultados obtendríamos si cada generación aplicáramos búsqueda local sobre el 10 % de los mejores cromosomas de la población? Podemos suponer que se obtendrá un valor relacionado con AGG-CA y BL, pues todas las generaciones se ejecutan estos dos algoritmos. Los resultados por particiones de AM(1,0.1mej) serán los siguientes:

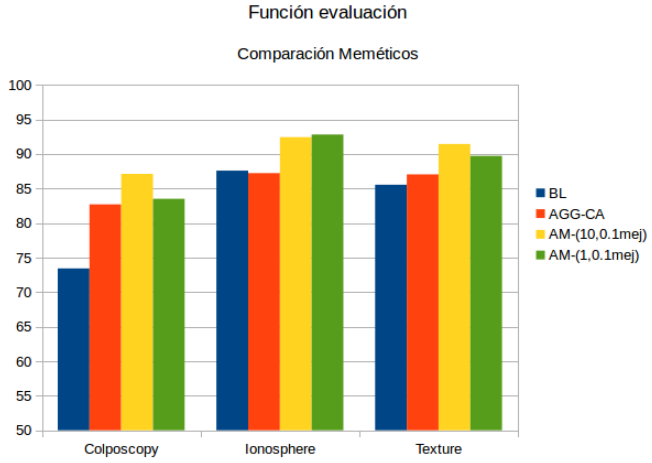
Tabla 5.13: Resultados obtenidos por el algoritmo AM(1, 0.1mej) en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	84,7458	70,9677	77,85675	52,7905	98,5915	88,2353	93,4134	61,2692	93,6364	87,5	90,5682	153,225
Partición 2	91,2281	75,8065	83,5173	57,077	100	85,2941	92,64705	69,5416	96,3636	82,5	89,4318	175,293
Partición 3	94,7368	77,4194	86,0781	53,9841	98,5714	88,2353	93,40335	67,8323	94,5455	85	89,77275	172,615
Partición 4	82,4561	79,0323	80,7442	51,9422	95,7143	85,2941	90,5042	68,7789	95,4545	82,5	88,97725	169,098
Partición 5	96,4912	82,2581	89,37465	54,1005	97,1429	91,1765	94,1597	70,6428	97,2727	82,5	89,88635	169,095
Media	89,9316	77,0968	83,5142	53,97886	98,00402	87,64706	92,82554	67,61296	95,45454	84	89,72727	167,8652

La media de estos resultados se encuentra en la tabla de la imagen **??**, por lo que podemos comparar los resultados desde ahí.

Ya que los algoritmos implicados en AM son AGG-CA y BL, comparemos AM(10,0.1mej) y AM(1,0.1mej) con éstos.





El hecho de realizar la búsqueda local en cada generación hace que AM(1,0.1mej) obtenga unos resultados más parecidos[1] a los de la BL que a los de AGG-CA; esto se puede comprobar fácilmente si nos fijamos en el dataset `ionosphere`, el cual aporta una tasa de clasificación más alta de lo normal a la búsqueda local, y por tanto en este caso AM(1,0.1mej) obtiene mejores resultados que AM(10,0.1mej). De esto podemos deducir que, cuantas más generaciones esperemos para ejecutar la búsqueda local, más se parecerán los resultados al algoritmo genético, mientras que cuantas menos generaciones esperemos, más se parecerán los resultados a BL.

¿Por qué no vemos este mismo patrón en la tasa de clasificación sobre `texture`? Porque la tasa de reducción de la BL es considerablemente menor que la de AGG-CA, y el algoritmo memético no solo tiene en cuenta la tasa de clasificación para ajustar los pesos, sino que usa el agregado de las dos tasas para optimizar generación tras generación, lo que hace que la tasa de clasificación de AM(1,0.1mej) no supere a la de AM(10,0.1mej) en el dataset `texture`.

[1] Con parecidos me refiero a que cuando la búsqueda local es buena, AM(1,0.1mej) es bueno y cuando AGG-CA es bueno, AM(10,0.1mej) es bueno.

Notemos que el parecido entre BL y AM(1,0.1mej) y entre AGG-CA y AM(10,0.1mej) no se daría si el número de llamadas a la función de evaluación no se modificase dentro de la búsqueda local, lo que supondría un mayor tiempo de ejecución para AM(1,0.1mej) sin garantías de una mejor solución.

Tras este análisis, podemos concluir que el uso del algoritmo memético generacional basado en AC, AM(10,0.1mej), es una buena opción para ajustar el Problema de Aprendizaje de Pesos en Características sobre un dataset genérico, pero se pueden obtener resultados mayores realizando un previo estudio de nuestro conjunto datos y ajustando las variables que determinan cada cuantas generaciones se realiza la búsqueda local y el porcentaje de población sobre el que queremos aplicar BL.