Universidad de Granada

Grado en Ingeniería Informática. Tercero.

Problema del Aprendizaje de Pesos en Características (APC).

Nombre de la asignatura:
Metaheurísticas. Grupo 2: Viernes de 17:30 a 19:30.

Realizado por:
Néstor Rodríguez Vico. DNI: 75573052C.
email: nrv23@correo.ugr.es

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS
INFORMÁTICA Y DE TELECOMUNICACIÓN.

Granada, 10 de abril de 2017.

Índice

1. Descripción del problema.

El problema del Aprendizaje de Pesos en Características (APC) es un problema de búsqueda con codificación real en el espacio *n-dimensional*, para n características. Consiste obtener un vector de pesos que permita ponderar las características asociadas a un dato con la intención de obtener un mejor porcentaje de clasificación de datos futuros. Tenemos una muestra de datos $X = \{x_1, x_2, ..., x_n\}$, cada dato del conjunto x_i está formado por un conjunto de características $\{c_1, c_2, ..., c_m, CLASE\}$. El problema consiste en obtener un vector de pesos $W = \{w_1, w_2, ..., w_m\}$ donde el peso w_i pondera la característica c_i y que ese vector de pesos nos permita obtener un alto porcentaje en otro conjunto de datos desconocido hasta la comprobación de la calidad de nuestro vector de pesos W.

A lo largo de nuestro proceso vamos a realizar dos tareas: primero obtendremos el vector de pesos (*aprendizaje*) y a continuación valoraremos la calidad del mismo (*validación*). Para obtener unos resultados mas fiables en esta tarea vamos a usar la técnica de validación llamada *5x2 cross validation*, es decir, usaremos 5 particiones distintas de datos al 50 %. Aprenderemos con una mitad de los datos (conjunto al que me voy a referir como *train*) y validaremos con la otra mitad (conjunto al que me voy a referir como *test*). Luego repetiremos el proceso a la inversa, es decir, usando como conjunto de entrenamiento el conjunto *test* y usando como conjunto de validación el conjunto *train*.

Como clasificador se he implementado el *kNN* (con k=1), el cual a cada dato le asigna la clase del vecino más cercano a dicho dato. El vecino más cercano será aquel con el cual se obtenga la menor *distancia euclídea* calculada de la siguiente forma:

$$d_e(e_1, e_2) = \sqrt{\sum_{n}^{i=1}(e_1^i - e_2^i)^2} \tag{1}$$

Nuestra función de evaluación va a ser el rendimiento del clasificador *1-NN* usando el vector de pesos W calculado por cada algoritmo y el posterior cálculo del porcentaje asociado. La tasa de clasificación se calcula como:

$$tasa\_clas = 100 * \frac{n^\circ\ instancias\ bien\ clasificadas\ de\ T}{n^o\ instancias\ en\ T} \tag{2}$$

Se ha implementado el clasificador *1-NN*, un algoritmo *Greedy* (algoritmo *Relief*), un algoritmo de búsqueda local del primer mejor, cuatro algoritmos genéticos y tres algoritmos meméticos. Los distintos algoritmos se van a ejecutar sobre tres conjuntos de datos distintos; *sonar*, *spambase* y *wdbc*. Debemos tener en cuenta que debemos normalizar los datos. Todo el proceso de normalización de los datos y partición de los conjuntos lo he realizado en *R*. [1]

[1]He adjuntado el fichero .R, llamado *GenerarDatos.R* para que se puedan generar las particiones.

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

Una solución a nuestro problema es un vector de pesos, W, que permite realizar una clasificación de calidad de un conjunto de datos futuros. Una solución para nuestro problema es un vector de n componentes, donde n es el número de características de cada dato de nuestro conjunto. En cada componente tendremos un valor real $w_i \in [0, 1]$:

- Un valor 1 en la componente w_i indica que la característica i se considera completamente en el cálculo de la distancia.
- Un valor 0 en la componente w_i indica que la característica i no se considera en el cálculo de la distancia.
- Cualquier otro valor intermedio gradúa el peso asociado a cada característica y pondera su importancia.

De una manera gráfica tendríamos lo siguiente:

$W =$	w_1	w_2	$\cdots$	w_{n-1}	w_n
	$\downarrow$	$\downarrow$	$\cdots$	$\downarrow$	$\downarrow$
$x_i =$	c_1	c_2	$\cdots$	c_{n-1}	c_n

En el caso de algoritmos que sólo traten con una solución usarán un vector como el descrito. En algoritmos que usen una población de soluciones, como son los algoritmos genéticos y meméticos, trataremos con un vector de vectores solución (lo que se podría ver como una matriz en la cual cada fila es una solución, es decir, un vector de pesos W).

Como función objetivo vamos medir el rendimiento del clasificador *1-NN* con los pesos calculados por cada algoritmo. La forma de proceder va a ser siempre la misma:

1. Ejecutamos el algoritmo para un conjunto de datos de entrenamiento, *train* y obtenemos un vector de pesos, *W*.
2. Llamamos al clasificador *1-NN* con el conjunto de prueba, *test*, y el vector de pesos obtenido por el algoritmo, *W*, el cual nos devuelve el conjunto de etiquetas, *clasificacion*, asociado a los datos del conjunto *test*.
3. Comparamos las etiquetas calculadas por el clasificador, *clasificacion*, con las etiquetas reales del conjunto *test*, *clases_test* y medimos un porcentaje de acierto.

El pseudocódigo del clasificador *1-NN* es el siguiente:

También he aportado los conjuntos de datos dentro de la carpeta *Data*, los cuales son leídos por mi algoritmos.

```
1: function KNN(train, clases_train, conjunto_test, pesos)
2:     for i = 0, i < test.size(), i += 1 do
3:         vecino_mas_cercano = vecinoMasCercano(train, test[i], i, pesos)
4:         clasificacion[i] = clases_train[vecino_mas_cercano]
5:     end for
6: end function
7:     ▷ El argumento clasificacion es modificado y al finalizar el algoritmo contiene la clasificación calculada
```

La manera en la que se calcula el vecino más cercano es trivial. Se calcula en base a la fórmula de la *distancia Euclídea*. El vecino elegido será el que consiga una mejor distancia Euclídea con respecto al dato que estamos clasificando. El pseudocódigo de este proceso es el siguiente:

```
1: function vecinoMasCercano(datos, actual, posicion, pesos)
2:     mejor_distancia = 9999
3:     for i = 0, i < datos.size(), i += 1 do
4:         if i ≠ posicion then                                  ▷ leave_one_out
5:             distancia_actual = distanciaEuclidea(datos_i, actual)
6:             if distancia_actual < mejor_distancia then
7:                 mejor_distancia = distancia_actual
8:                 vecino_mas_cercano = i
9:             end if
10:        end if
11:    end for
12:    return clasificacion
13: end function
```

A la hora de calcular la distancia Euclídea debemos aplicar la técnica *leave_one_out*, la cual consiste en no tener en cuenta como posible vecino más cercano el propio dato que estamos procesando. Si no se aplicase el vecino más cercano de cada dato sería el mismo y por tanto la distancia sería 0. Basta con comprobar que los identificadores de los dos datos que estamos procesando sean distintos. Si pensamos sobre como se han generado los conjuntos de datos, vemos que un dato no puede estar en *train* y *test* simultáneamente, así que parece que no es necesario aplicar *leave_one_out*. Pero debemos tener en cuenta que nuestros algoritmos van a conocer el conjunto *test* en el proceso de cálculo del vector de pesos W y que cuando llamemos a nuestro clasificador *1-NN* dentro de los algoritmos lo le pasaremos el conjunto que recibe nuestro algoritmo, *train*, como conjunto de entrenamiento y conjunto de prueba. Aquí es cuando surge la necesidad de aplicar *leave_one_out*.

Una vez hemos obtenido la clasificación de los datos de *test* en función del vector de pesos calculado, debemos evaluar como de buena ha sido esta clasificación. Para ello simplemente comparamos la clasificación obtenida con la clasificación real (la que hemos leído desde los conjuntos de datos) y ver que porcentaje se han clasificado correctamente,

tal y cómo nos indica la fórmula *tasa_class*. A mayor tasa de clasificación, mejor es el vector de pesos W generado por nuestro algoritmo.

Todo algoritmo parte de una solución inicial (o un conjunto de soluciones iniciales). Estas se van a generar de manera aleatoria, creando un vector W donde para cada componente se genera un número aleatorio en el rango [0, 1]. Sea cual sea el algoritmo debemos generar soluciones nuevas o modificar la que ya tenemos, por lo tanto debemos tener un operador de generación de vecino/mutación. Para ello vamos a modificar el gen i del cromosoma j siguiendo una distribución normal de media 0 y varianza σ^2. Debemos tener en cuenta que esta distribución puede generar valores negativos, los cuales debemos truncar a 0 ya que, en nuestro problema, no tiene sentido un peso por debajo de 0. Para ello tenemos la siguiente función:

```
1: function TRUNCAR(numero)
2:     salida = numero
3:     if numero < 0 then
4:         salida = 0
5:     else if numero > 1 then
6:         salida = 1
7:     end if
8:     return salida
9: end function
```

En los algoritmos genéticos la población inicial va a ser un conjunto de 30 soluciones iniciales generadas de manera aleatoria mientras que en el caso de los algoritmos meméticos el tamaño de la población es de 10 individuos. El proceso de selección de los padres a cruzar se va a realizar usando *torneo binario*, el cual genera dos número aleatorios, el índice de dos padres de la población, y se queda con el mejor individuo de los dos. El pseudocódigo es el siguiente:

```
1: function TORNEOBINARIO(poblacion, pcts_poblacion)
2:     uno = aleatorio ∈ [0, 29]              ▷ Si es memético: uno = aleatorio ∈ [0, 9]
3:     repeat
4:         dos = aleatorio ∈ [0, 29]          ▷ Si es memético: dos = aleatorio ∈ [0, 9]
5:     until uno ≠ dos
6:     if pcts_poblacion[uno] ¿pcts_poblacion[dos] then
7:         dos = uno
8:     end if
9:     return dos
10: end function
```

Aplicando *torneo binario* generamos los padres que luego van a ser cruzados. Vamos a tener dos operadores de cruce: BLX-α (con $\alpha = 0{,}3$) y el cruce aritmético:

- **BLX-α (con $\alpha = 0{,}3$)**: Este operador de cruce genera dos descendientes, llamémoslos $hijo_1$ e $hijo_2$. La idea general es asignar a cada componente del $hijo_i$ un valor

en el rango $[min(caracteristica_i\_padre_1, caracteristica_i\_padre_2) - tam\_intervalo * \alpha, max(caracteristica_i\_padre_1, caracteristica_i\_padre_2) + tam\_intervalo * \alpha]$. El pseudocódigo que genera estos dos hijos es el siguiente:

```
1: function cruceBLX(padre_1, padre_2)
2:     for i = 0, i < padre_1.size(), i += 1 do
3:         max = max(padre_1[i], padre_2[i])
4:         min = min(padre_1[i], padre_2[i])
5:         intervalo = max - minimo
6:         numero = random ∈ [min - intervalo * 0.3, max + intervalo * 0.3]
7:         numero2 = truncar(numero)
8:         hijo_1[i] = numero2
9:         numero = random ∈ [min - intervalo * 0.3, max + intervalo * 0.3]
10:        numero2 = truncar(numero)
11:        hijo_2[i] = numero2
12:    end for
13:    return <hijo_1, hijo_2>
14: end function
```

- **Cruce aritmético:** Este operador de cruce genera un único descendiente cuya característica i es la media de la característica i del primer padre y la característica i del segundo padre. El pseudocódigo es el siguiente:

```
1: function cruceAritmetico(padre_1, padre_2)
2:     for i = 0, i < padre_1.size(), i += 1 do
3:         hijo[i] = (padre_1[i] + padre_2[i])/2
4:     end for
5:     return hijo
6: end function
```

El operador de mutación empleado es común para los algoritmos genéticos y meméticos. Coincide con la manera en la que se genera un nuevo vecino explicada anteriormente. Mutaremos el gen i del cromosoma j usando una distribución normal de media 0 y $\sigma^2 = 0{,}09$ teniendo en cuenta que ese valor se debe truncar si no cumple las condiciones de nuestro problema. El pseudocódigo es el siguiente:

```
1: function mutarCromosoma(cromosoma, gen_mutar)
2:     cromosoma[gen_mutar] += random ∈ distribucion_normal(0, 0.09)
3:     numero = truncar(cromosoma[gen_mutar])
4:     cromosoma[gen_mutar] = numero
5: end function
```

3. Pseudocódigo de los algoritmos.

Una aclaración para todos los algoritmos descritos a continuación: ciertos algoritmos no devuelven nada de forma explicita (usando *return*) sino que modifican los parámetros que reciben cuando son llamados. Esto está hecho por comodidad de no tener que declarar las variables que recogen las salidas de los algoritmos y porque ciertos algoritmos deben "devolver" varias cosas, como puede ser el vector de pesos calculado y el número de evaluaciones empleadas, siendo imposible hacer un *return* de varios objetos sin tener que crear una estructura auxiliar que los almacene.

3.1. Algoritmo Greedy: Relief.

```
1: function RELIEF(train, clases, pesos)
2:     pesos = {0, 0, 0, ... , 0}
3:     for i = 0, i < train.size(), i += 1 do                ▷ Para todos las muestras
4:         amigo_enemigoCercano(train, clases, train[i], i, amigo, enemigo)
5:         actualizarPesos(pesos, amigo, enemigo, train[i])
6:     end for
7:     max_peso = elemento mayor de pesos  ▷ Llamada a una función de una librería
8:     for i = 0, i < pesos.size(), i += 1 do                ▷ Para todos los pesos
9:         if pesos[i] < 0 then
10:            pesos[i] = 0
11:        else
12:            pesos[i] = pesos[i] / max_peso
13:        end if
14:    end for
15: end function
```

```
1: function AMIGO_ENEMIGOCERCANO(train, clases, actual, posicion, amigo, enemigo)
2:     mejor_distancia_amigo = mejor_distancia_enemigo = 9999
3:     for i = 0, i < train.size(), i += 1 do                ▷ Para todos los datos
4:         distancia_actual = distanciaEuclidea(actual, train[i])
5:         if clases[i] == clases[posicion] then
6:             if posicion ≠ i && distancia_actual < mejor_distancia_amigo then
7:                 mejor_distancia_amigo = distancia_actual
8:                 mejor_amigo = i
9:             end if
10:        else
11:            if posicion ≠ i && distancia_actual < mejor_distancia_enemigo then
12:                mejor_distancia_enemigo = distancia_actual
13:                mejor_enemigo = i
14:            end if
15:        end if
16:    end for
```

```
17:     amigo = train[mejor_amigo]
18:     enemigo = train[mejor_enemigo]
19: end function
```

Esta función lo que hace es calcular el amigo y el enemigo más cercano a un dato dado, *actual*, en función de la *distancia Euclídea*. Un dato se considera *enemigo* si tiene la clase distinta y *amigo* si tiene la misma clase que el datos que estamo considerando.

```
1: function actualizarPesos(pesos, amigo, enemigo, actual)
2:     for i = 0, i < pesos.size(), i += 1 do
3:         pesos[i] = pesos[i] + abs(actual[i] - enemigo[i]) - abs(actual[i] - amigo[i])
4:     end for
5:     return hijo
6: end function
```

El vector de pesos se actualiza en función del *amigo* y el *enemigo* más cercano del dato que estamos procesando, *actual*.

3.2. Búsqueda Local.

```
1: function localSearch(train, clases_train, pesos, num_eval)
2:     indices = {1, 2, 3, ..., num_caracteristicas}
3:     pesos = solInicialAleatoria(num_caracteristicas)
4:     KNN(train, clases_train, train, clasificacion, pesos)
5:     porcentaje_ant = tasaAcierto(clases_train, clasificacion)
6:     i_aux = 0
7:     while num_eval < 15000 && num_vecinos < 20 * num_caracteristicas do
8:         mejora = false
9:         for i = 0, i < num_caracteristicas && !mejora, i += 1 do
10:            sol_nueva = pesos
11:            elegido = random ∈ [i_auxiliar, indices.size() - 1]
12:            posicion_auxiliar = indices[i_auxiliar]
13:            indices[i_auxiliar] = indices[elegido]
14:            indices[elegido] = posicion_auxiliar
15:            modificarPeso(sol_nueva, indices[i_auxiliar])
16:            num_vecinos += 1
17:            i_auxiliar += 1
18:            if i_auxiliar == num_caracteristicas then
19:                i_auxiliar = 0          ▷ Se han cambiado todas, empezamos de cero
20:            end if
21:            KNN(train, clases_train, train, clasificacion, sol_nueva)
22:            porcentaje_nuevo = tasaAcierto(clases_train, clasificacion)
23:            num_eval += 1
```

```
            if porcentaje_nuevo > porcentaje_ant then
                pesos = sol_nueva
                porcentaje_ant = porcentaje_nuevo
                mejora = true
                num_vecinos = 0
            end if
        end for
    end while
    return pesos
end function
```

La búsqueda local implementada se trata de *búsqueda primero el mejor*. El vector *indices* nos indica en que orden se van a modificar las componentes. En cada paso queremos modificar una componente aleatoria que no hayamos modificado antes, es decir, que esté desde la última que se modifico en *índices*, *i_auxiliar* y la última que se modifico, *indices.size() - 1*. De esta manera tenemos una forma aleatoria de modificar las componentes.

```
function solInicialAleatoria(size)
    for i = 0, i < size, i += 1 do
        solucion_inicial[i] = random ∈ [0, 1]
    end for
    return solucion_inicial
end function
```

```
function modificarPeso(cromosoma, gen_mutar)
    cromosoma[gen_mutar] += random ∈ distribucion_normal(0, 0.09)
    numero = truncar(cromosoma[gen_mutar])
    cromosoma[gen_mutar] = numero
end function
```

```
function tasaAcierto(correctas, calculadas)
    for i = 0, i < num_caracteristicas, i += 1 do
        if correctas[i] == calculadas[i] then
            correctos += 1
        end if
    end for
    porcentaje = (correctos * 1.0 / num_caracteristicas)*100.0
    return porcentaje
end function
```

La función *tasaAcierto* simplemente calcula cuantas etiquetas calculadas por nuestro clasificador *1-NN* en función del vector de pesos, *W*, son correctas.

3.3. Algoritmos genéticos generacionales.

Un par de aclaraciones para estos algoritmos:

- El tamaño de la población es de 30 individuos.
- La forma en la que se mantiene ordenada la población por tema de eficiencia es usando dos vectores de índices, *index_mejores* y *index_peores*. Ambos contienen el índice de todos los elementos de la población ordenados según su porcentaje, usando el vector *pcts_** (el nombre depende de si se trata de la población original, los padres o los hijos, pero todos comienzan por *pct_*). *index_mejores* ordena de mejor a peor (de mayor a menor porcentaje) mientras que *index_peores* ordena de peor a mejor (de menor a mayor porcentaje). Se tiene dos índices para usar el más adecuado en cada momento y actualizar cada uno de ellos en su momento correcto.
- Debemos aplicar *elitismo*, es decir, mantener el mejor individuo de la población en la generación anterior si en la nueva generación el mejor individuo es peor que el mejor individuo de la generación anterior. En este caso nos resulta útil tener los dos índices mencionados en el punto anterior para no tener que buscar cuales son dichos individuos.
- Tenemos un vector de variables booleanas auxiliar llamado *flag_eval*. La componente i de dicho vector valdrá *true* si se ha evaluado el cromosoma i de la población y *false* en caso contrario. Este vector nos permite ahorrarnos evaluaciones innecesarias, las cuales son limitadas, y aprovecharlas en generaciones futuras.
- Para el proceso de mutación se calcula cuantos genes se producen en cada generación y en base a eso se decide cada cuantas generaciones se debe mutar para mutar uno de cada mil genes producidos.

3.3.1. Algoritmo genético generacional con cruce BLX-α.

```
1:  function GeneticoGeneracionalBLX(train, clases_train)
2:      num_caracteristicas = train[0].size()
3:      index_mejores = {1, 2, 3, ..., num_caracteristicas}
4:      index_peores = {1, 2, 3, ..., num_caracteristicas}
5:      poblacion = poblacionInicial(num_caracteristicas)
6:      evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval, num_eval)
7:      genes_generacion = 30 * num_caracteristicas
8:      mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
9:      sort(index_mejores segun pcts_poblacion)  ▷ Mantenemos ordenada la poblacion
10:     while num_eval < 15000 do
11:         for i = 0, i < 30, i += 1 do
12:             elegido = torneoBinario(poblacion, pcts_poblacion)
13:             padres[i] = poblacion[elegido]
```

```
        pcts_padres[i] = pcts_poblacion[elegido]
    end for
    num_cruces = 0.7*30                                  ▷ Proceso de cruce
    for i = 0, i < num_cruces, i+=2 do
        hijosBLX = cruceBLX(padres[i], padres[i + 1])
        hijos[i] = hijosBLX.first
        hijos[i + 1] = hijosBLX.second
        flag_eval[i] = false
        flag_eval[i + 1] = false
        pcts_hijos[i] = 0
        pcts_hijos[i + 1] = 0
    end for
    for i = num_cruces, i < 30, i += 1 do
        hijos[i] = padres[i]
        pcts_hijos[i] = pcts_padres[i]
    end for
    if generacion % mutar == 0 then                      ▷ Proceso de mutacion
        cromosoma_mutar = random ∈ [0, 30)
        gen_mutar = random ∈ [0, num_caracteristicas)
        mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
        flag_eval[cromosoma_mutar] = false
    end if
    evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
    sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
    mejor = poblacion[index_mejores[0]]          ▷ Guardamos el mejor (elitismo)
    pct_mejor = pcts_poblacion[index_mejores[0]]
    poblacion = hijos
    pcts_poblacion = pcts_hijos
    if pct_mejor > pcts_poblacion[index_peores[index_peores.size() - 1]] then
        poblacion[index_peores[0]] = mejor                          ▷ Elitismo
        pcts_poblacion[index_peores[0]] = pct_mejor
    end if
    sort(index_mejores segun pcts_poblacion)
    generacion += 1
  end while
  return poblacion[index_mejores[0]]
end function
```

```
function POBLACIONINICIAL(num_caracteristicas)
  for i = 0, i < 30, i += 1 do
      poblacion_inicial = solInicialAleatoria(num_caracteristicas)
  end for
  return poblacion_inicial
end function
```

```
1: function evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval,
   num_eval)
2:     for i = 0, i < poblacion.size(), i += 1 do
3:         if !flag_eval[i] then
4:             KNN(train, clases_train, train, clasificacion, poblacion[i])
5:             num_eval += 1
6:             pcts_poblacion[i] = tasaAcierto(clases_train, clasificacion)
7:             flag_eval[i] = true
8:         end if
9:     end for
10: end function
```

Teniendo en cuenta el vector *flag_eval* mencionado al principio de esta sección se evalúa de nuevo la población para obtener los porcentajes correctos. Este proceso de evaluación aumenta el número de evaluaciones empleadas, por eso es interesante el uso del vector *flag_eval*.

3.3.2. Algoritmo genético generacional con cruce aritmético.

```
1: function GeneticoGeneracionalCA(train, clases_train)
2:     num_caracteristicas = train[0].size()
3:     hijos_generados = 0
4:     auxiliar = 0
5:     index_mejores = {1, 2, 3, ..., num_caracteristicas}
6:     index_peores = {1, 2, 3, ..., num_caracteristicas}
7:     poblacion = poblacionInicial(num_caracteristicas)
8:     evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval, num_eval)
9:     genes_generacion = 30 * num_caracteristicas
10:    mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
11:    sort(index_mejores segun pcts_poblacion)  ▷ Mantenemos ordenada la poblacion
12:    while num_eval < 15000 do
13:        for i = 0, i < 60, i += 1 do
14:            elegido = torneoBinario(poblacion, pcts_poblacion)
15:            padres[i] = poblacion[elegido]
16:            pcts_padres[i] = pcts_poblacion[elegido]
17:        end for
18:        num_cruces = 0.7*30                                  ▷ Proceso de cruce
19:        for i = 0, i < num_cruces, i+=1 do
20:            hijos[i] = cruceAritmetico(padres[auxiliar], padres[auxiliar+1])
21:            hijos_generados +=1
22:            flag_eval[i] = false
23:            pcts_hijos[i] = 0
24:            auxiliar += 2
25:        end for
```

```
26:         for i = num_cruces, i < 30, i += 1 do
27:             hijos[i] = padres[i]
28:             pcts_hijos[i] = pcts_padres[i]
29:         end for
30:         if generacion % mutar == 0 then                    ▷ Proceso de mutacion
31:             cromosoma_mutar = random ∈ [0, 30)
32:             gen_mutar = random ∈ [0, num_caracteristicas)
33:             mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
34:             flag_eval[cromosoma_mutar] = false
35:         end if
36:         evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
37:         sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
38:         mejor = poblacion[index_mejores[0]]          ▷ Guardamos el mejor (elitismo)
39:         pct_mejor = pcts_poblacion[index_mejores[0]]
40:         poblacion = hijos
41:         pcts_poblacion = pcts_hijos
42:         if pct_mejor > pcts_poblacion[index_peores[index_peores.size() - 1]] then
43:             poblacion[index_peores[0]] = mejor                              ▷ Elitismo
44:             pcts_poblacion[index_peores[0]] = pct_mejor
45:         end if
46:         sort(index_mejores segun pcts_poblacion)
47:         generacion += 1
48:     end while
49:     return poblacion[index_mejores[0]]
50: end function
```

La filosofía de este algoritmo es igual que el anterior, simplemente cambia el operador de cruce. Este nuevo operador, *cruce aritmético*, genera un único hijo en vez de dos y por lo tanto debemos tener el doble de padres para generar el mismo número de hijos, ya que si no la población reduciría de tamaño en cada generación.

3.4. Algoritmos genéticos estacionarios.

Un par de aclaraciones para estos algoritmos:

- El tamaño de la población es de 30 individuos.
- La forma en la que se mantiene ordenada la población por tema de eficiencia es usando el vector índices *index_peores*. Contiene el índice de todos los elementos de la población ordenados según su porcentaje, usando el vector *pcts_** (el nombre depende de si se trata de la población original, los padres o los hijos, pero todos comienzan por *pct_*). *index_peores* ordena de peor a mejor (de menor a mayor porcentaje).
- No debemos aplicar *elitismo*, simplemente los dos hijos generados compiten contra los peores de la población para entrar.

- No existe probabilidad de cruce, siempre se cruzan los padres para generar dos hijos.
- En el caso de los algoritmos genéticos generacionales teníamos un vector de variables booleanas auxiliar llamado *flag_eval*. Por la filosofía de los algoritmos estacionarios este vector ya no es necesario dado que la población no se tiene que volver a evaluar, sólo se evalúan los dos hijos candidatos a entrar en la población.
- Para el proceso de mutación se calcula cuantos genes se producen en cada generación y en base a eso se decide cada cuantas generaciones se debe mutar para mutar uno de cada mil genes producidos.

3.4.1. Algoritmo genético estacionario con cruce BLX-α.

```
1: function GeneticoEstacionarioBLX(train, clases_train)
2:     num_caracteristicas = train[0].size()
3:     index_peores = {1, 2, 3, ..., num_caracteristicas}
4:     poblacion = poblacionInicial(num_caracteristicas)
5:     evalPoblacionEstacionario(train, clases_train, poblacion, pcts_poblacion,
   num_eval)
6:     genes_generacion = 2 * num_caracteristicas
7:     mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
8:     sort(index_peores segun pcts_poblacion)    ▷ Mantenemos ordenada la poblacion
9:     while num_eval < 15000 do
10:        elegido1 = torneoBinario(poblacion, pcts_poblacion)
11:        padre_1 = poblacion[elegido]
12:        elegido2 = torneoBinario(poblacion, pcts_poblacion)
13:        padre_2 = poblacion[elegido]
14:        hijosBLX = cruceBLX(padre_1, padre_2)
15:        hijos[0] = hijosBLX.first
16:        hijos[1] = hijosBLX.second
17:        if generacion % mutar == 0 then                ▷ Proceso de mutacion
18:            cromosoma_mutar = random ∈ [0, 2)
19:            gen_mutar = random ∈ [0, num_caracteristicas)
20:            mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
21:        end if
22:        evalPoblacionEstacionario(train, clases_train, hijos, pcts_hijos, num_eval)
23:        if pcts_hijos[0] >= pcts_hijos[1] then
24:            mejor = pcts_hijos[0]
25:            i_mejor = 0
26:            peor = pcts_hijos[1]
27:            i_peor = 1
```

```
        else
            mejor = pcts_hijos[1]
            i_mejor = 1
            peor = pcts_hijos[0]
            i_peor = 0
        end if
        if mejor > pcts_poblacion[index_peores[1]] then
            poblacion[index_peores[1]] = hijos[i_mejor]
            pcts_poblacion[index_peores[1]] = mejor
            if peor > pcts_poblacion[index_peores[0]] then
                poblacion[index_peores[0]] = hijos[i_peor]
                pcts_poblacion[index_peores[0]] = peor
            else if mejor > index_peores[0] then
                poblacion[index_peores[0]] = hijos[i_mejor]
                pcts_poblacion[index_peores[0]] = mejor
            end if
        end if
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        generacion += 1
    end while
    return poblacion[index_peores[index_peores.size() - 1]]
end function
```

```
function evalPoblacionEstacionario(train, clases_train, poblacion,
pcts_poblacion, num_eval)
    for i = 0, i < poblacion.size(), i += 1 do
        KNN(train, clases_train, train, clasificacion, poblacion[i])
        num_eval += 1
        pcts_poblacion[i] = tasaAcierto(clases_train, clasificacion)
    end for
end function
```

3.4.2. Algoritmo genético estacionario con cruce aritmético.

```
function GeneticoEstacionarioCA(train, clases_train)
    num_caracteristicas = train[0].size()
    index_peores = {1, 2, 3, ..., num_caracteristicas}
    poblacion = poblacionInicial(num_caracteristicas)
    evalPoblacionEstacionario(train, clases_train, poblacion, pcts_poblacion,
num_eval)
    genes_generacion = 2 * num_caracteristicas
    mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
    sort(index_peores segun pcts_poblacion)    ▷ Mantenemos ordenada la poblacion
```

```
    while num_eval < 15000 do
        elegido1 = torneoBinario(poblacion, pcts_poblacion)
        padre_1 = poblacion[elegido]
        elegido2 = torneoBinario(poblacion, pcts_poblacion)
        padre_2 = poblacion[elegido]
        hijos[0] = cruceAtirmetico(padre_1, padre_2)
        elegido1 = torneoBinario(poblacion, pcts_poblacion)
        padre_1 = poblacion[elegido]
        elegido2 = torneoBinario(poblacion, pcts_poblacion)
        padre_2 = poblacion[elegido]
        hijos[1] = cruceAtirmetico(padre_1, padre_2)
        if generacion % mutar == 0 then                    ▷ Proceso de mutacion
            cromosoma_mutar = random ∈ [0, 2)
            gen_mutar = random ∈ [0, num_caracteristicas)
            mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
        end if
        evalPoblacionEstacionario(train, clases_train, hijos, pcts_hijos, num_eval)
        if pcts_hijos[0] >= pcts_hijos[1] then
            mejor = pcts_hijos[0]
            i_mejor = 0
            peor = pcts_hijos[1]
            i_peor = 1
        else
            mejor = pcts_hijos[1]
            i_mejor = 1
            peor = pcts_hijos[0]
            i_peor = 0
        end if
        if mejor > pcts_poblacion[index_peores[1]] then
            poblacion[index_peores[1]] = hijos[i_mejor]
            pcts_poblacion[index_peores[1]] = mejor
            if peor > pcts_poblacion[index_peores[0]] then
                poblacion[index_peores[0]] = hijos[i_peor]
                pcts_poblacion[index_peores[0]] = peor
            else if mejor > index_peores[0] then
                poblacion[index_peores[0]] = hijos[i_mejor]
                pcts_poblacion[index_peores[0]] = mejor
            end if
        end if
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        generacion += 1
    end while
    return poblacion[index_peores[index_peores.size() - 1]]
end function
```

Al igual que pasaba en los algoritmos genéticos generacionales es necesario obtener el doble de padres para producir el mismo número de hijos, ya que el cruce aritmético sólo genera un hijo.

3.5. Algoritmos meméticos.

Las versiones de algoritmos meméticos implementadas a continuación están basadas en la implementación del algoritmos genético generacional con cruce BLX-α y por lo tanto cumplen las mismas indicaciones que se dieron para dicho algoritmo en páginas anteriores excepto que en este caso el tamaño de la población se reduce a 10 individuos.

3.5.1. Algoritmo memético con búsqueda local sobre toda la población cada 10 generaciones.

```
1: function MemeticoGeneracionalBLX_10_1(train, clases_train)
2:     num_caracteristicas = train[0].size()
3:     index_mejores = {1, 2, 3, ..., num_caracteristicas}
4:     index_peores = {1, 2, 3, ..., num_caracteristicas}
5:     poblacion = poblacionInicialMemeticos(num_caracteristicas)
6:     evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval, num_eval)
7:     genes_generacion = 10 * num_caracteristicas
8:     mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
9:     sort(index_mejores segun pcts_poblacion)  ▷ Mantenemos ordenada la poblacion
10:    while num_eval < 15000 do
11:        for i = 0, i < 10, i += 1 do
12:            elegido = torneoBinario(poblacion, pcts_poblacion)
13:            padres[i] = poblacion[elegido]
14:            pcts_padres[i] = pcts_poblacion[elegido]
15:        end for
16:        num_cruces = 0.7*10                                  ▷ Proceso de cruce
17:        for i = 0, i < num_cruces, i+ = 2 do
18:            hijosBLX = cruceBLX(padres[i], padres[i + 1])
19:            hijos[i] = hijosBLX.first
20:            hijos[i + 1] = hijosBLX.second
21:            flag_eval[i] = false
22:            flag_eval[i + 1] = false
23:            pcts_hijos[i] = 0
24:            pcts_hijos[i + 1] = 0
25:        end for
26:        for i = num_cruces, i < 10, i+ = 1 do
27:            hijos[i] = padres[i]
28:            pcts_hijos[i] = pcts_padres[i]
29:        end for
```

```
        if generacion % mutar == 0 then                          ▷ Proceso de mutacion
            cromosoma_mutar = random ∈ [0, 10)
            gen_mutar = random ∈ [0, num_caracteristicas)
            mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
            flag_eval[cromosoma_mutar] = false
        end if
        if generacion % 10 == 0 then                    ▷ Aplicamos la búsqueda local
            for i = 0, i < hijos.size(), i+=1 do
                localSearchMemeticos(train, clases_train, hijos[i], num_eval)
                flag_eval[i] = false
            end for
        end if
        evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        mejor = poblacion[index_mejores[0]]          ▷ Guardamos el mejor (elitismo)
        pct_mejor = pcts_poblacion[index_mejores[0]]
        poblacion = hijos
        pcts_poblacion = pcts_hijos
        if pct_mejor > pcts_poblacion[index_peores[index_peores.size() - 1]] then
            poblacion[index_peores[0]] = mejor                              ▷ Elitismo
            pcts_poblacion[index_peores[0]] = pct_mejor
        end if
        sort(index_mejores segun pcts_poblacion)
        generacion += 1
    end while
    return poblacion[index_mejores[0]]
end function
```

En este caso se aplica la búsqueda local cada 10 generaciones a todos los individuos de la población.

```
function POBLACIONINICIALMEMETICOS(num_caracteristicas)
    for i = 0, i < 10, i += 1 do
        poblacion_inicial = solInicialAleatoria(num_caracteristicas)
    end for
    return poblacion_inicial
end function
```

Genera una población inicial de 10 individuos, cada uno de tamaño *num_caracteristicas*.

```
1: function localSearchMemeticos(train, clases_train, pesos, num_eval)
2:     indices = {1, 2, 3, ..., num_caracteristicas}
3:     pesos = solInicialAleatoria(num_caracteristicas)
4:     KNN(train, clases_train, train, clasificacion, pesos)
5:     porcentaje_ant = tasaAcierto(clases_train, clasificacion)
6:     i_aux = 0
7:     while num_vecinos < 2 * num_caracteristicas do
8:         mejora = false
9:         for i = 0, i < num_caracteristicas && !mejora, i += 1 do
10:            sol_nueva = pesos
11:            elegido = random ∈ [i_auxiliar, indices.size() - 1]
12:            posicion_auxiliar = indices[i_auxiliar]
13:            indices[i_auxiliar] = indices[elegido]
14:            indices[elegido] = posicion_auxiliar
15:            modificarPeso(sol_nueva, indices[i_auxiliar])
16:            num_vecinos += 1
17:            i_auxiliar += 1
18:            if i_auxiliar == num_caracteristicas then
19:                i_auxiliar = 0          ▷ Se han cambiado todas, empezamos de cero
20:            end if
21:            KNN(train, clases_train, train, clasificacion, sol_nueva)
22:            porcentaje_nuevo = tasaAcierto(clases_train, clasificacion)
23:            num_eval += 1
24:            if porcentaje_nuevo > porcentaje_ant then
25:                pesos = sol_nueva
26:                porcentaje_ant = porcentaje_nuevo
27:                mejora = true
28:            end if
29:        end for
30:    end while
31:    return pesos
32: end function
```

Es la misma búsqueda local que la empleada como algoritmo independiente sólo que la condición de parada se reduce a que se exploren *2*n* vecinos siendo n le número de características, y esto de forma independiente a que haya mejora o no.

3.5.2. Algoritmo memético con búsqueda local sobre un individuo aleatorio de la población cada 10 generaciones.

```
1: function MemeticoGeneracionalBLX_10_0_1(train, clases_train)
2:     num_caracteristicas = train[0].size()
3:     index_mejores = {1, 2, 3, ..., num_caracteristicas}
4:     index_peores = {1, 2, 3, ..., num_caracteristicas}
5:     poblacion = poblacionInicialMemeticos(num_caracteristicas)
6:     evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval, num_eval)
7:     genes_generacion = 10 * num_caracteristicas
8:     mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
9:     sort(index_mejores segun pcts_poblacion)  ▷ Mantenemos ordenada la poblacion
10:    while num_eval < 15000 do
11:        for i = 0, i < 10, i += 1 do
12:            elegido = torneoBinario(poblacion, pcts_poblacion)
13:            padres[i] = poblacion[elegido]
14:            pcts_padres[i] = pcts_poblacion[elegido]
15:        end for
16:        num_cruces = 0.7*10                                  ▷ Proceso de cruce
17:        for i = 0, i < num_cruces, i+=2 do
18:            hijosBLX = cruceBLX(padres[i], padres[i + 1])
19:            hijos[i] = hijosBLX.first
20:            hijos[i + 1] = hijosBLX.second
21:            flag_eval[i] = false
22:            flag_eval[i + 1] = false
23:            pcts_hijos[i] = 0
24:            pcts_hijos[i + 1] = 0
25:        end for
26:        for i = num_cruces, i < 10, i += 1 do
27:            hijos[i] = padres[i]
28:            pcts_hijos[i] = pcts_padres[i]
29:        end for
30:        if generacion % mutar == 0 then                      ▷ Proceso de mutacion
31:            cromosoma_mutar = random ∈ [0, 10)
32:            gen_mutar = random ∈ [0, num_caracteristicas)
33:            mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
34:            flag_eval[cromosoma_mutar] = false
35:        end if
36:        if generacion % 10 == 0 then                 ▷ Aplicamos la búsqueda local
37:            elegidoLS = random ∈ [0, 10)
38:            localSearchMemeticos(train, clases_train, hijos[elegidoLS], num_eval)
39:            flag_eval[elegidoLS] = false
40:        end if
```

```
        evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        mejor = poblacion[index_mejores[0]]          ▷ Guardamos el mejor (elitismo)
        pct_mejor = pcts_poblacion[index_mejores[0]]
        poblacion = hijos
        pcts_poblacion = pcts_hijos
        if pct_mejor > pcts_poblacion[index_peores[index_peores.size() - 1]] then
            poblacion[index_peores[0]] = mejor                              ▷ Elitismo
            pcts_poblacion[index_peores[0]] = pct_mejor
        end if
        sort(index_mejores segun pcts_poblacion)
        generacion += 1
    end while
    return poblacion[index_mejores[0]]
end function
```

En este caso se aplica la búsqueda local cada 10 generaciones a un individuo de la población elegido de forma aleatoria.

3.5.3. Algoritmo memético con búsqueda local sobre el mejor individuo aleatorio de la población cada 10 generaciones.

```
function MemeticoGeneracionalBLX_10_0_1_mej(train, clases_train)
    num_caracteristicas = train[0].size()
    index_mejores = {1, 2, 3, ..., num_caracteristicas}
    index_peores = {1, 2, 3, ..., num_caracteristicas}
    poblacion = poblacionInicialMemeticos(num_caracteristicas)
    evalPoblacion(train, clases_train, poblacion, pcts_poblacion, flag_eval, num_eval)
    genes_generacion = 10 * num_caracteristicas
    mutar = 1000.0 / genes_generacion          ▷ Mutamos cada mutar generaciones
    sort(index_mejores segun pcts_poblacion)  ▷ Mantenemos ordenada la poblacion
    while num_eval < 15000 do
        for i = 0, i < 10, i += 1 do
            elegido = torneoBinario(poblacion, pcts_poblacion)
            padres[i] = poblacion[elegido]
            pcts_padres[i] = pcts_poblacion[elegido]
        end for
        num_cruces = 0.7*10                                    ▷ Proceso de cruce
        for i = 0, i < num_cruces, i+=2 do
            hijosBLX = cruceBLX(padres[i], padres[i + 1])
            hijos[i] = hijosBLX.first
            hijos[i + 1] = hijosBLX.second
            flag_eval[i] = false
            flag_eval[i + 1] = false
```

```
            pcts_hijos[i] = 0
            pcts_hijos[i + 1] = 0
        end for
        for i = num_cruces, i < 10, i += 1 do
            hijos[i] = padres[i]
            pcts_hijos[i] = pcts_padres[i]
        end for
        if generacion % mutar == 0 then                    ▷ Proceso de mutacion
            cromosoma_mutar = random ∈ [0, 10)
            gen_mutar = random ∈ [0, num_caracteristicas)
            mutarCromosoma(hijos[cromosoma_mutar], gen_mutar)
            flag_eval[cromosoma_mutar] = false
        end if
        evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        if generacion % 10 == 0 then                    ▷ Aplicamos la búsqueda local
            mejor = index_peores[index_peores.size() - 1]
            localSearchMemeticos(train, clases_train, hijos[mejor], num_eval)
            flag_eval[index_peores[index_peores.size() - 1]] = false
        end if
        evalPoblacion(train, clases_train, hijos, pcts_hijos, flag_eval, num_eval)
        sort(index_peores segun pcts_hijos)      ▷ Mantenemos ordenada la poblacion
        mejor = poblacion[index_mejores[0]]          ▷ Guardamos el mejor (elitismo)
        pct_mejor = pcts_poblacion[index_mejores[0]]
        poblacion = hijos
        pcts_poblacion = pcts_hijos
        if pct_mejor > pcts_poblacion[index_peores[index_peores.size() - 1]] then
            poblacion[index_peores[0]] = mejor                              ▷ Elitismo
            pcts_poblacion[index_peores[0]] = pct_mejor
        end if
        sort(index_mejores segun pcts_poblacion)
        generacion += 1
    end while
    return poblacion[index_mejores[0]]
end function
```

En este caso se aplica la búsqueda local cada 10 generaciones al mejor individuo de la población.

4. Descripción en pseudocódigo del algoritmo de comparación.

```
1: pesos = algoritmo a comparar(train, clases_train)
2: KNN(train, clases_train, test, clasificacion, pesos)
3: porcentaje = tasaAcierto(clases_test, clasificacion)
```

La idea es sencilla, obtenemos los pesos usando un cierto algoritmo de los descritos anteriormente, llamamos al clasificador *1-NN* con los datos de *train* como conjunto de entrenamiento y *test* como conjunto de prueba y obtenemos la clasificación para los datos de *test*, *clasificación*. Finalmente, calculamos la tasa de acierto comparando la clasificación obtenida por nuestros algoritmos con la clasificación real, *clases_test* para obtener un porcentaje de acierto.

5. Proceso de desarrollo y manual de usuario.

Todo el código ha sido desarrollado por mí, usando como apoyo los seminarios y transparencias de teoría. Como páginas de consulta de Internet principalmente he usado cplusplus, sobre todo para la funciones de librerías externas, como puede ser para la generación de datos siguiendo una distribución normal. El proceso de desarrollo ha sido el lógico, al menos para mí. Primero he empezado por desarrollar el esqueleto de la práctica, es decir, crear la estructura de archivos, desarrollar el código de preparación de los conjuntos de datos y la lectura de los mismos. Toda la práctica se ha desarrollado en C++ excepto el proceso de manipulación, normalización y partición de datos, el cual se ha hecho en R. Una vez hecho todo el proceso preliminar, desarrollé el clasificador *1-NN* y las funciones de comparación de resultados, ya que el resto de algoritmos lo necesitan. Finalmente, desarrollé la búsqueda local, los algoritmos genéticos y los algoritmos meméticos.

Para poder reproducir una ejecución de la práctica se ha proporcionado un *makefile*, el cual realiza tanto la creación de carpetas necesarias como la compilación. Una vez compilado, se ejecuta mediante el comando *./bin/main* (suponiendo que se use un entorno UNIX). Este ejecutable cambia su funcionamiento según el valor de la variable booleana *all* dentro del fichero *main.cpp*. Si dicha variable está en *true*, se ejecutaran todos los algoritmos para todas las particiones de todos los conjuntos de datos. Si dicha variable está en *false*, se preguntará por pantalla al usuario que conjunto de datos quiere usar y posteriormente se ejecutarán todos los algoritmos para todas las particiones del conjunto de datos seleccionado.

Los conjuntos de datos leídos por mis algoritmos se han aportado dentro de la carpeta *./Data/csv*. Igualmente se ha entregado el fichero *GenerarDatos.R*, el cual genera los conjuntos de datos.

6. Experimentos y análisis de resultados.

Los conjuntos de datos usados han sido los siguientes:

- **Sonar:** Conjunto de datos de detección de materiales mediante señales de sonar, discriminando entre objetos metálicos y rocas. 208 ejemplos con 60 características que deben ser clasificados en 2 clases.
- **Spambase**: Conjunto de datos de detección de SPAM frente a correo electrónico seguro. 460 ejemplos con 57 características que deben ser clasificados en 2 clases.
- **Wdbc (Wisconsin Database Breast Cancer):** Esta base de datos contiene 30 características calculadas a partir de una imagen digitalizada de una aspiración con aguja fina (FNA) de una masa en la mama. Se describen las características de los núcleos de las células presentes en la imagen. La tarea consiste en determinar si un tumor encontrado es benigno o maligno (M = maligno, B = benigna). 569 ejemplos con 30 características que deben ser clasificados en 2 clases.

Se debe tener en cuenta que para que la ejecución de nuestros algoritmos sea uniforme independientemente del conjunto de datos usado, estos mismos han sido modificados para tratarlos de manera uniforme. Las etiquetas han sido cambiadas por *-1* y *+1* y en todos los conjuntos se han puesto al final de cada ejemplo.

Varios algoritmos dependen de la elección de ciertos parámetros para su ejecución. El primero de ellos es la semilla para le generación de números aleatorios. En mi caso he elegido como semilla mi DNI, *75573052*. Para la generación de vecinos y para la mutación se van a usar los datos generados por una distribución normal de media *0* y varianza σ^2 donde $\sigma = 0{,}3$. Por lo general, como criterio de parada de los algoritmos, se va a usar el número de evaluaciones que se han realizado, en nuestro caso dicho valor será *15000* evaluaciones, aunque más delante veremos que pasa con este valor cuando realicemos el estudio de la convergencia. En el caso del cruce *BLX-*α se va a usar $\alpha = 0{,}3$.

Para los algoritmos genéticos se va a usar una población de *30* cromosomas, una probabilidad de cruce de *0.7* para el caso de los generacionales y de *1* para los estacionarios. La probabilidad de mutación es de *0.001* y la condición de parada consiste en realizar *15000* evaluaciones de la función objetivo.

Para los algoritmos meméticos se va a usar una población de *10* cromosomas, una probabilidad de cruce de *0.7*. La probabilidad de mutación es de *0.001* y la condición de parada consiste en realizar *15000* evaluaciones de la función objetivo. Cuando se realiza una búsqueda local dentro de un algoritmo memético esta se detendrás tras evaluar *2*n* vecinos, siendo n el tamaño del cromosoma.

Una vez aclarado todo, veamos los resultados. Primero voy a mostrar un conjunto de tablas en las que se recogen los resultados obtenidos para todas las particiones de datos en los tres conjuntos de prueba. Veamos dichos resultados:

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	90.3846	0	83.913	0.003	95.7895	0.002
1.Test-Train	79.8077	0.001	84.7826	0.003	92.2535	0.003
2.Train-Test	81.7308	0	86.5217	0.003	94.0351	0.002
2.Test-Train	81.7308	0.001	83.913	0.003	95.4225	0.003
3.Train-Test	78.8462	0	83.913	0.003	92.9825	0.002
3.Test-Train	82.6923	0.001	80.4348	0.003	95.4225	0.004
4.Train-Test	77.8846	0	80.8696	0.003	95.7895	0.002
4.Test-Train	79.8077	0.001	86.9565	0.003	95.4225	0.003
5.Train-Test	88.4615	0	84.3478	0.003	94.386	0.002
5.Test-Train	85.5769	0.001	80	0.003	94.3662	0.003
Media	**82.69231**	**0.0005**	**83.5652**	**0.003**	**94.58698**	**0.0026**

Tabla 6.1: 1-NN

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	84.6154	0	86.087	0.003	97.193	0.002
1.Test-Train	83.6538	0	86.5217	0.003	95.4225	0.003
2.Train-Test	83.6538	0	85.2174	0.003	94.386	0.002
2.Test-Train	78.8462	0	80.4348	0.003	94.0141	0.003
3.Train-Test	76.9231	0	84.3478	0.003	92.9825	0.002
3.Test-Train	85.5769	0	86.5217	0.003	96.1268	0.003
4.Train-Test	82.6923	0	84.7826	0.003	96.8421	0.002
4.Test-Train	82.6923	0	83.4783	0.003	94.0141	0.003
5.Train-Test	91.3462	0	86.087	0.003	93.6842	0.002
5.Test-Train	83.6538	0	86.5217	0.003	94.3662	0.003
Media	**83.36538**	**0**	**85**	**0.003**	**94.90315**	**0.0025**

Tabla 6.2: Relief

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	86.5385	0.927	79.5652	5.562	93.6842	2.043
1.Test-Train	77.8846	1.099	76.9565	6.193	92.2535	2.103
2.Train-Test	80.7692	1.025	86.087	9.32	93.6842	1.502
2.Test-Train	78.8462	1.439	80.8696	9.754	95.0704	2.167
3.Train-Test	79.8077	1.152	86.9565	9.034	91.5789	1.665
3.Test-Train	85.5769	0.899	81.7391	9.925	94.3662	2.084
4.Train-Test	80.7692	1.582	80.4348	5.104	96.1404	1.568
4.Test-Train	83.6538	0.972	86.087	8.173	94.3662	1.611
5.Train-Test	82.6923	1.308	83.0435	8.138	92.9825	3.539
5.Test-Train	86.5385	1.147	81.7391	7.084	95.4225	1.629
Media	**82.30769**	**1.155**	**82.34783**	**7.8287**	**93.9549**	**1.9911**

Tabla 6.3: LocalSearch

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	90.3846	10.627	83.4783	52.646	94.7368	36.66
1.Test-Train	80.7692	10.696	83.0435	58.855	92.9577	37.166
2.Train-Test	85.5769	10.636	85.2174	54.008	94.386	36.692
2.Test-Train	78.8462	10.645	85.2174	58.227	96.1268	37.894
3.Train-Test	81.7308	10.671	86.5217	58.129	94.0351	36.747
3.Test-Train	82.6923	10.607	86.087	54.675	96.1268	38.681
4.Train-Test	78.8462	10.736	81.3043	52.806	95.4386	36.684
4.Test-Train	85.5769	10.754	85.2174	54.693	95.0704	37.099
5.Train-Test	86.5385	11.68	86.9565	54.159	93.6842	36.632
5.Test-Train	90.3846	13.754	83.4783	59.589	94.7183	37.163
Media	**84.13462**	**11.0806**	**84.65218**	**55.7787**	**94.72807**	**37.1418**

Tabla 6.4: GeneticoGeneracionalBLX

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	87.5	12.259	82.6087	52.806	97.5439	36.043
1.Test-Train	83.6538	13.107	83.4783	50.405	91.5493	36.64
2.Train-Test	82.6923	11.482	86.5217	52.04	95.0877	36.121
2.Test-Train	84.6154	11.738	84.3478	50.148	95.4225	37.352
3.Train-Test	83.6538	10.68	83.4783	51.247	93.6842	37.906
3.Test-Train	82.6923	11.044	83.0435	54.377	95.7746	37.111
4.Train-Test	80.7692	11.98	81.7391	50.571	97.5439	37.009
4.Test-Train	77.8846	12.757	88.6957	50.131	95.4225	37.702
5.Train-Test	88.4615	11.939	82.1739	51.582	95.0877	36.373
5.Test-Train	88.4615	11.651	83.4783	55.727	95.0704	37.608
Media	**84.03844**	**11.8637**	**83.95653**	**51.9034**	**95.21867**	**36.9865**

Tabla 6.5: GeneticoGeneracionalCA

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	90.3846	11.972	81.7391	50.279	94.7368	36.24
1.Test-Train	79.8077	11.744	84.7826	50.883	94.0141	36.537
2.Train-Test	81.7308	11.86	87.8261	49.83	93.3333	36.312
2.Test-Train	81.7308	12.038	81.3043	49.396	93.662	36.444
3.Train-Test	78.8462	12.321	84.7826	50.735	93.3333	36.129
3.Test-Train	84.6154	12.872	84.3478	49.889	95.7746	36.43
4.Train-Test	77.8846	12.172	84.3478	53.725	95.0877	36.269
4.Test-Train	81.7308	11.77	84.7826	52.293	95.4225	36.411
5.Train-Test	87.5	11.121	85.2174	53.198	94.7368	36.282
5.Test-Train	86.5385	11.273	82.6087	50.393	95.0704	36.581
Media	**83.07694**	**11.9143**	**84.1739**	**51.0621**	**94.51715**	**36.3635**

Tabla 6.6: GeneticoEstacionarioBLX

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	89.4231	11.878	81.3043	47.787	95.7895	36.289
1.Test-Train	77.8846	11.086	83.0435	47.749	92.6056	36.501
2.Train-Test	79.8077	10.799	86.9565	47.848	94.7368	36.308
2.Test-Train	84.6154	10.733	86.5217	47.709	95.0704	36.383
3.Train-Test	76.9231	10.748	83.4783	47.682	94.0351	36.323
3.Test-Train	86.5385	10.747	79.1304	47.708	96.1268	36.473
4.Train-Test	76.9231	11.436	86.087	47.723	96.4912	36.293
4.Test-Train	82.6923	11.405	86.9565	47.809	94.3662	36.381
5.Train-Test	89.4231	11.873	81.3043	47.798	94.7368	36.295
5.Test-Train	89.4231	13.22	79.5652	47.703	94.7183	36.427
Media	**83.3654**	**11.3925**	**83.43477**	**47.7516**	**94.86767**	**36.3673**

Tabla 6.7: GeneticoEstacionarioCA

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1.Train-Test	87.5	12.176	78.2609	49.232	93.6842	37.004
1.Test-Train	78.8462	11.283	83.913	49.404	91.1972	37.681
2.Train-Test	77.8846	11.416	87.3913	49.531	95.0877	36.721
2.Test-Train	76.9231	14.118	83.913	48.122	95.7746	37.561
3.Train-Test	80.7692	12.355	83.0435	49.228	92.2807	36.935
3.Test-Train	84.6154	11.895	81.3043	49.567	95.7746	37.294
4.Train-Test	77.8846	11.959	84.3478	48.986	96.1404	36.783
4.Test-Train	76.9231	12.103	86.5217	48.211	94.0141	38.419
5.Train-Test	90.3846	12.942	83.913	49.31	94.386	36.484
5.Test-Train	87.5	12.43	83.4783	48.925	95.0704	37.563
Media	**81.92308**	**12.2677**	**83.60868**	**49.0516**	**94.34099**	**37.2445**

Tabla 6.8: MemeticoGeneracionalBLX_10_1

	sonar		spambase		wdbc	
	%_clas	T	%_clas	T	%_clas	T
1.Train-Test	88.4615	10.968	83.4783	48.067	95.4386	36.473
1.Test-Train	80.7692	11.054	84.3478	48.352	94.0141	36.878
2.Train-Test	81.7308	10.941	86.5217	48.315	95.4386	36.452
2.Test-Train	81.7308	11.588	82.1739	47.999	96.1268	36.851
3.Train-Test	77.8846	10.932	81.3043	47.977	92.9825	36.563
3.Test-Train	81.7308	11.504	87.8261	48.465	96.1268	36.898
4.Train-Test	77.8846	12.111	83.0435	47.916	95.4386	36.356
4.Test-Train	84.6154	11.119	87.3913	48.134	94.7183	36.859
5.Train-Test	87.5	10.943	83.913	48.325	94.7368	36.366
5.Test-Train	88.4615	11.544	81.7391	48.364	95.4225	36.923
Media	**83.07692**	**11.2704**	**84.1739**	**48.1914**	**95.04436**	**36.6619**

Tabla 6.9: MemeticoGeneracionalBLX_10_0_1

	sonar		spambase		wdbc	
	%_clas	T	%_clas	T	%_clas	T
1.Train-Test	89.4231	13.554	83.913	48.367	95.4386	36.477
1.Test-Train	81.7308	11.805	81.3043	47.964	90.8451	37.295
2.Train-Test	81.7308	12.354	85.6522	48.193	93.6842	37.194
2.Test-Train	82.6923	11.906	82.6087	48.172	94.3662	37.155
3.Train-Test	75.9615	13.062	83.0435	48.117	91.9298	36.362
3.Test-Train	83.6538	13.105	83.4783	48.149	96.831	36.726
4.Train-Test	77.8846	13.408	82.6087	48.02	96.1404	36.293
4.Test-Train	78.8462	12.275	88.2609	48.192	94.0141	36.71
5.Train-Test	88.4615	11.948	84.3478	48.119	94.386	36.315
5.Test-Train	83.6538	12.681	79.1304	47.948	95.7746	36.739
Media	**82.40384**	**12.6098**	**83.43478**	**48.1241**	**94.341**	**36.7266**

Tabla 6.10: MemeticoGeneracionalBLX_10_0_1_mej

	sonar		spambase		wdbc	
	%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
1-NN	82.69231	0.0005	83.5652	0.003	94.58698	0.0026
Relief	83.36538	0	85	0.003	94.90315	0.0025
LocalSearch	82.30769	1.155	82.34783	7.8287	93.9549	1.9911
AGG-BLX	84.13462	11.0806	84.65218	55.7787	94.72807	37.1418
AGG-CA	84.03844	11.8637	83.95653	51.9034	95.21867	36.9865
AGE-BLX	83.07694	11.9143	84.1739	51.0621	94.51715	36.3635
AGE-CA	83.3654	11.3925	83.43477	47.7516	94.86767	36.3673
AM-10-1	81.92308	12.2677	83.60868	49.0516	94.34099	37.2445
AM-10-0.1	83.07692	11.2704	84.1739	48.1914	95.04436	36.6619
AM-10-0.1mej	83.6538	12.681	79.1304	47.948	95.7746	36.739

Tabla 6.11: Comparativa de todos los algoritmos.

Una vez recogidos los datos, pasemos al análisis de los mismos. Lo primero que quiero comentar es que que los tiempos recogidos han sido obtenidos aplicando optimización en tiempo de compilación, en concreto el nivel *-O2* ofrecido por el compilador *g++*. Como dato curioso, la ejecución completa del programa (la cual incluye la ejecución de todos los algoritmos para todos los conjuntos de datos para todas las bases de datos, la carga de los datos cada vez que son necesitados y todo el proceso de comparación de soluciones y evaluación del vector de pesos obtenido) ha tardado *7031.95 segundos*, es decir, *una hora y 57 minutos* aproximadamente.

Analizando los resultados recogidos en la tabla 6.11 podemos ver que los mejores resultados para *sonar* y *spambase* se obtienen con el algoritmo genético generacional con cruce BLX-α y para *wdbc* es el algoritmo memético que aplica la búsqueda local cada 10 generaciones al mejor individuo de la población.

En termino medio obtenemos una tasa de acierto similar independientemente del algoritmos empleado. Esto nos lleva a plantearnos si realmente merece la pena usar algoritmos más complejos como pueden ser los genéticos o meméticos, ya que la relación entre la mejora obtenida y el tiempo empleado no parece muy prometedora. Esto se debe a dos razones, la principal de ellas es los conjuntos de datos usados. Estos conjuntos de datos son bastante pequeños por lo cual los algoritmos no evolutivos se comportan bastante bien y los algoritmos evolutivos no desarrollan completamente su potencial. La segunda razón es el *sobreaprendizaje*, tema que trataremos en la siguiente parte del análisis de los datos obtenidos.

A modo extra quiero añadir una tabla más para la comparación:

sonar		spambase		wdbc	
%_clas	**T**	**%_clas**	**T**	**%_clas**	**T**
83.163458	8.36257	83.404339	35.95215	94.793654	25.95007

Tabla 6.12: Valores medios de la tabla comparativa 6.11.

Como podemos ver en la tabla de los valores medios 6.12 el conjunto de datos más lento de clasificar es *spambase* mientras que en el que obtenemos un mejor porcentaje es en *wdbc*.

Si nos ceñimos a los datos recogidos en las tablas no hay mucho más que explicar que no se pueda ver claramente, así que pasemos a estudiar dos aspectos que he aprendido con esta práctica, el mencionado anteriormente *sobreaprendizaje* y un estudio de la *convergencia*.

6.1. Sobreaprendizaje.

El *sobreaprendizaje* es el efecto de sobreentrenar un algoritmo. Este efecto puede provocar que el algoritmo sea muy bueno para los datos de entrenamiento pero que luego con los datos de prueba no funcione también, es decir, el algoritmo se especializa en los datos de entrenamiento y pierde la generalidad necesaria para poder realizar un buen trabajo fuera del conjunto de entrenamiento.

Para ver que algoritmos sufren de este problema simplemente debemos realizar lo siguiente:

- Calculamos el vector de pesos W usando el conjunto de entrenamiento, *train*.
- Llamamos al clasificador *1-NN* pasándole como conjunto de entrenamiento y prueba el mismo, *train*. De esta manera valoramos como se comporta W para clasificar los mismos datos con los que ha aprendido.
- Calculamos el porcentaje de acierto para compararlo posteriormente.
- Llamamos de nuevo al clasificador *1-NN* pasándole como conjunto de entrenamiento *train* y como conjunto de prueba *test*. De esta manera valoramos como se comporta W para clasificar unos datos nuevos
- Calculamos el porcentaje de acierto para compararlo posteriormente.
- Comparamos los dos porcentajes.
- Repetimos el proceso pero intercambiando *train* y *test*.

Los resultados recogidos para la base de datos *sonar* los podemos ver en las tablas 6.13 y 6.14, para *spambase* en las tablas 6.15 y 6.16 y para *wdbc* en las tablas 6.17 y 6.18. En dichas tablas se recogen las tasas de clasificación siguiendo el proceso comentado anteriormente.

	LocalSearch	AGG-BLX	AGG-CA	AGE-BLX	AGE-CA
1. Train-Train	81.7308	91.3462	89.4231	93.2692	88.4615
1. Train-Test	86.5385	90.3846	87.5	90.3846	89.4231
1. Test-Test	88.4615	95.1923	91.3462	95.1923	90.3846
1. Test-Train	77.8846	80.7692	83.6538	79.8077	77.8846
2.Train-Train	86.5385	99.0385	93.2692	97.1154	95.1923
2.Train-Test	80.7692	85.5769	82.6923	81.7308	79.8077
2.Test-Test	90.3846	92.3077	90.3846	93.2692	92.3077
2.Test-Train	78.8462	78.8462	84.6154	81.7308	84.6154
3.Train-Train	86.5385	94.2308	92.3077	94.2308	91.3462
3.Train-Test	79.8077	81.7308	83.6538	78.8462	76.9231
3.Test-Test	92.3077	96.1538	92.3077	95.1923	91.3462
3.Test-Train	85.5769	82.6923	82.6923	84.6154	86.5385
4.Train-Train	90.3846	97.1154	94.2308	95.1923	96.1538
4.Train-Test	80.7692	78.8462	80.7692	77.8846	76.9231
4.Test-Test	87.5	94.2308	91.3462	96.1538	90.3846
4.Test-Train	83.6538	85.5769	77.8846	81.7308	82.6923
5.Train-Train	81.7308	89.4231	88.4615	91.3462	88.4615
5.Train-Test	82.6923	86.5385	88.4615	87.5	89.4231
5.Test-Test	87.5	91.3462	88.4615	90.3846	88.4615
5.Test-Train	86.5385	90.3846	88.4615	86.5385	89.4231

Tabla 6.13: Sobreaprendizaje en *sonar* (1)

	AM-10-1	AM-10-0.1	AM-10-0.1mej
1. Train-Train	90.3846	95.1923	88.4615
1. Train-Test	87.5	88.4615	89.4231
1. Test-Test	92.3077	93.2692	92.3077
1. Test-Train	78.8462	80.7692	81.7308
2.Train-Train	95.1923	96.1538	95.1923
2.Train-Test	77.8846	81.7308	81.7308
2.Test-Test	93.2692	92.3077	91.3462
2.Test-Train	76.9231	81.7308	82.6923
3.Train-Train	92.3077	95.1923	91.3462
3.Train-Test	80.7692	77.8846	75.9615
3.Test-Test	95.1923	93.2692	93.2692
3.Test-Train	84.6154	81.7308	83.6538
4.Train-Train	95.1923	97.1154	95.1923
4.Train-Test	77.8846	77.8846	77.8846
4.Test-Test	92.3077	90.3846	93.2692
4.Test-Train	76.9231	84.6154	78.8462
5.Train-Train	85.5769	88.4615	92.3077
5.Train-Test	90.3846	87.5	88.4615
5.Test-Test	84.6154	88.4615	91.3462
5.Test-Train	87.5	88.4615	83.6538

Tabla 6.14: Sobreaprendizaje en *sonar* (2)

	LocalSearch	AGG-BLX	AGG-CA	AGE-BLX	AGE-CA
1. Train-Train	83.0435	92.1739	88.2609	89.5652	89.1304
1. Train-Test	79.5652	83.4783	82.6087	81.7391	81.3043
1. Test-Test	88.2609	94.7826	89.1304	93.913	91.7391
1. Test-Train	76.9565	83.0435	83.4783	84.7826	83.0435
2.Train-Train	92.1739	93.913	90	95.6522	92.1739
2.Train-Test	86.087	85.2174	86.5217	87.8261	86.9565
2.Test-Test	86.087	92.1739	87.8261	89.1304	86.9565
2.Test-Train	80.8696	85.2174	84.3478	81.3043	86.5217
3.Train-Train	88.2609	91.3043	88.6957	87.8261	89.5652
3.Train-Test	86.9565	86.5217	83.4783	84.7826	83.4783
3.Test-Test	92.1739	95.2174	91.7391	95.2174	93.0435
3.Test-Train	81.7391	86.087	83.0435	84.3478	79.1304
4.Train-Train	89.1304	91.3043	88.6957	93.4783	91.7391
4.Train-Test	80.4348	81.3043	81.7391	84.3478	86.087
4.Test-Test	88.2609	92.6087	86.087	90.4348	89.1304
4.Test-Train	86.087	85.2174	88.6957	84.7826	86.9565
5.Train-Train	89.1304	93.4783	90	93.913	90.8696
5.Train-Test	83.0435	86.9565	82.1739	85.2174	81.3043
5.Test-Test	89.5652	92.6087	87.8261	93.4783	91.3043
5.Test-Train	81.7391	83.4783	83.4783	82.6087	79.5652

Tabla 6.15: Sobreaprendizaje en *spambase* (1)

	AM-10-1	**AM-10-0.1**	**AM-10-0.1mej**
1. Train-Train	90.8696	88.6957	90.4348
1. Train-Test	78.2609	83.4783	83.913
1. Test-Test	96.5217	93.913	92.6087
1. Test-Train	83.913	84.3478	81.3043
2.Train-Train	96.087	93.913	92.6087
2.Train-Test	87.3913	86.5217	85.6522
2.Test-Test	87.8261	90.4348	89.5652
2.Test-Train	83.913	82.1739	82.6087
3.Train-Train	89.1304	88.6957	90.4348
3.Train-Test	83.0435	81.3043	83.0435
3.Test-Test	96.5217	93.4783	96.5217
3.Test-Train	81.3043	87.8261	83.4783
4.Train-Train	92.6087	88.6957	93.0435
4.Train-Test	84.3478	83.0435	82.6087
4.Test-Test	88.2609	88.6957	90.4348
4.Test-Train	86.5217	87.3913	88.2609
5.Train-Train	93.4783	92.1739	92.6087
5.Train-Test	83.913	83.913	84.3478
5.Test-Test	93.4783	91.3043	92.6087
5.Test-Train	83.4783	81.7391	79.1304

Tabla 6.16: Sobreaprendizaje en *spambase* (2)

	LocalSearch	**AGG-BLX**	**AGG-CA**	**AGE-BLX**	**AGE-CA**
1. Train-Train	93.662	96.831	96.831	97.1831	96.1268
1. Train-Test	93.6842	94.7368	97.5439	94.7368	95.7895
1. Test-Test	99.2982	100	99.6491	99.6491	99.6491
1. Test-Train	92.2535	92.9577	91.5493	94.0141	92.6056
2.Train-Train	96.4789	98.2394	97.5352	98.9437	97.5352
2.Train-Test	93.6842	94.386	95.0877	93.3333	94.7368
2.Test-Test	95.4386	98.2456	96.8421	97.8947	97.193
2.Test-Train	95.0704	96.1268	95.4225	93.662	95.0704
3.Train-Train	96.1268	97.1831	96.831	97.5352	96.831
3.Train-Test	91.5789	94.0351	93.6842	93.3333	94.0351
3.Test-Test	94.7368	98.2456	97.5439	98.2456	97.8947
3.Test-Train	94.3662	96.1268	95.7746	95.7746	96.1268
4.Train-Train	95.0704	97.8873	97.5352	97.5352	96.4789
4.Train-Test	96.1404	95.4386	97.5439	95.0877	96.4912
4.Test-Test	97.8947	98.5965	98.2456	98.2456	98.2456
4.Test-Train	94.3662	95.0704	95.4225	95.4225	94.3662
5.Train-Train	96.831	98.2394	97.1831	97.8873	97.1831
5.Train-Test	92.9825	93.6842	95.0877	94.7368	94.7368
5.Test-Test	96.4912	98.2456	97.193	98.9474	97.5439
5.Test-Train	95.4225	94.7183	95.0704	95.0704	94.7183

Tabla 6.17: Sobreaprendizaje en *wdbc* (1)

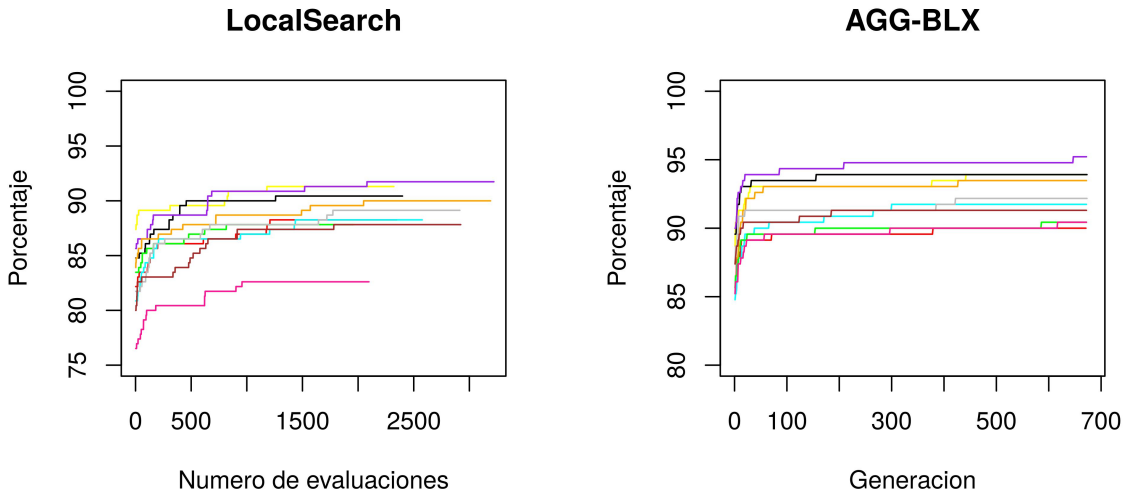
	AM-10-1	**AM-10-0.1**	**AM-10-0.1mej**
1. Train-Train	97.1831	96.831	96.4789
1. Train-Test	93.6842	95.4386	95.4386
1. Test-Test	99.2982	99.2982	100
1. Test-Train	91.1972	94.0141	90.8451
2.Train-Train	97.5352	98.2394	98.2394
2.Train-Test	95.0877	95.4386	93.6842
2.Test-Test	96.4912	97.5439	97.193
2.Test-Train	95.7746	96.1268	94.3662
3.Train-Train	97.1831	97.5352	96.4789
3.Train-Test	92.2807	92.9825	91.9298
3.Test-Test	97.193	97.5439	97.5439
3.Test-Train	95.7746	96.1268	96.831
4.Train-Train	97.1831	96.1268	97.5352
4.Train-Test	96.1404	95.4386	96.1404
4.Test-Test	98.5965	98.2456	98.2456
4.Test-Train	94.0141	94.7183	94.0141
5.Train-Train	97.5352	98.2394	97.8873
5.Train-Test	94.386	94.7368	94.386
5.Test-Test	96.4912	98.2456	97.8947
5.Test-Train	95.0704	95.4225	95.7746

Tabla 6.18: Sobreaprendizaje en *wdbc* (2)

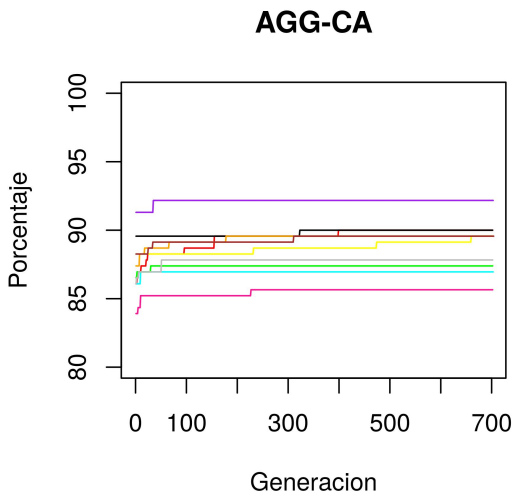
Como podemos ver las tasas de clasificación para los casos en los que se aprende y se prueba con el mismo conjunto (filas de las tablas etiquetadas como *Train-Train* o *Test-Test*) tienen un valor bastante más alto que las filas en las que el conjunto de entrenamiento es distinto al conjunto de test, llegando incluso a un 100 % de acierto para la partición 1 al hacer *test-test* con el algoritmo *AGG-BLX*. Esto se debe al *sobreaprendizaje*. Como ya he comentado anteriormente los algoritmos aprenden más de la cuenta y por eso obtienen unas tasas de acierto alta para conjuntos iguales pero cuando el conjunto de *test* cambia, el porcentaje decae, ya que el algoritmo no ha encontrado una solución lo suficientemente genérica capaz de clasificar correctamente los nuevos datos de *test*.

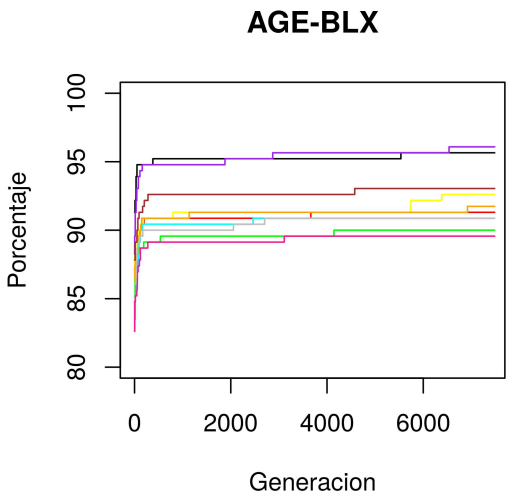
6.2. Estudio de la convergencia.

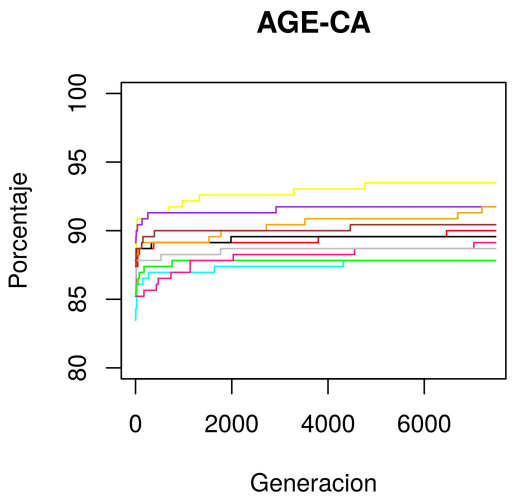
Como ya hemos comentado anteriormente el criterio de parada de nuestro algoritmos es el número de evaluaciones, en concreto, 15000 evaluaciones. Pero, ¿realmente son necesarias todas estas evaluaciones? Para comprobarlo vamos a hacer un estudio de la convergencia, el cual se basa en ver cuando los algoritmos empiezan a converger en una solución y por lo tanto no avanzan más. Para ello voy a modificar mis algoritmos añadiendo una línea de código que me escriba en un fichero el valor de la mejor solución encontrada en cada iteración y así podremos ver a partir de cual el algoritmo ha convergido y no va a mejorar la solución. Para no afectar al funcionamiento de la práctica voy a realizar dichas modificaciones en un proyecto nuevo, dentro de la carpeta APC_analisis. Dentro de dicha carpeta están todos los datos obtenidos, un fichero llamado *AnalisisDatos.R*, el cual se encarga de analizar los datos obtenidos y pintar las gráficas necesarias para este análisis [2]. Voy a realizar el estudio sobre el conjunto de datos *spambase*. Las gráficas obtenidas son las que podemos ver a continuación:

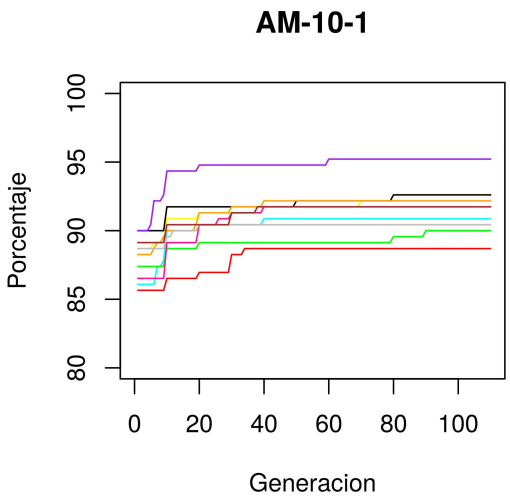


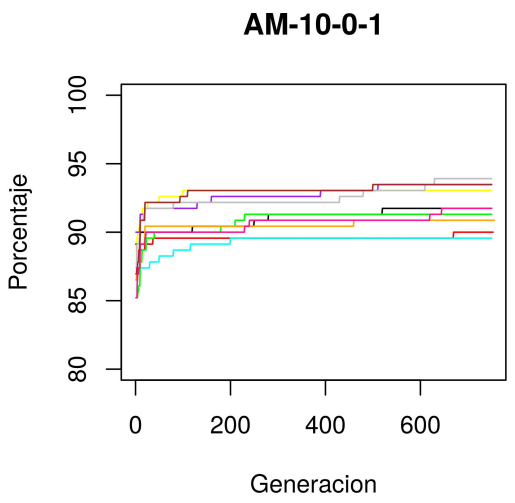
[2]Dicha carpeta no se ha entregado ya que no es requerido, en caso de querer verla, mándeme un correo y se lo envío.

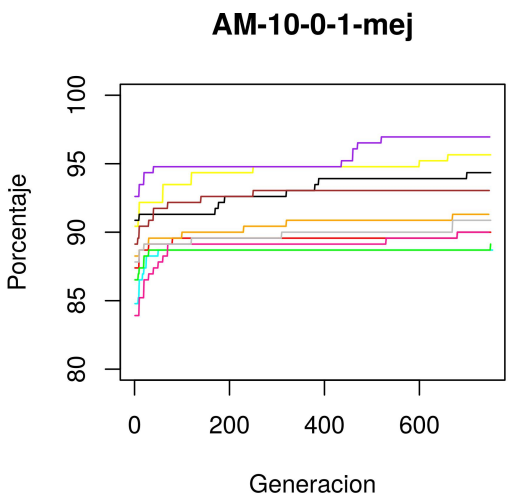

AGG-CA
Porcentaje
80
85
90
95
100
0
100
300
500
700
Generacion


AGE-BLX
Porcentaje
80
85
90
95
100
0
2000
4000
6000
Generacion


AGE-CA
Porcentaje
80
85
90
95
100
0
2000
4000
6000
Generacion


AM-10-1
Porcentaje
80
85
90
95
100
0
20
40
60
80
100
Generacion


AM-10-0-1
Porcentaje
80
85
90
95
100
0
200
400
600
Generacion


AM-10-0-1-mej
Porcentaje
80
85
90
95
100
0
200
400
600
Generacion

En cada gráfica se ha pintado el porcentaje obtenido por el mejor vector de pesos encontrado hasta el momento para cada iteración o para cada generación en el caso de los algoritmos evolutivos y para todas las particiones del conjunto de datos *spambase*. Concretamente:

- Partición 1, aprendemos con *train* → color rojo.
- Partición 1, aprendemos con *test* → color amarillo.
- Partición 2, aprendemos con *train* → color negro.
- Partición 2, aprendemos con *test* → color azul cyan.
- Partición 3, aprendemos con *train* → color verde.
- Partición 3, aprendemos con *test* → color morado.
- Partición 4, aprendemos con *train* → color naranja.
- Partición 4, aprendemos con *test* → color rosa fucsia.
- Partición 5, aprendemos con *train* → color gris.
- Partición 5, aprendemos con *test* → color marrón.

Lo primero que podemos observar es que todas de ellas tienen forma similar, esto se debe a la manera en la que se construye la solución: se parte de una solución y se modifica para ir mejorándola con el paso de las iteraciones. Pero lo interesante de estas gráficas no es esto, sino analizar la convergencia. Todos los algoritmos convergen, el problema es cuando lo hacen. Como podemos ver hay algoritmos los cuales con la mitad de evaluaciones/generaciones ya han alcanzado el máximo que van a alcanzar.

Este estudio nos lleva a pensar si realmente es necesario producir tantas generaciones cuando con la mitad de las mismas ya tenemos una solución de cierta calidad. Reducir el número de generaciones/evaluaciones se traduce en una reducción considerable del tiempo de ejecución. En conjuntos de datos pequeños como los empleados en esta práctica los tiempo de ejecución no son grandes, pero en otro tipo de conjuntos ahorrarnos unos segundos puede ser primordial.

Un último comentario para finalizar este estudio; si observamos con detenimiento la gráfica obtenida para el algoritmo *LocalSearch* podemos ver que apenas se alcanzan las 3200 evaluaciones cuando tenemos permitidas hasta 15000. Esto se debe a que la ejecución se detiene por la otra condición de parada, es decir, se generan *20*n* vecinos sin que se encuentre ninguna solución. Este fenómeno lo podemos ver con más detalle en la tabla 6.19.

	Evaluaciones	Vecinos sin mejora	%_clas
1. Train-Train	2348	1140	88.2609
1. Test-Test	2321	1140	91.3043
2. Train-Train	2399	1140	90.4348
2. Test-Test	2576	1140	88.2609
3. Train-Train	1956	1140	87.8261
3. Test-Test	3220	1140	91.7391
4. Train-Train	3190	1140	90
4. Test-Test	2097	1140	82.6087
5. Train-Train	2914	1140	89.1304
5. Test-Test	2922	1140	87.8261

Tabla 6.19: Evaluaciones vs. vecinos generados.

spambase tiene *57* características y por lo tanto *20*n = 20*57 = 1140*, es decir, si se generan 1140 vecinos y no hay mejora se detiene la búsqueda. Como podemos ver en la tabla 6.19 todas las ejecuciones se detienen por no encontrar mejora, no por agotar el número máximo de evaluaciones permitidas.

6.3. Un paso más allá.

Se podría mejorar la función objetivo para obtener unos mejores resultados. Una de las posibles mejoras sería fomentar pesos, w_i, que sean 0 o 1, es decir, que descarten o tomen completamente una característica. En el caso de realizarse este cambio, el problema que estamos tratando en esta práctica pasaría a mezclarse con un problema denominado *selección de características*, el cual consiste en seleccionar que características de un conjunto de datos influyen a la hora de realizar una posterior clasificación.

Sólo se ha tenido que cambiar las siguientes funciones con respecto a la propuesta original [3]:

```
1: function TRUNCAR(numero)
2:     salida = numero
3:     if numero < 0.1 then
4:         salida = 0
5:     else if numero > 0.9 then
6:         salida = 1
7:     end if
8:     return salida
9: end function
```

[3]El código no se ha entregado ya que no es requerido, en caso de querer verlo, mándeme un correo y se lo envío.

```
function cruceAritmetico(padre_1, padre_2)
    for i = 0, i < padre_1.size(), i += 1 do
        numero = (padre_1[i] + padre_2[i])/2
        hijo[i] = truncar(numero)
    end for
    return hijo
end function
```

```
function solInicialAleatoria(size)
    for i = 0, i < size, i += 1 do
        numero = random ∈ [0, 1]
        solucion_inicial[i] = truncar(numero)
    end for
    return solucion_inicial
end function
```

De esta manera conseguimos que cuando generemos un nuevo gen, en vez de truncarlo a 0 si está por debajo de 0 o a 1 si está por encima de 1, lo truncamos a 0 si está por debajo de α y a 1 si está por encima de $1 - \alpha$. En esta implementación se ha usado $\alpha = 0{,}1$. El resto de funciones no se han tenido que modificar, ya que hacían llamadas a la función *truncar*, así que con modificar esta última ha sido suficiente.

Para recoger datos se va a repetir una ejecución completa, exceptuando los algoritmos *1-NN* y *Relief*, ya que los resultados no van a variar. Los resultados obtenidos son los siguientes:

	LocalSearch			**AGG-BLX**		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	86.5385	1.221	23.3333	89.4231	10.731	13.3333
1 Test-Train	79.8077	1.96	6.66667	83.6538	10.998	8.33333
2. Train-Test	80.7692	1.008	11.6667	82.6923	10.804	10
2. Test-Train	83.6538	1.036	11.6667	78.8462	10.919	10
3. Train-Test	76.9231	1.356	5	72.1154	10.749	11.6667
3. Test-Train	87.5	1.064	11.6667	81.7308	10.977	13.3333
4. Train-Test	77.8846	1.36	13.3333	79.8077	10.825	8.33333
4. Test-Train	80.7692	1.233	13.3333	81.7308	10.949	8.33333
5. Train-Test	85.5769	1.262	13.3333	90.3846	10.938	11.6667
5. Test-Train	87.5	1.842	6.66667	85.5769	11.084	13.3333

Tabla 6.20: sonar (1)

	AGG-CA			AGE-BLX		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	90.3846	10.811	11.6667	91.3462	11.944	6.66667
1 Test-Train	82.6923	10.784	0	80.7692	12.056	10
2. Train-Test	86.5385	10.756	11.6667	78.8462	12.418	10
2. Test-Train	81.7308	10.733	1.66667	75.9615	12.009	11.6667
3. Train-Test	77.8846	10.833	15	80.7692	12.663	10
3. Test-Train	88.4615	10.824	8.33333	85.5769	12.115	11.6667
4. Train-Test	78.8462	10.899	5	77.8846	12.364	15
4. Test-Train	79.8077	10.919	1.66667	82.6923	12.673	18.3333
5. Train-Test	87.5	10.768	11.6667	89.4231	13.726	13.3333
5. Test-Train	89.4231	10.808	6.66667	86.5385	12.827	10

Tabla 6.21: sonar (2)

	AGE-CA			AM-10-1		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	91.3462	12.602	0	87.5	11.092	8.33333
1 Test-Train	80.7692	12.873	3.33333	80.7692	11.263	10
2. Train-Test	77.8846	12.049	1.66667	79.8077	10.927	8.33333
2. Test-Train	84.6154	12.042	6.66667	77.8846	11.313	8.33333
3. Train-Test	75.9615	12.998	1.66667	79.8077	10.834	15
3. Test-Train	85.5769	12.623	1.66667	81.7308	11.163	6.66667
4. Train-Test	77.8846	12.834	5	77.8846	11.008	8.33333
4. Test-Train	79.8077	12.89	1.66667	82.6923	11.221	10
5. Train-Test	88.4615	13.513	3.33333	85.5769	11.053	8.33333
5. Test-Train	86.5385	12.084	3.33333	86.5385	11.567	16.6667

Tabla 6.22: sonar (3)

	AM-10-0.1			AM-10-0.1mej		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	89.4231	10.963	15	87.5	10.825	13.3333
1 Test-Train	80.7692	11.068	13.3333	79.8077	10.961	13.3333
2. Train-Test	80.7692	10.789	11.6667	83.6538	10.983	15
2. Test-Train	77.8846	11.063	11.6667	81.7308	10.928	21.6667
3. Train-Test	76.9231	10.848	21.6667	77.8846	10.875	18.3333
3. Test-Train	82.6923	10.872	8.33333	83.6538	11.02	15
4. Train-Test	80.7692	10.79	16.6667	73.0769	10.945	23.3333
4. Test-Train	83.6538	11.012	5	78.8462	10.956	21.6667
5. Train-Test	87.5	10.907	11.6667	80.7692	10.854	16.6667
5. Test-Train	86.5385	11.072	11.6667	88.4615	11.025	15

Tabla 6.23: sonar (4)

	LocalSearch			AGG-BLX		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	83.913	8.463	21.0526	86.087	48.822	15.7895
1 Test-Train	80.8696	7.406	8.77193	87.8261	50.468	12.2807
2. Train-Test	86.5217	9.049	8.77193	83.913	49.368	8.77193
2. Test-Train	81.7391	6.712	7.01754	83.913	48.895	14.0351
3. Train-Test	83.4783	14.699	7.01754	84.3478	49.098	14.0351
3. Test-Train	83.4783	11.536	7.01754	83.4783	49.05	12.2807
4. Train-Test	80.4348	16.239	17.5439	83.4783	49.031	14.0351
4. Test-Train	80.4348	15.935	10.5263	82.1739	49.082	10.5263
5. Train-Test	84.3478	6.653	12.2807	88.2609	49.113	10.5263
5. Test-Train	79.1304	6.528	7.01754	80.4348	48.693	8.77193

Tabla 6.24: spambase (1)

	AGG-CA			AGE-BLX		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	83.913	48.678	1.75439	86.9565	53.014	7.01754
1 Test-Train	82.6087	48.75	7.01754	85.2174	53.603	3.50877
2. Train-Test	84.3478	48.944	5.26316	88.6957	53.558	14.0351
2. Test-Train	84.3478	48.769	5.26316	83.4783	53.874	7.01754
3. Train-Test	87.3913	48.714	8.77193	85.2174	53.869	10.5263
3. Test-Train	85.6522	48.509	1.75439	86.9565	52.896	8.77193
4. Train-Test	77.8261	48.515	8.77193	84.3478	53.12	14.0351
4. Test-Train	85.6522	48.733	7.01754	87.8261	53.907	3.50877
5. Train-Test	83.4783	48.502	8.77193	89.1304	53.681	10.5263
5. Test-Train	80	48.877	5.26316	80.8696	54.32	7.01754

Tabla 6.25: spambase (2)

	AGE-CA			AM-10-1		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	85.6522	53.36	1.75439	83.913	48.974	14.0351
1 Test-Train	85.2174	53.629	3.50877	86.087	49.61	8.77193
2. Train-Test	85.6522	53.481	1.75439	81.7391	49.413	7.01754
2. Test-Train	83.913	53.651	3.50877	83.4783	50.02	8.77193
3. Train-Test	81.3043	53.964	5.26316	83.913	49.37	5.26316
3. Test-Train	80.8696	53.254	7.01754	84.7826	49.967	8.77193
4. Train-Test	80.4348	52.696	5.26316	84.7826	48.886	17.5439
4. Test-Train	83.913	53.513	1.75439	85.2174	51.075	21.0526
5. Train-Test	84.7826	53.749	3.50877	85.6522	49.382	14.0351
5. Test-Train	83.4783	53.27	8.77193	82.1739	49.215	15.7895

Tabla 6.26: spambase (3)

	AM-10-0.1			AM-10-0.1mej		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	81.3043	48.948	14.0351	78.6957	48.97	12.2807
1 Test-Train	84.3478	48.82	8.77193	85.2174	49.122	14.0351
2. Train-Test	86.5217	49.193	12.2807	84.3478	48.949	22.807
2. Test-Train	87.3913	48.927	21.0526	83.4783	48.929	7.01754
3. Train-Test	82.6087	49.095	7.01754	80.8696	49.266	14.0351
3. Test-Train	85.6522	49.151	12.2807	78.6957	49.114	7.01754
4. Train-Test	83.913	48.691	15.7895	80	48.9	10.5263
4. Test-Train	87.8261	49.227	14.0351	83.4783	48.952	14.0351
5. Train-Test	83.913	49.082	12.2807	83.0435	49.134	12.2807
5. Test-Train	80.4348	48.919	12.2807	85.2174	49.016	12.2807

Tabla 6.27: spambase (4)

	LocalSearch			AGG-BLX		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	94.7368	3.608	10	94.386	37.609	13.3333
1 Test-Train	92.9577	1.65	6.66667	93.662	37.507	16.6667
2. Train-Test	94.0351	2.274	6.66667	93.6842	37.71	13.3333
2. Test-Train	94.7183	3.172	10	96.831	37.315	10
3. Train-Test	91.2281	2.039	6.66667	93.6842	37.845	10
3. Test-Train	95.7746	2.74	13.3333	96.1268	37.437	3.33333
4. Train-Test	94.386	1.918	10	96.4912	37.641	3.33333
4. Test-Train	91.9014	2.872	6.66667	95.0704	37.394	13.3333
5. Train-Test	92.9825	3.019	13.3333	93.6842	37.711	10
5. Test-Train	94.7183	3.384	13.3333	93.662	37.788	3.33333

Tabla 6.28: wdbc (1)

	AGG-CA			AGE-BLX		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	94.386	37.612	10	93.3333	41.14	16.6667
1 Test-Train	93.3099	37.818	6.66667	92.2535	43.053	6.66667
2. Train-Test	95.4386	37.872	0	94.7368	41.714	10
2. Test-Train	94.7183	37.823	3.33333	95.7746	41.85	6.66667
3. Train-Test	91.2281	37.634	23.3333	95.0877	41.442	10
3. Test-Train	95.7746	37.75	20	94.7183	42.514	0
4. Train-Test	95.7895	38.113	6.66667	95.7895	41.653	13.3333
4. Test-Train	92.9577	37.763	3.33333	94.0141	42.697	3.33333
5. Train-Test	94.7368	37.556	6.66667	94.386	41.85	10
5. Test-Train	94.0141	37.467	3.33333	94.0141	43.042	16.6667

Tabla 6.29: wdbc (2)

	AGE-CA			AM-10-1		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	95.0877	42.342	0	95.4386	37.845	6.66667
1 Test-Train	92.9577	43.033	13.3333	93.3099	38.635	10
2. Train-Test	96.8421	42.116	10	94.0351	39.943	6.66667
2. Test-Train	95.0704	42.316	6.66667	95.4225	38.784	10
3. Train-Test	92.9825	42.057	0	94.7368	37.712	13.3333
3. Test-Train	96.1268	42.162	0	95.4225	37.835	20
4. Train-Test	96.1404	41.838	3.33333	96.8421	37.813	13.3333
4. Test-Train	95.0704	42.025	6.66667	95.0704	37.659	10
5. Train-Test	93.6842	41.665	0	94.7368	37.538	10
5. Test-Train	94.3662	42.704	10	96.1268	38.042	13.3333

Tabla 6.30: wdbc (3)

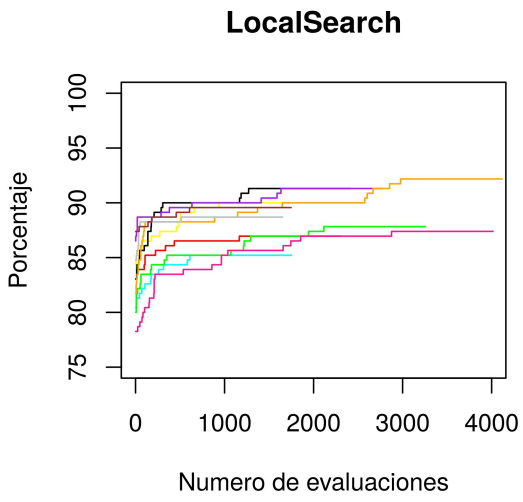
	AM-10-0.1			AM-10-0.1mej		
	%_clas	**T**	**%_red**	**%_clas**	**T**	**%_red**
1. Train-Test	95.0877	38.69	16.6667	94.386	41.613	10
1 Test-Train	94.0141	42.673	6.66667	93.3099	41.096	16.6667
2. Train-Test	94.0351	42.693	13.3333	94.386	40.054	13.3333
2. Test-Train	95.0704	38.397	16.6667	96.1268	39.785	20
3. Train-Test	95.0877	38.799	23.3333	93.3333	38.777	10
3. Test-Train	94.3662	38.23	10	95.4225	38.114	16.6667
4. Train-Test	96.1404	38.56	20	96.1404	38.537	6.66667
4. Test-Train	94.7183	38.045	10	94.3662	37.927	23.3333
5. Train-Test	94.386	38.632	16.6667	94.0351	38.435	23.3333
5. Test-Train	94.0141	42.753	16.6667	95.0704	37.851	13.3333

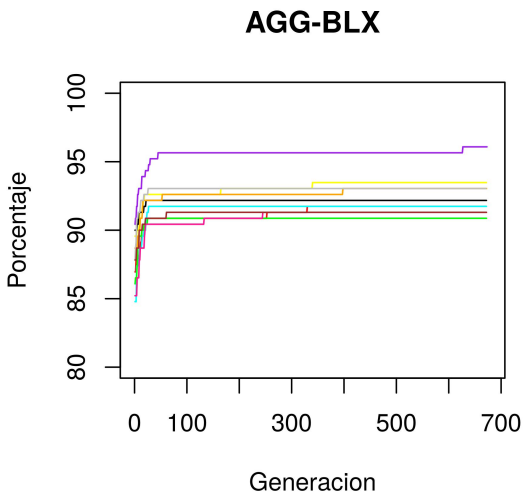
Tabla 6.31: wdbc (4)

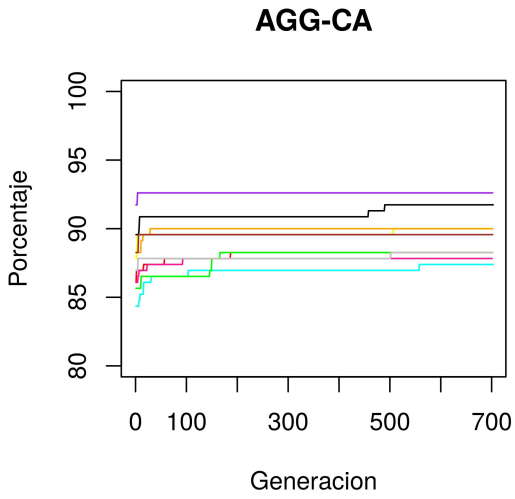
Podemos ver que el porcentaje de acierto ha variado, mejorando en algunos casos y empeorando en otros. En cuanto al tiempo se mantiene más o menos igual, ya que el código apenas ha variado. Pero con la idea implementada el objetivo no es mejorar el porcentaje que nos da el vector de pesos, sino recudir el número de características (tener el mayor número de ceros posibles en el vector de pesos). Esto lo podemos ver en las columnas llamadas *%_red* (porcentaje de reducción), el cual se calcula contando el número de ceros en el vector de pesos y dividido por su tamaño. Con esta técnica conseguimos soluciones más simples ya que al aparecer pesos a 0 obtenemos características que no se contemplan.

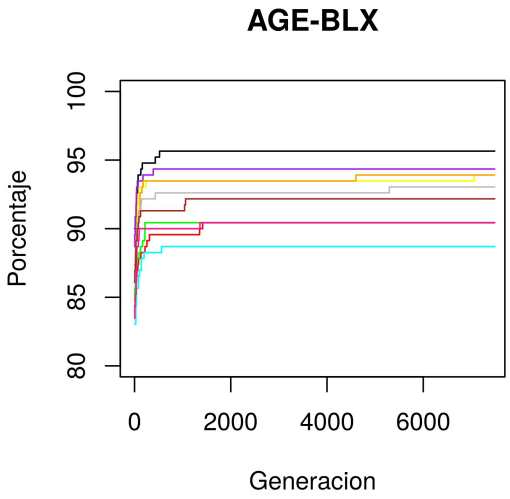
6.3.1. Estudio de la convergencia (dos).

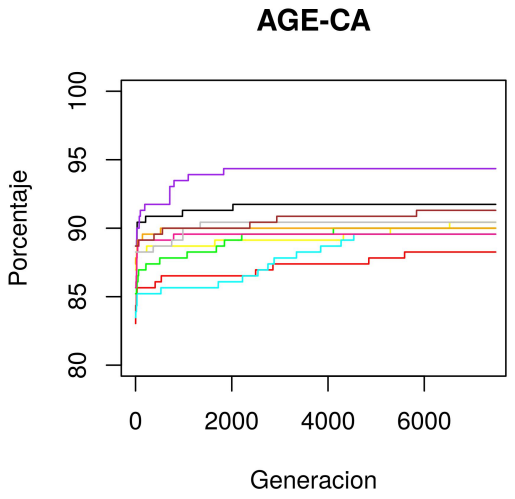
Voy a hacer lo mismo que hice para la idea original de la práctica, un pequeño estudio de la convergencia de los algoritmos. Los pasos del estudio son los mismos que los aplicados en el caso inicial, así que vamos directamente con las gráficas para el conjunto de datos *spambase*:

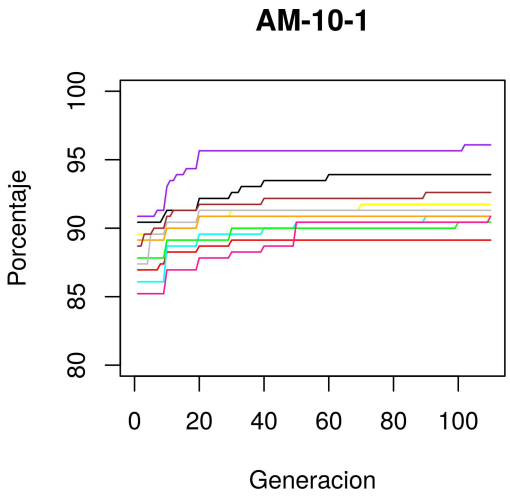

LocalSearch
Porcentaje
75 80 85 90 95 100
0 1000 2000 3000 4000
Numero de evaluaciones

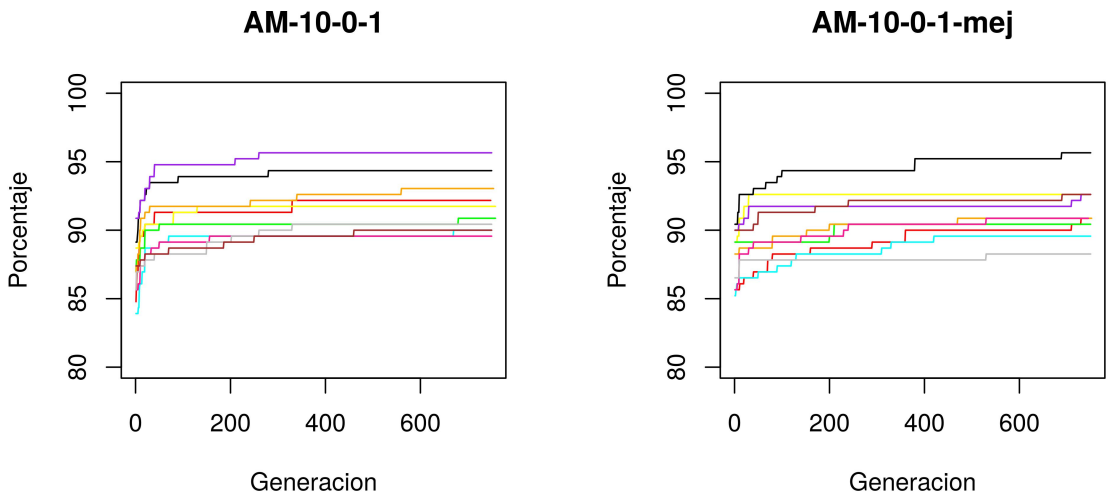

AGG-BLX
Porcentaje
80 85 90 95 100
0 100 300 500 700
Generacion


AGG-CA
Porcentaje
80 85 90 95 100
0 100 300 500 700
Generacion


AGE-BLX
Porcentaje
80 85 90 95 100
0 2000 4000 6000
Generacion


AGE-CA
Porcentaje
80 85 90 95 100
0 2000 4000 6000
Generacion


AM-10-1
Porcentaje
80 85 90 95 100
0 20 40 60 80 100
Generacion



En cada gráfica se ha pintado el porcentaje obtenido por el mejor vector de pesos encontrado hasta el momento para cada iteración o para cada generación en el caso de los algoritmos evolutivos y para todas las particiones del conjunto de datos *spambase*. Concretamente:

- Partición 1, aprendemos con *train* → color rojo.
- Partición 1, aprendemos con *test* → color amarillo.
- Partición 2, aprendemos con *train* → color negro.
- Partición 2, aprendemos con *test* → color azul cyan.
- Partición 3, aprendemos con *train* → color verde.
- Partición 3, aprendemos con *test* → color morado.
- Partición 4, aprendemos con *train* → color naranja.
- Partición 4, aprendemos con *test* → color rosa fucsia.
- Partición 5, aprendemos con *train* → color gris.
- Partición 5, aprendemos con *test* → color marrón.

Podemos ver que convergen igual que en la idea original y que se siguen estancando rápidamente. Esto se debe a que los conjuntos de prueba son pequeños.

6.4. Conclusión.

Fomentar que los pesos sean 0 y 1, particularmente que sean 0, es una buena idea, ya que obtenemos soluciones más simples y siguen manteniendo la calidad de las soluciones originales. Esta idea es con la que se va a trabajar en la segunda práctica.

7. Bibliografía.

No se ha usado ninguna bibliografía concreta más allá de los apuntes de teoría y seminarios aportados de la signatura.

Para el código desarrollado en *R* se ha usado como apoyo los propios manuales de *.R* (mediante el comando *?**, donde *** es la instrucción de la que se quiere obtener información).

Para el código desarrollado en *C++*, como ya se ha comentado anteriormente, se ha usado la página cplusplus para consultar ciertas funciones.