UNIVERSIDAD DE GRANADA

E.T.S.I INFORMÁTICA Y TELECOMUNICACIONES

Práctica 1: Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

Curso 2016-2017

Problema del Aprendizaje de Pesos en Características

Algoritmos considerados: 1-NN, RELIEF, BL, AGG-BLX, AGG-CA, AGE-BLX, AGE-CA, AM-(10,1.0), AM-(10,0.1) y AM(10,0.1mej)

Arthur Mickael Rodríguez Nesterenko

DNI: Y1680851W

E-mail: arthur18@correo.ugr.es

Grupo 2: Viernes 17:30 – 19:30

Indice de contenidos

1. Descripción del problema

El objetivo principal de esta práctica trata sobre la utilización de las ***Técnicas de Búsqueda Basadas en Poblaciones*** para la resolución del problema del aprendizaje de pesos en características (APC). En este tipo de problemas disponemos de un conjunto predeterminado de objetos con una clasificación previa, y los mismos vienen representados en función de sus valores para una serie de atributos. Cada objeto pertenecerá a una determinada clase dentro de un conjunto $\{C_1,\ldots,C_M\}$ y nuestro objetivo principal es diseñar un sistema que me permita clasificar esos objetos de forma automática.

Para diseñar este sistema serán necesarias dos tareas: ***Aprendizaje*** y ***Validación***. Dividiremos el conjunto de objetos en dos subconjuntos: un subconjunto de ***Entrenamiento*** que será el que utilizaremos para aprender el clasificador y otro subconjunto de ***Prueba*** que será el que validará el clasificador aprendido. Estos subconjuntos se construirán siguiendo la técnica de validación ***5x2 cross-validation***: 5 particiones distintas de los datos al 50%, con una mitad aprendemos el clasificador y con la otra lo validamos, y acto seguido repetimos el proceso a la inversa.

El clasificador utilizado será el ***K-NN (con K=1),*** que básicamente le asigna a un objeto e' la clasificación c_{min} de un objeto perteneciente al subconjunto $\{e_1,\ldots,e_m\}$ tal que la distancia Euclídea entre e' y e_i sea mínima. Este clasificador será nuestra función de evaluación a la hora de que, a través de distintas técnicas, asignemos ponderaciones (pesos) a cada una de las características del ejemplo, en cuanto a términos de clasificación se refiere.

Por lo tanto nuestro objetivo será ajustar un conjunto de ponderaciones $W=\{w_1,\ldots,w_n\}$
donde ***n*** representa el número de características de cada ejemplo, de tal forma que los clasificadores que construyamos a partir de él sean mejores. Para encontrar estas ponderaciones vamos a valernos de distintas técnicas como la búsqueda secuencial (***RELIEF***), búsqueda local (***BL***) y búsqueda con metaheurísticas donde entrarán en acción las principales técnicas: ***algoritmos genéticos generacionales y estacionarios*** (cada uno con un operador de cruce, tanto ***BLX*** como ***CA***) y los ***algoritmos meméticos***, hibridación de los primeros con una BL que permite explotar las soluciones encontradas.

Para desarrollar la práctica nos valdremos de tres bases de datos proporcionadas por los profesores: ***Sonar***, una base de datos de detección de materiales mediante señales de sonar, discriminando entre objetos metálicos y rocas, ***Wdbc***, base de datos que contiene atributos calculados a partir de una imagen digitalizada de una aspiración con aguja fina de una masa en la mama y ***Spambase,*** una base de datos de detección de SPAM frente a correo electrónico seguro**.**

2. Introducción a la aplicación de los algoritmo empleados

2.1 Esquema de representación de las soluciones

La representación de las soluciones comienza organizando una serie de datos en una clase llamada ***ConjuntoDatos*** que será la que encapsule la información necesaria para el funcionamiento de los algoritmos así como los propios algoritmos. El conjunto de vectores de características, así como las clasificaciones correspondientes a cada uno e incluso las particiones generadas para la base de datos a estudiar se encapsulan dentro de esta clase. Los algoritmos proporcionan como salida un ***vector de pesos*** con los que posteriormente se evaluará la partición de ***test***, por lo tanto una vez conocida la organización de la información vamos a realizar una descripción detallada de los operadores y operaciones comunes a todos los algoritmos, así como de la función objetivo, todo en un orden cronológico desde que se lee el fichero de datos, pasando por el encapsulamiento de la información y la preparación de la misma para la ejecución de los algoritmos.

2.2 Operadores y operaciones comunes – Pseudocódigo

2.2.1 Lectura de fichero de datos, normalización y creación de particiones

En el fichero fuente ***Utilidades.cpp*** se implementan dos funciones que sirven para leer los datos directamente desde los ficheros ***.arff***. La función con sufijo ****_Formato1*** me permite leer los ficheros ***sonar.arff*** y ***spambase-460.arff*** ya que ambos cuentan con la misma estructura: por cada línea de datos nos encontramos al final de ésta con la clasificación para ese conjunto de datos. La función ****_Formato2*** será la que me permita leer el fichero ***wdbc.arff*** ya que este tiene un formato ligeramente distinto: la etiqueta de clasificación al principio y después todos los datos.

leerDatosFicheroARFF_Formato1:
Abrir(flujo de entrada)
Saltar Lineas (2)
Leer (Primer atributo)
Inicializar (contador de atributos) = -1
Mientras primer carácter == @ hacer
    Leer (linea)
    contador de atributos++
Fin Mientras
Saltar Lineas(1)
Leer (Primera característica)
Mientras !Fin de Fichero hacer
    caracteristicas.añadir(caracteristica)
    Para todo i desde 1 hasta contador de atributos
        Leer(caracteristica)
        caracteristicas.añadir(caracteristica)
    Fin Para todo
    datos.añadir(caracteristicas)
    Leer(clasificacion)
    clasificaciones.añadir(clasificaciones)

Fin Mientras
Cerrar(flujo de entrada)

La descripción en pseudocódigo de ***leerDatosFicheroARFF_Formato2*** es innecesaria si se matiza que ahora lo primero que hacemos antes de entrar a *Mientras !Fin de Fichero es Leer(clasificacion)* para después añadirla a *clasificaciones*, además de que el bucle interno se convierte en *Para todo i=0 && i<contador de atributos* ya que ahora por delante tiene todos los atributos/características.

Una vez leídos los datos el siguiente paso consiste en encapsularlos dentro de la clase. Primero tenemos que normalizarlos, pero dada que la operación de normalización es irrelevante no la incluiremos en esta memoria (en cuanto a pseudocódigo) pero si la explicaremos brevemente: obtenemos la matriz traspuesta en la que cada fila representará una columna de características, obtenemos el máximo y mínimo de cada fila y posteriormente normalizamos los datos originales. Una vez realizada la normalización pasamos a generar las particiones para el ***5x2 cross-validation***, operación sencilla que no precisa ser incluida en pseudocódigo. Llegados a este punto estamos listos para describir los operadores comunes.

2.2.2 Operadores comunes

a) Generador de mutación/vecino: partimos de un generador de números dentro de una distribución normal de ***media 0*** y ***varianza 0.3*** (parámetros establecidos por defecto). Este operador sirve tanto en la **BL** para **generar vecinos** como para **generar mutaciones** en los **A.Gs.**

Generar vecino-mutación

Obtener el valor perteneciente a la distribución normal
Actualizar Pesos(componente, valor normal)
Si Pesos (componente) > 1 entonces
    Pesos(componente) =1
Si Pesos(componente) < 0 entonces
    Pesos(componente) = 0
Devolver Pesos

b) Generación de soluciones aleatorias: la generación de soluciones aleatorias es un proceso sencillo, donde *C* representará el número de cromosomas (soluciones) y *G* el número de genes de cada cromosoma. El algoritmo devuelve la Población generada de forma aleatoria.

Generar soluciones aleatorias

Para i=0 hasta C hacer
    Para j=0 hasta G hacer
        cromosoma ← generar valor aleatorio (0, 10000) /10000
    Fin Para (2)
    Poblacion ← cromosoma
    Vaciar(cromosoma)
Fin Para (1)
Devolver Poblacion

c) Operador de selección de los A.Gs – Torneo Binario: Se eligen de forma aleatoria dos padres y quien tenga mejor tasa de clasificación) será el que seleccionemos. Así, para los AGG aplicaremos tantos torneos como cromosomas de la población, de forma que se generen la misma cantidad de

posibles padres a cruzar. En los AGE aplicaremos el torneo dos veces para seleccionar los dos padres a cruzar. Describimos el algoritmo a continuación: *P* representa el número de padres a generar y *C* el tamaño de la población. El algoritmo devuelve los *Padres* a cruzar.

Torneo Binario

Para i=0 hasta P hacer
padre1 ← generar valor aleatorio (0, C)
padre2 ← generar valor aleatorio (0, C)
Mientras padre1 == padre2 hacer
padre2 ← generar valor aleatorio (0, C)

Si Evaluar(padre1) > Evaluar(padre2) entonces
Padres ← padre1
Si no
Padres ← padre2
Fin Para (1)
Devolver Padres

d) Operador de generación de gen aleatorio perteneciente a un intervalo: es el operador necesario para generar un gen perteneciente a un intervalo, mecanismo utilizado en el operador ***BLX-α***. Recibe una cota superior (*CS*) e inferior (*CI)* para generar el valor aleatorio y devuelve el *Gen*.

Generar gen aleatorio

Gen = generar valor aleatorio (CI, CS)
Si Gen > 1 entonces
Gen =1
Si Gen < 0 entonces
Gen = 0
Devolver Gen

e) Operadores de cruce BLX-α y CA: el operador *BLX-α* calcula un intervalo [*CI,CS*] en función de los valores de los *G* genes (máximos y mínimos) de cada uno de los dos padres a cruzar, intervalo que tendrá una parte de *Exploración* y otra de *Explotación*, generando dos hijos. El Cruce Aritmético es sencillo: se suman ambos padres componentes a componentes y se dividen por dos, dando como resultado del cruce un hijo. Ambos operadores devuelven el/los descendiente/s de cruzar los dos padres.

BLX-α con α=0.3 por defecto (leído de fichero)

Calcular máximo y mínimo de padre 1: M_1,m_1
Calcular máximo y mínimo de padre 2: M_2,m_2
Calcular máximo (M_1 , M_2) y mínimo (m_1 , m_2): MAX, MIN
I = MAX -MIN
CS = MAX + (I α)*
CI = MIN - (I α)*

Para i=0 hasta G

Gen1 ← generar gen aleatorio (CI,CS)
Gen2 ← generar gen aleatorio(CI,CS)
Hijo1 ← Gen1
Hijo2 ← Gen2

Devolver Hijo1, Hijo2
CA (Cruce aritmético)

Para i=0 hasta G
Hijo ← (Padre1(i) + Padre2(i))/2
Devolver Hijo

f) Operador de cálculo de la Distancia Euclídea: este operador, imprescindible en nuestra función objetivo, calcula la distancia Euclídea entre dos ejemplos *P1* y *P2* teniendo en cuenta además un vector de pesos *W* para ponderar cada una de las *G* características del ejemplo *P1*. Devuelve el cálculo de la distancia Euclídea haciendo uso del desenrollado de bucles para reducir el número de saltos en el cálculo.

Calcular distancia Euclídea

Para i=0 hasta G hacer
*Distancia ← W_i * $(P1_i - P2_i)^2$*
Distancia ← √Distancia
Devolver Distancia

g) Función objetivo 1-NN: nuestro objetivo será maximizar la tasa de acierto que proporciona nuestro clasificador 1-NN sobre un determinado subconjunto que llamaremos *Train* en base a otro subconjunto *Test*, teniendo en cuenta unos determinados pesos *W* y utilizando la técnica *leave-one-out* cuando sea necesaria. A continuación se muestra el pseudocódigo de nuestro clasificador, que devuelve la *tasa de clasificación* teniendo en cuenta los individuos *N* que forman parte de *Train* en función de la clase c_{min} que se le asigne tras las evaluaciones de distancias euclídeas.

Clasificador 1-NN: función objetivo

Para cada T_i en Train hacer
inicializar(d_{min})
Para cada T_j en Test hacer
Si T_i != T_j // Leave one out
d_j ← calcular distancia Euclidea (T_i , T_j)
Si d_j < d_{min} entonces
d_{min} = d_j
c_{min} = clase(T_j)
Fin Para (2)
Si c_{min} == clase(T_i) entonces
contabilizar acierto
Fin Para (1)
tasa de clasificacion ← 100(aciertos/N)
Devolver tasa de clasificacion

3. Estructura del método de búsqueda y operaciones relevantes de los algoritmos

En este apartado vamos a abordar los aspectos de la implementación de los distintos métodos de búsqueda considerados a lo largo de la práctica. Primero vamos a describir el **método de exploración del entorno** que utiliza la **BL** (sin pasar por detalles como la inicialización de distintas estructuras utilizadas, ya que se encuentran comentadas en el código fuente); una vez explicado todo este procedimiento pasaremos con los **A.Gs**, tratando principalmente **dos esquemas**: el de **reemplazamiento de la población** en la generación *t* por aquella de la generación *t+1,* siendo *t* una iteración en la que se genera una nueva población, y el **esquema de evolución**, pasando por las **decisiones de mutación** y la garantía de **conservar el mejor padre** de una generación a otra. Por último para los **A.Ms** trataremos el **esquema de selección de cromosomas** a los que se les aplicará la BL, así como su **evolución** dentro de ésta. El pseudocódigo de cada algoritmo refleja exhaustivamente la implementación del código fuente, sin llegar a ser demasiado general ni entrar en cuestiones específicas de implementación. Las características mencionadas anteriormente para cada algoritmo aparecerán comentadas en negrita para localizarlas mejor.

3.1 Búsqueda Local (BL): exploración del entorno

La búsqueda local parte de un conjunto de pesos *W* generados de forma aleatoria y de una partición determinada de entrenamiento *Train* con *N* características, sobre la que efectuará las exploraciones para mejorar *W* en función de la *tasa de clasificación* que proporciona la función objetivo *Clasificador 1-NN*. Mientras queden vecinos que generar (máximo de *20*N)* seleccionamos de forma aleatoria una de las *N* componentes para generar un vecino y evaluamos la tasa de clasificación del vecino generado: si es mejor que la actual, éste sustituirá la solución actual. Se realizará un máximo de 15000 evaluaciones de la función objetivo antes de terminar la búsqueda. El algoritmo devuelve la solución *W* que mejores resultados ha conseguido mostrar.

BL : exploración del entorno

// Inicialización//
W ← generar solución aleatoria
clasificacion ← 1-NN(Train,W)

*Mientras vecinos generados < 20*N & evaluaciones <15000 hacer*
    aleatorizar(N)

    // Exploración del entorno//
    Para cada N_i en N & no hay mejora hacer
        W_i ← Generar vecino (W, N_i)
        $clasificacion_i$ ← 1-NN(Train,W_i)

        // Entrar en este condicional indica mejora//
        Si $clasificacion_i$ > clasificacion entonces
            clasificacion = $clasificacion_i$
            W = W_i
    Fin Para cada
Fin Mientras

Devolver W

3.2 Algoritmos Genéticos Generacionales (AGG) y estacionarios (AGE): esquema de reemplazamiento y evolución, mutación y conservación del mejor padre

Antes de comenzar con las particularidades de cada algoritmo vamos a expresar las generalidades de cada uno de ellos, ya que se repiten a lo largo de los 4 tipos de algoritmos:

a) Evaluación de la población en la generación *t*: el esquema de evaluación de una población, ya sea la inicial o la que se obtiene tras el cruce de los mejores padres en una generación *t*, es siempre el mismo. Sea *N* el tamaño de la población *P*, B_N el mejor padre de la generación actual (con una tasa de clasificación $T(B_N)$ y W_N el mejor padre de la generación actual (con una tasa de clasificación $T(W_N)$ *y* , se realiza el siguiente proceso sobre una partición que llamaremos *Train*. El proceso devuelve el conjunto de tasas de *clasificaciones* para cada individuo de *P*.

Evaluar población de la generación t
B_N , $W_N \leftarrow -1$
Para i=0 hasta N hacer

// Obtener la tasa de clasificacion de ese cromosoma con respecto a la partición//
$clasificacion_i \leftarrow$ *1-NN(Train,*P_i*)*
clasificaciones $\leftarrow$ $clasificacion_i$

// Guardar el mejor padre//
Si $clasificacion_i > T(B_N)$ *entonces*
$B_N = i$
$T(B_N) =$ $clasificacion_i$

// Guardar el peor padre//
Si $clasificacion_i < T(W_N)$ *entonces*
$W_N = i$
$T(W_N) =$ $clasificacion_i$

Fin para (1)
Devuelve clasificaciones

b) Decisiones en la operación de mutación: otra de las generalidades que nos encontramos en los algoritmos genéticos es la operación de mutación basada en la generación de un vecino por mutación normal a partir de una población. El número de mutaciones *M* se calcula previamente, siendo redondeado al alza (decisión propia) de forma que por cada generación siempre se mute, al menos, un gen de un cromosoma. Se generan dos números aleatorios para decidir, con el primero, el cromosoma de los *C* a mutar y con el segundo, el gen de los *G* a mutar.

Operación de mutación

Para i=0 hasta numero de mutaciones
cromosoma ← generar valor aleatorio (0, C)
gen ← generar valor aleatorio (0, G)
población ← generar vecino/mutación(cromosoma,gen)

Fin Para

3.2.1 AGG - BLX

Operación de cruce (BLX): la operación de cruce posterior al Torneo Binario se puede ver a continuación. Partiendo de un conjunto de *mejores* padres que hemos seleccionado en el Torneo Binario y de un número de *cruces* que se determina de forma previa una vez conocida la *probabilidad de cruce*, realizamos el siguiente proceso para obtener una nueva población. También se considera como parte de la operación de cruce el completar la nueva población con aquellos mejores padres que no se han cruzado sólo hasta completar el tamaño de la población deseada *N*. Cabe destacar que en este paso se comprueba si se ha perdido el mejor padre (M_N) de la generación anterior.

Esquema de evolución – Generación t+1

// Primera parte de la evolución: cruce de los mejores padres//
Para i=0 hasta cruces hacer
Operador de cruce BLX-α (mejores(pareja 1)): Hijo1,Hijo2
población ← Hijo1, Hijo2
Fin Para (1)

//Segunda parte: completar con los mejores padres de la generación anterior//
Para i= tamaño población actual hasta N
población ← mejores(i)
Si mejores(i) == B_N
no se ha perdido el mejor padre
Devuelve población

Teniendo en cuenta los operadores comunes a todos los algoritmos (Apartado 2 de la memoria) y las generalidades que nos encontraremos en los A.Gs (Apartado 3.2) podemos pasar a describir el pseudocódigo del algoritmo AGG-BLX. Cabe destacar que no entramos en declaración de variables necesarias, ya que es un tema más de implementación de código. Partimos de una población inicial P_t (generada de forma aleatoria) y en cada iteración generamos una población P_{t+1}. Cada población cuenta con $C=30$ cromosomas y G genes por cromosoma y se evalúa cada uno en base a una partición *Train*. Se realizarán un numero máximo de evaluaciones de la función objetivo igual a 15000. El algoritmo devuelve la mejor solución, entendido como el mejor cromosoma, que representa un vector de pesos que posteriormente utilizaremos en el proceso de validación. Hay que tener en cuenta la operación de cruce con el operador BLX, descrita previamente y además válida para los AGEs.

AGG-BLX

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
clasificaciones$_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

// Esquema de evolución//

padres ← Torneo Binario(P_t)
P_{t+1} ← Esquema de evolución BLX (padres)

// Operación de mutación con la población P_{t+1} //
P_{t+1} ← operación de mutación(P_{t+1})

//Evaluamos la nueva población //
$clasificaciones_{t+1}$ ← evaluar población (P_{t+1})
evaluaciones += C

//Comprobación en caso de que el mejor padre se la población P_t no se encuentre //en la población P_{t+1}
Si B_N (P_t) se ha perdido entonces
W_N (P_{t+1}) = B_N (P_t)

//Actualización del mejor cromosoma en caso de ocurrir
Si $T(B_N(P_{t+1}))$ < $T(B_N(P_t))$ entonces
$T(B_N(P_{t+1}))$ = $T(B(P_t))$
Si no entonces
B_N (P_t) = B_N (P_{t+1})

//Esquema de reemplazamiento una vez hechos los cruces y mutaciones
P_t = P_{t+1}
$clasificaciones_t$ = $clasificaciones_{t+1}$

Fin Mientras
//Devolvemos el mejor padre de la ultima generación
Devolver $B_N(P_t)$

3.2.2 AGG – CA

Volvemos a considerar los operadores comunes a todos los algoritmos (Apartado 2 de la memoria) y las generalidades que nos encontraremos en los A.Gs (Apartado 3.2) para describir el pseudocódigo del algoritmo AGG-CA. Partimos de una población inicial P_t (generada de forma aleatoria) y en cada iteración generamos una población P_{t+1}. Cada población cuenta con *C* cromosomas y *G* genes por cromosoma y se evalúa cada uno en base a una partición *Train*. Se realizarán un numero máximo de evaluaciones de la función objetivo igual a 15000. El algoritmo devuelve la mejor solución, entendido como el mejor cromosoma, que representa un vector de pesos que posteriormente utilizaremos en el proceso de validación. Hay que tener en cuenta la operación de cruce con el operador CA, descrita previamente y además válida para los AGEs.

AGG-CA

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
$clasificaciones_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //

Mientras evaluaciones < 15000 hacer

// Esquema de evolución//
padres ← Torneo Binario(P_t)

// Primera parte de la evolución: cruce de los mejores padres//
Para i=0 hasta cruces hacer
Operador de cruce CA (mejores(pareja 1)): Hijo1
población ← Hijo1
Fin Para (1)

//Segunda parte: completar con los mejores padres de la generación anterior//
Para i= tamaño población actual hasta N
*población ← mejores(i*2)*
Si mejores(i) == B_N
no se ha perdido el mejor padre

// Operación de mutación con la población P_{t+1} //
P_{t+1} ← operación de mutación(P_{t+1})

//Evaluamos la nueva población //
clasificaciones$_{t+1}$ ← evaluar población (P_{t+1})
evaluaciones += C

//Comprobación en caso de que el mejor padre se la población P_t no se encuentre
//en la población P_{t+1}
Si B_N (P_t) se ha perdido entonces
W_N (P_{t+1}) = B_N (P_t)

//Actualización del mejor cromosoma en caso de ocurrir
Si T(B_N(P_{t+1})) < T(B_N(P_t)) entonces
T(B_N(P_{t+1})) = T(B(P_t))
Si no entonces
B_N (P_t) = B_N (P_{t+1})

//Esquema de reemplazamiento una vez hechos los cruces y mutaciones
P_t = P_{t+1}
clasificaciones$_t$ = clasificaciones$_{t+1}$

Fin Mientras
//Devolvemos el mejor padre de la ultima generación
Devolver B_N(P_t)

3.2.3 AGE – BLX

Para los algoritmos genéticos estacionarios el reemplazamiento de la población ocurre de forma distinta a como ocurre en los generacionales: se generarán únicamente dos hijos que provienen de los mejores padres, por lo que el Torneo Binario se ejecutará 2 veces para el operador BLX y 4 veces para el CA (dado que necesitamos 2 parejas para generar 2 hijos). Por lo tanto si partimos de una población P_t de tamaño $N=30$, aplicaremos la operación de cruce pertinente para generar los dos hijos, posteriormente realizaremos la mutación sobre la propia población (recalculando la tasa de clasificación de los cromosomas que se han mutado – siempre se muta al menos 1 cromosoma por generación), calcularemos los dos peores padres de la población P_t y en última instancia sustituiremos los dos hijos por los dos peores padres en caso de que éstos sean mejores (se garantiza que de ser al menos uno mejor, ese siempre sustituirá al peor padre). El algoritmo devuelve la mejor solución, entendido como el mejor cromosoma, que representa un vector de pesos que posteriormente utilizaremos en el proceso de validación.

AGE-BLX

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
$clasificaciones_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

*// **Esquema de evolución**//*
padres ← Torneo Binario(P_t)

*// **Cruzamos la pareja de mejores padres y generamos los hijos***
hijos ← Esquema de evolución BLX (padres)

*// **Operación de mutación con la población P_t**//*
P_t ← operación de mutación(P_t)
$clasificaciones_t$(mutados) ← evaluar población (P_t(mutados))

//Calculamos los dos peores padres
Si $clasificaciones_t$(0) < $clasificaciones_t$(1) entonces
peor1 = 0, peor2 =1
Si no
peor1 =1, peor2 =0

Para i=2 hasta N
Si $clasificaciones_t$(i) < $clasificaciones_t$(peor1) entonces
peor2= peor1
peor1= i
Si $clasificaciones_t$(i) < $clasificaciones_t$(peor2) entonces
peor2 = i
Fin Para

//Evaluamos los hijos antes de decidir si reemplazarlos o no
$clasificaciones_{hijos}$ ← evaluar población (hijos)
ordenar hijos en orden decreciente de $clasificaciones_{hijos}$

//Reemplazamiento en caso de ocurrir
Si mejor(hijos) > $clasificaciones_t$(peor1) entonces
P_t(peor1) = mejor(hijos)
$clasificaciones_t$(peor1) = $clasificaciones_{hijos}$(mejor)

Si segundo mejor(hijos) > $clasificaciones_t$(peor2) entonces
P_t(peor2) = segundo mejor(hijos)
$clasificaciones_t$(peor2) = $clasificaciones_{hijos}$(segundo mejor)

Fin Mientras

//Calculo de la mejor solución
mejor = 0
Para i=1 hasta N hacer
Si $clasificaciones_t$(i) > $clasificaciones_t$(mejor)
mejor = i

Devolver P_t (mejor)

3.2.4 AGE – CA

El único cambio en cuanto a funcionamiento de este algoritmo respecto de AGE-BLX es únicamente el operador de cruce, que utiliza 4 padres (2 parejas) para generar los dos hijos, por lo que mantenemos todas las demás condiciones igual y pasamos a describir el pseudocódigo del algoritmo. El algoritmo devuelve naturalmente la mejor de las soluciones tras calcularla cuando acaba la búsqueda por la población. En negrita se puede apreciar el único aspecto que cambia frente a su homólogo, AGE-BLX.

AGE-CA

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
$clasificaciones_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

// Esquema de evolución//
padres ← Torneo Binario(P_t)

// Cruzamos las dos parejas de mejores padres y generamos los hijos
hijos ← Esquema de evolución CA (padres(pareja1))
hijos ← Esquema de evolución CA (padres(pareja2))

*// **Operación de mutación con la población P_t**//*
$P_t \leftarrow$ operación de mutación(P_t)
clasificaciones$_t$(mutados) $\leftarrow$ evaluar población (P_t(mutados))

//Calculamos los dos peores padres
Si clasificaciones$_t$(0) < clasificaciones$_t$(1) entonces
peor1 = 0, peor2 =1
Si no
peor1 =1, peor2 =0

Para i=2 hasta N
Si clasificaciones$_t$(i) < clasificaciones$_t$(peor1) entonces
peor2= peor1
peor1= i
Si clasificaciones$_t$(i) < clasificaciones$_t$(peor2) entonces
peor2 = i
Fin Para

//Evaluamos los hijos antes de decidir si reemplazarlos o no
clasificaciones$_{hijos}$ $\leftarrow$ evaluar población (hijos)
ordenar hijos en orden decreciente de clasificaciones$_{hijos}$

*//**Reemplazamiento en caso de ocurrir***
Si mejor(hijos) > clasificaciones$_t$(peor1) entonces
P_t(peor1) = mejor(hijos)
clasificaciones$_t$(peor1) = clasificaciones$_{hijos}$(mejor)

Si segundo mejor(hijos) > clasificaciones$_t$(peor2) entonces
P_t(peor2) = segundo mejor(hijos)
clasificaciones$_t$(peor2) = clasificaciones$_{hijos}$(segundo mejor)

Fin Mientras

//Calculo de la mejor solución
mejor = 0
Para i=1 hasta N hacer
Si clasificaciones$_t$(i) > clasificaciones$_t$(mejor)
mejor = i

Devolver P_t (mejor)

3.2.5 AM – (10,1.0)

Los tres tipos de algoritmos meméticos que implementaremos son hibridaciones del algoritmo genético generacional con el operador CA, ya que éste ha sido el que ha proporcionado resultados ligeramente mejores. Como los algoritmos meméticos aplican la búsqueda local (BL) sobre un conjunto determinado de cromosomas, vamos a remarcar en el pseudocódigo de nuestro algoritmo esas dos operaciones: el **esquema de selección de cromosomas** a los que se les aplicará la BL, así como su **evolución** dentro de ésta. Estas operaciones aparecerán comentadas en negrita para localizarlas con mayor facilidad. Partimos de los mismos principios: una población P_t de tamaño $N=10$ con G genes ,en la que siempre se muta al menos 1 cromosoma por generación, en esta versión del AM tendremos en cuenta que la BL se aplicará sobre **todos** los cromosomas, generando un máximo de 2*G vecinos para cada cromosoma. La BL se aplicará cada 10 generaciones y se considerará la partición de *Train* para obtener las tasas de clasificación en función de los pesos (cromosomas) que se generen. El pseudocódigo se indica a continuación:

AM-(10,1.0)

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
clasificaciones$_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

// Esquema de evolución//
padres ← Torneo Binario(P_t)

// Primera parte de la evolución: cruce de los mejores padres//
Para i=0 hasta cruces hacer
Operador de cruce CA (mejores(pareja 1)): Hijo1
población ← Hijo1
Fin Para (1)

//Segunda parte: completar con los mejores padres de la generación anterior//
Para i= tamaño población actual hasta N
*población ← mejores(i*2)*
Si mejores(i) == B_N
no se ha perdido el mejor padre

// Operación de mutación con la población P_{t+1} //
P_{t+1} ← operación de mutación(P_{t+1})

//Actualizamos el numero de generaciones
generaciones++

//Aplicación de la BL a toda la población//
Si generaciones es múltiplo de 10 entonces
Para i=0 hasta N hacer
BL(Train, P_{t+1} (i), 2G vecinos)

evaluaciones += 2G +1

//Evaluamos la nueva población //
clasificaciones$_{t+1}$ ← evaluar población (P_{t+1})
evaluaciones += C

//Comprobación en caso de que el mejor padre se la población P_t no se encuentre //en la población P_{t+1}
Si B_N (P_t) se ha perdido entonces
$W_N(P_{t+1}) = B_N$ (P_t)

//Actualización del mejor cromosoma en caso de ocurrir
Si $T(B_N(P_{t+1}))$ < $T(B_N(P_t))$ entonces
$T(B_N(P_{t+1})) = T(B(P_t))$
Si no entonces
B_N (P_t) = B_N (P_{t+1})

//Esquema de reemplazamiento una vez hechos los cruces y mutaciones
$P_t = P_{t+1}$
clasificaciones$_t$ = clasificaciones$_{t+1}$

Fin Mientras
//Devolvemos el mejor padre de la ultima generación
Devolver $B_N(P_t)$

3.2.6 AM – (10,0.1)

Este algoritmo memético considera una probabilidad de aplicación de la BL a un 10% *PLS* de la población (0.1 ** N, siendo N=10 el tamaño de la población*) de cromosomas disponibles seleccionados de forma aleatoria. Dado que éste es el único cambio con respecto al AM-(10,0.1) vamos a pasar inmediatamente con el pseudocódigo de éste algoritmo, resaltando en negrita el proceso de selección del conjunto de cromosomas a los que se les aplicará la BL.

AM-(10,0.1)

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
clasificaciones$_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

// Esquema de evolución//
padres ← Torneo Binario(P_t)

// Primera parte de la evolución: cruce de los mejores padres//
Para i=0 hasta cruces hacer
Operador de cruce CA (mejores(pareja 1)): Hijo1

población ← Hijo1
Fin Para (1)

//Segunda parte: completar con los mejores padres de la generación anterior//
Para i= tamaño población actual hasta N
*población ← mejores(i*2)*
Si mejores(i) == B_N
no se ha perdido el mejor padre

// Operación de mutación con la población P_{t+1} //
P_{t+1} ← operación de mutación(P_{t+1})

//Actualizamos el numero de generaciones
generaciones++

//CAMBIO : Aplicación de la BL a un 10% de la población//
Si generaciones es múltiplo de 10 entonces
//Aleatorizamos la población para empezar por un cromosoma distinto
aleatorizar(P_{t+1})
Para i=0 hasta PLS hacer
BL(Train, P_{t+1} (i), 2G vecinos)
evaluaciones += 2G +1

//Evaluamos la nueva población //
clasificaciones$_{t+1}$ ← evaluar población (P_{t+1})
evaluaciones += C

//Comprobación en caso de que el mejor padre se la población P_t no se encuentre
//en la población P_{t+1}
Si B_N (P_t) se ha perdido entonces
W_N (P_{t+1}) = B_N (P_t)

//Actualización del mejor cromosoma en caso de ocurrir
Si T(B_N(P_{t+1})) < T(B_N(P_t)) entonces
T(B_N(P_{t+1})) = T(B(P_t))
Si no entonces
B_N (P_t) = B_N (P_{t+1})

//Esquema de reemplazamiento una vez hechos los cruces y mutaciones
P_t = P_{t+1}
clasificaciones$_t$ = clasificaciones$_{t+1}$

Fin Mientras
//Devolvemos el mejor padre de la ultima generación
Devolver B_N(P_t)

3.2.7 AM – (10,0.1mej)

El último de los algoritmos meméticos tiene un comportamiento similar al anterior, con la única diferencia de que ahora la BL se aplicará a un 10% *PLS* de los mejores cromosomas de la población (0.1 * *N, siendo N=10 el tamaño de la población*). Por lo tanto la forma de abordar este algoritmo es ligeramente distinta: sigue todos los pasos de su homólogo anterior con la sutil diferencia de que ahora aplicará la BL sobre la población P_{t+1} una vez se hayan evaluado los cromosomas, para intentar explotar los $0.1 * N$ mejores individuos de la población. El pseudocódigo y las modificaciones con respecto al anterior algoritmo, así como sus principales operaciones (selección de mejores cromosomas y evolución dentro de la BL) se encuentran comentados y en negrita.

AM-(10,0.1mej)

// Generación y evaluación de la solución inicial //
P_t ← generar soluciones aleatorias(C,G)
clasificaciones$_t$ ← evaluar población (P_t)
evaluaciones += C

// Cuerpo del algoritmo //
Mientras evaluaciones < 15000 hacer

// Esquema de evolución//
padres ← Torneo Binario(P_t)

// Primera parte de la evolución: cruce de los mejores padres//
Para i=0 hasta cruces hacer
Operador de cruce CA (mejores(pareja 1)): Hijo1
población ← Hijo1
Fin Para (1)

//Segunda parte: completar con los mejores padres de la generación anterior//
Para i= tamaño población actual hasta N
*población ← mejores(i*2)*
Si mejores(i) == B_N
no se ha perdido el mejor padre

// Operación de mutación con la población P_{t+1} //
P_{t+1} ← operación de mutación(P_{t+1})

//Actualizamos el numero de generaciones
generaciones++

//CAMBIO : Evaluamos la nueva población antes de aplicar la BL //
clasificaciones$_{t+1}$ ← evaluar población (P_{t+1})
evaluaciones += C

//CAMBIO: ordenamos los cromosomas en orden creciente de tasa de clasificacion//
P_{t+1} ← ordenar por tasa de clasificacion creciente (P_{t+1})

//CAMBIO : Aplicación de la BL a un 10% de los mejores cromosomas de la población//

Si generaciones es múltiplo de 10 entonces

//Al estar ordenados en orden creciente, el mejor será el ultimo, por lo que hay que empezar por el final

Para i=N hasta N-PLS hacer

BL(Train, P_{t+1} (i), 2G vecinos)

evaluaciones += 2G +1

//Comprobación en caso de que el mejor padre se la población P_t no se encuentre //en la población P_{t+1}

Si B_N (P_t) se ha perdido entonces

W_N (P_{t+1}) = B_N (P_t)

//Actualización del mejor cromosoma en caso de ocurrir

Si $T(B_N(P_{t+1}))$ < $T(B_N(P_t))$ entonces

$T(B_N(P_{t+1}))$ = $T(B(P_t))$

Si no entonces

B_N (P_t) = B_N (P_{t+1})

//Esquema de reemplazamiento una vez hechos los cruces y mutaciones

P_t = P_{t+1}

clasificaciones$_t$ = clasificaciones$_{t+1}$

Fin Mientras

//Devolvemos el mejor padre de la ultima generación

Devolver $B_N(P_t)$

4. Algoritmo de Comparación: RELIEF

El algoritmo de comparación utilizado para esta práctica es una solución de tipo Greedy al problema del APC llamado RELIEF. El funcionamiento del algoritmo se basa en partir de un vector de pesos *W* inicializados a cero de tal forma que para cada ejemplo considerado de la partición de *Train* se calcule el amigo y el enemigo más cercano según la partición de *Test* considerada. El objetivo del algoritmo, tal y como se expresa en el guión de práctica, es doble: primero aumentará los pesos de aquellas características que separen ejemplos que son enemigos entre si, mientras que reduce la distancia entre las características que separan ejemplos que son de la misma clase. Los operadores de calcular el amigo y el enemigo más cercano son sencillos y expresaremos el pseudocódigo del primero (amigo mas cercano) y comentaremos la diferencia en el propio pseudocódigo (en negrita) con su contrario, el del enemigo más cercano. Acto seguido definiremos el funcionamiento del RELIEF y pasaremos a tratar su pseudocódigo.

4.1 Calcular amigo más cercano

Para este operador hacemos uso del operador *calcular distancia Euclídea* con una ligera modificación al original, y ésta es que guardamos en un vector la diferencia entre cada una de las características de los ejemplos que estamos comparando, de forma que se agilice el cálculo del amigo más cercano, devolviendo no solo a distancia Euclídea total sino tambien la diferencia componente a componente. El pseudocódigo lo vemos a continuación, previo al del operador en cuestión.

Calcular distancia Euclídea RELIEF

Para i=0 hasta G hacer
Diferencia $\leftarrow P1_i - P2_i$
Distancia $\leftarrow W_i * (P1_i - P2_i)^2$
Distancia $\leftarrow \sqrt{Distancia}$
Devolver Distancia, Diferencia

Una vez conocido este operador vamos a expresar el operador de calcular el amigo más cercano (resaltando en negrita la diferencia con su contrario). Hay que recordar que partimos de un ejemplo de $Train_i$ y una partición completa de *Test*.

Calcular amigo mas cercano

$D_{min} = 100$, $distancia_{amigo}$
$C_{min} = clase(Train_i)$
Para cada $Test_j$ *en Test hacer*
Distancia, Diferencia $\leftarrow$ *calcular distancia Euclidea RELIEF* ($Train_i$, $Test_j$)

//En el calculo del enemigo, la condición cambia a clase($Test_j$ ***) !=*** C_{min} ***//***
Si Distancia < D_{min} *&& clase(*$Test_j$*) ==* C_{min} *entonces*
$D_{min} = Distancia$
$distancia_{amigo} = Diferencia$
Fin Para (1)
Devolver $distancia_{amigo}$

Ya estamos listos para formular el pseudocódigo del RELIEF.

Algoritmo RELIEF

Para cada Tr_i en Train hacer
$distancia_{amigo}$ ← calcular amigo mas cercano (Tr_i, Test)
$distancia_{enemigo}$ ← calcular enemigo mas cercano (Tr_i, Test)

Para cada W_j en W hacer
$W_j = W_j + distancia_{enemigo}(j) - distancia_{amigo}(j)$
Fin Para cada (2)
Fin Para cada (1)

W_{max}= maximo(W)

Para cada W_j en W hacer
Si W_j <0 entonces
W_j =0
Si no
$W_j = W_j / W_{max}$
Fin Para cada

Devolver W

Hasta aquí todo lo referente al pseudocódigo de los algoritmos. Hemos abordado todos los aspectos en cuanto a operadores comunes a todos los algoritmos, los propios algoritmos en si y el algoritmo de comparación. Cabe destacar que se ha seguido un esquema parecido al que se utiliza en la implementación de código fuente: primero hemos definido los operadores comunes con unos nombres específicos que después hemos utilizado dentro del pseudocódigo de los algoritmos como si fuesen “llamadas” a estas funciones. Hemos intentando utilizar siempre los mismos nombres para los distintos datos que se utilizan dentro de los operadores y de los algoritmos para que sea coherente y se pueda hacer un seguimiento a lo largo de todos los pseudocódigos de forma que se entienda cada paso, cada operación y cada algoritmo implementado.

5. Procedimiento considerado para desarrollar la práctica

El desarrollo de esta práctica ha seguido un procedimiento parecido al que se ha explicado en los apartados anteriores. Todo el código fuente ha sido implementado desde cero por mi persona, si bien para implementar los distintos operadores y algoritmos me he basado tanto en los pseudocódigos que aparecen en las transparencias de los seminarios de prácticas como en las de teoría. A continuación vamos a hacer un repaso del proceso completo en forma de **manual de usuario** de forma que el profesor de prácticas pueda replicar el propio proceso.

Manual de Usuario

1. Diseño de la clase "ConjuntoDatos": la clase contiene los datos a utilizar y las cabeceras de los operadores y algoritmos que se implementarán más adelante.

2. Implementación de operadores comunes: lectura de ficheros, normalización de datos, cálculo de la distancia Euclídea, la tasa de clasificación, amigo y enemigo más cercano, generación de soluciones aleatorias y la generación de vecino/mutación normal.

3. Implementación de la función objetivo 1NN

4. Implementación del algoritmo de comparación RELIEF: utilizando los operadores de amigo y enemigo más cercano.

5. Implementación de la BL: utilizando la función objetivo y la generación de vecinos por mutación normal.

6. Implementación de los operadores comunes de los A.Gs: los operadores de cruce BLX-α y CA, así como el Torneo Binario y la generación de un gen en un intervalo determinado.

7. Implementación de los A.Gs: AGG-BLX, AGG-CA, AGE-BLX y AGE-CA.

8. Implementación de los A.Ms: AM-(10,1.0), AM-(10,0.1) y AM-(10,0.1mej)

6. Experimentos y análisis de resultados

Para la realización de los experimentos vamos a considerar tres bases de datos: sonar.arff, spambase-460.arff y wdbc.arff. Para cada una de estas bases de datos vamos a aplicar los 10 algoritmos implementados y para cada uno de ellos vamos a describir los valores de los parámetros que requieren cada uno de éstos.

Comenzamos estableciendo el valor de la semilla aleatoria para todos los algoritmos al valor de los dígitos de mi DNI, esto es, **1680851**.

6.1 Clasificador 1-NN

Las 5 particiones de los datos al 50% con sus resultados de tasa de clasificación y tiempos para cada base de datos se pueden observar en la siguiente tabla. Las comparaciones realizadas se han efectuado sobre las 5 particiones y para el caso en particular del Clasificador 1-NN (sin formar parte de ningún otro algoritmo) los mejores resultados en cuanto a clasificación media se han obtenido para la base de datos **wdbc**.

Tabla 6.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	77,8846	0,00094548	96,4789	0,00368302	80,0000	0,00422913
Partición 1-2	86,5385	0,00094147	96,1404	0,00286969	80,8696	0,00321031
Partición 2-1	86,5385	0,00104565	95,4225	0,00330130	84,7826	0,00308959
Partición 2-2	86,5385	0,00091998	95,7895	0,00256099	81,3043	0,00344980
Partición 3-1	77,8846	0,00092807	96,4789	0,00259686	79,5652	0,00263893
Partición 3-2	86,5385	0,00091954	97,1930	0,00254455	84,3478	0,00257956
Partición 4-1	84,6154	0,00075856	94,0141	0,00261790	83,0435	0,00258493
Partición 4-2	81,7308	0,00069146	94,7368	0,00323324	81,7391	0,00257482
Partición 5-1	82,6923	0,00074918	95,4225	0,00232269	77,3913	0,00263433
Partición 5-2	88,4615	0,00076211	96,4912	0,00276714	80,4348	0,00260076
Media	83,9423	0,00086615	95,8168	0,00284974	81,3478	0,00295922

Si analizamos detenidamente los resultados en cuanto a la diferencia existente entre el valor máximo y mínimo para cada una de las bases de datos obtenemos lo siguiente. Este será el estándar que seguiremos para comparar los distintos resultados y algoritmos a lo largo de todo el análisis.

1NN	**Sonar**	**Wdbc**	**Spambase**
Media	83,94232	95,81678	81,34782
Máximo	88,4615	97,193	84,7826
Mínimo	77,8846	94,0141	77,3913
Diferencia	10,5769	3,1789	7,3913

La base de datos de Sonar contiene muy pocos ejemplos (208) y las particiones que se generan de estos hacen ver que los datos son muy disjuntos, por lo que podemos apreciar una gran diferencia

entre los valores máximo y mínimo. Algo similar ocurre con la base de datos de Spam, que esta vez cuenta con 460 ejemplos y en ésta la diferencia es menor que con la del Sonar, dando a entender que cuantos más ejemplos tengamos mejor será el ajuste que se puede realizar, que es el último caso, el de la base de datos de Wdbc, muchos ejemplos y esta vez particiones muy comunes entre si (apenas un 3,18%) de diferencia entre la mejor y la peor tasa de clasificación. Estos resultados parecen indicar que con ésta base de datos se realizarán mejores ajustes a lo largo de los distintos algoritmos, hecho que iremos comprobando conforme analicemos los resultados de los demás algoritmos.

6.2 Algoritmo de comparación: RELIEF

El algoritmo RELIEF que aplica una estrategia de tipo Greedy será nuestra cota de comparación con los resultados de los demás algoritmos. Pasamos a analizar los resultados que ha obtenido para cada base de datos:

Tabla 6.2: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	76,9231	0,00463082	95,0704	0,01480340	84,7826	0,01683810
Partición 1-2	86,5385	0,00355257	96,1404	0,01117090	80,8696	0,01242000
Partición 2-1	83,6538	0,00385246	95,7746	0,01369880	90,0000	0,01604360
Partición 2-2	85,5769	0,00335361	94,3860	0,01121310	82,6087	0,01249950
Partición 3-1	77,8846	0,00320369	96,4789	0,01095840	84,3478	0,01437450
Partición 3-2	89,4231	0,00290834	95,7895	0,01182040	85,2174	0,01349070
Partición 4-1	87,5000	0,00272594	94,0141	0,01420610	89,1304	0,01240270
Partición 4-2	83,6538	0,00264948	94,0351	0,01261930	84,3478	0,01238950
Partición 5-1	78,8462	0,00269713	94,7183	0,01216120	83,9130	0,01233150
Partición 5-2	91,3462	0,00265925	96,1404	0,01124510	85,6522	0,01242360
Media	84,1346	0,00322333	95,2548	0,01238967	85,0870	0,01352137

Volvemos a hacer uso de nuestra tabla de máximos, mínimos y medias, que resume la información a grandes rasgos y nos permite hacer mejores comparaciones.

RELIEF	**Sonar**	**Wdbc**	**Spambase**
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304

1NN	**Sonar**	**Wdbc**	**Spambase**
Media	83,94232	95,81678	81,34782
Máximo	88,4615	97,193	84,7826
Mínimo	77,8846	94,0141	77,3913
Diferencia	10,5769	3,1789	7,3913

Nuevamente es muy evidente como influye el tamaño del conjunto de ejemplos considerados y de la distribución de los datos en las tres bases de datos sobre las que trabajamos. Si bien es cierto que

con respecto al algoritmo 1NN, el RELIEF muestra ligeramente mejores resultados medios, pero la diferencia entre los resultados máximos y mínimos en los conjuntos de Sonar y Spam son mucho mayores, lo que puede indicar que la estrategia Greedy funciona mejor con conjuntos de datos más pequeños (aumenta la media del Sonar y del Spam -éste último notablemente-) que con conjuntos de datos más grandes (Wdbc incluso empeora por algunas décimas, aunque la diferencia entre cotas superior e inferior disminuya). Este es nuestro algoritmo RELIEF, el que servirá como base para comparar todos los algoritmos, un algoritmo que tiene un comportamiento bastante predecible: trabaja bien con pocos datos y con conjuntos distintos entre sí, dado que al considerar el cálculo del amigo y enemigo más cercano se adapta mejor a datos distintos. El siguiente paso consistirá en comparar los resultados obtenidos por este algoritmo con los demás algoritmos implementados, comenzando por la BL.

6.3 BL del primer mejor vs RELIEF

A continuación se muestran los resultados del algoritmo de BL para las 3 bases de datos consideradas. Añadimos también las tablas de máximos, mínimos y medias tanto de la BL como del algoritmo de comparación RELIEF.

Tabla 6.3: Resultados obtenidos por el algoritmo BL en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	80,7692	0,73130100	95,7746	1,39231000	82,6087	3,25134000
Partición 1-2	85,5769	0,70056900	95,7895	1,39969000	79,1304	3,54908000
Partición 2-1	81,7308	0,70984600	94,7183	1,41976000	81,7391	3,42074000
Partición 2-2	87,5000	0,72702300	96,1404	1,39329000	80,0000	3,33670000
Partición 3-1	81,7308	0,74601700	95,4225	1,44916000	81,7391	3,24874000
Partición 3-2	80,7692	0,71102400	95,7895	1,46643000	83,9130	3,36765000
Partición 4-1	81,7308	0,69384900	94,0141	1,35819000	89,5652	3,23374000
Partición 4-2	81,7308	0,69841400	93,3333	1,40270000	83,0435	3,10330000
Partición 5-1	82,6923	0,69762700	95,4225	1,37131000	86,0870	3,17286000
Partición 5-2	91,3462	0,74643900	95,0877	1,44652000	83,9130	3,03815000
Media	83,5577	0,71621090	95,1492	1,40993600	83,1739	3,27223000

BL	**Sonar**	**Wdbc**	**Spambase**
Media	83,5577	95,14924	83,1739
Máximo	91,3462	96,1404	89,5652
Mínimo	80,7692	93,3333	79,1304
Diferencia	10,577	2,8071	10,4348

RELIEF	**Sonar**	**Wdbc**	**Spambase**
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304

El algoritmo de BL suele proporcionar muy buenos resultados cuando se aplica sobre una gran cantidad de ejemplos de **buena calidad**, ya que explota los vecindarios de soluciones parciales y de

una calidad determinada. En nuestros experimentos la BL comienza con un vector de pesos generado de forma aleatoria en el intervalo [0,1] por lo tanto la bondad de la solución final dependerá naturalmente de los valores iniciales del vector *W* y de la cantidad de vecinos que le permitamos explorar a nuestro algoritmo, en este caso un valor de 20 por el número de características. Los resultados obtenidos tras ejecutar el algoritmo sobre las 5 particiones muestran que de media ha reducido un poco su calidad frente al RELIEF aunque para el conjunto de Sonar ha sido capaz de encontrar una solución mínima mejor que la del RELIEF; a grandes rasgos podemos decir que los resultados son los esperados por las siguientes razones:
1. El conjunto de datos no es lo suficientemente grande como para hacer frente a una estrategia de tipo voraz.

2. La solución inicial de la que parte la BL puede tener una calidad baja (dada la componente aleatoria del experimento) y el número de vecinos generados antes de concluir el algoritmo puede ser insuficiente, razones que llevan a desaprovechar la cualidad de explotación del algoritmo.

3. La desviación típica considerada (σ=0.3) puede no ser suficiente a la hora de generar un vecino por mutación normal, razón que afecta de lleno sobre el peso de la característica i-ésima y que puede que no lleve a una buena solución.

Además, teniendo en cuenta que el tiempo que tarda en ejecutarse una BL es bastante superior a una ejecución del algoritmo RELIEF no merece la pena realmente dado que el conjunto de datos no es lo suficientemente grande. Como conclusión cabe destacar que es mejor aplicar la BL sobre una solución de cierta calidad para forzarla a mejorar, si bien no es capaz de hacer frente, en término medio, al algoritmo RELIEF sobre los conjuntos de datos considerados en los experimentos.

6.4 AGG-BLX vs RELIEF

El primero de nuestros algoritmos genéticos es del tipo generacional (la población de la generación *t+1* sustituye de forma completa a la de la generación *t*) y el operador de cruce que se ha empleado es el *BLX-α*, donde se calcula un intervalo (del que saldrán los dos hijos) con doble propósito: una parte de explotación y otra de exploración. Para este algoritmo consideramos una población de 30 cromosomas, donde cada cromosoma representa un vector de pesos de un número de componentes igual al número de características consideradas para cada ejemplo, generado de forma aleatoria de la misma forma que lo hacíamos para la BL. Además, consideramos un valor de $\alpha=0.3$ y una probabilidad de cruce del *0.7* (se cruzarán el 70% de las parejas) y una probabilidad de mutación de un gen del *0,001* (al redondear al alza, conseguimos que por cada generación se mute al menos un gen). Una vez conocidos los parámetros que definen a nuestro algoritmo vamos a pasar directamente con las comparaciones frente al RELIEF.

Tabla 6.4: Resultados obtenidos por el algoritmo AGG-BLX en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	78,8462	8,41974000	96,8310	33,63830	80,0000	43,540500
Partición 1-2	85,5769	8,49414000	95,7895	34,19510	80,0000	41,826300
Partición 2-1	83,6538	8,39112000	95,7746	34,77720	85,2174	39,866500
Partición 2-2	82,6923	8,48757000	96,4912	35,55260	82,6087	39,882900
Partición 3-1	78,8462	8,37390000	94,3662	33,80120	80,4348	39,272300
Partición 3-2	84,6154	9,04757000	95,0877	33,90270	84,3478	39,996500
Partición 4-1	81,7308	8,82925000	94,3662	34,43090	82,6087	40,262800
Partición 4-2	81,7308	8,34568000	95,4386	34,38610	85,2174	40,103800
Partición 5-1	83,6538	8,40719000	94,3662	33,76100	81,7391	39,833200
Partición 5-2	87,5000	8,45547000	96,1404	33,86770	83,0435	39,451000
Media	82,8846	8,52516300	95,4652	34,231280	82,5217	40,403580

AGG-BLX	Sonar	Wdbc	Spambase
Media	82,88462	95,46516	82,52174
Máximo	87,5	96,831	85,2174
Mínimo	78,8462	94,3662	80
Diferencia	8,6538	2,4648	5,2174
Tiempo	8,52516300	34,231280	40,40358

RELIEF	Sonar	Wdbc	Spambase
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304
Tiempo	0,00322333	0,01238967	0,01352137

Empezamos con los valores medios: RELIEF presenta mejores medias para los conjuntos más pequeños y distintos entre si (Sonar y Spam) pero AGG-BLX ha conseguido una pequeña mejora en el conjunto más grande y con ejemplos parecidos entre sí, dando a entender que dentro de una población con soluciones de cierta calidad que son evaluadas para un conjunto parecido en sus distintas particiones y al combinarlas, somos capaces de “mejorar la especie” y conseguir mejores resultados también en valores de máximo. También cabe destacar que las “peores soluciones” del AGG-BLX son de media ligeramente mejores que las que llega a calcular el RELIEF, entrando otra vez por el mismo camino de antes: un algoritmo genético es capaz de gestionar de mejor forma una población determinada haciendo que tras los cruces las mejores soluciones vayan arrojando resultados parecidos y por eso los valores de diferencia son más pequeños, porque las soluciones tienden a llegar a una cierta calidad, aunque por el número de evaluaciones que se realizan no sean capaces de alcanzar las mejores cotas de calidad.

El punto de partida del algoritmo genético condiciona mucho los resultados obtenidos dado que parte de soluciones aleatorias que pueden tener muy baja calidad y las mismas pueden necesitar muchas generaciones para mejorar. La probabilidad de cruce y mutación pueden influir en estos resultados de igual forma, aunque en menor medida. Por último teniendo en cuenta los tiempos de ejecución no parece muy adecuado utilizar esta técnica con un conjunto de ejemplos tan reducido y con soluciones iniciales aleatorias frente al RELIEF, aunque si partiésemos de mejores soluciones y con una probabilidad de cruce ligeramente mayor, es muy probable que los resultados obtenidos sean mucho mejores y si merezca la pena sacrificar un poco de tiempo para obtener soluciones de la más alta calidad posible.

6.5 AGG-CA vs RELIEF

El segundo de nuestros algoritmos genéticos implementa el cruce aritmético (CA) con todos los parámetros iguales frente al anterior, por lo que pasamos directamente a evaluar y analizar los resultados obtenidos comparando los dos genéticos generacionales con el RELIEF.

AGG-CA	Sonar	Wdbc	Spambase
Media	82,8846	95,60637	82,65218
Máximo	88,4615	96,8421	86,087
Mínimo	76,9231	93,6842	80,4348
Diferencia	11,5384	3,1579	5,6522
Tiempo	8,59355200	34,214950	40,34044
AGG-BLX	**Sonar**	**Wdbc**	**Spambase**
Media	82,88462	95,46516	82,52174
Máximo	87,5	96,831	85,2174
Mínimo	78,8462	94,3662	80
Diferencia	8,6538	2,4648	5,2174
Tiempo	8,52516300	34,231280	40,40358

RELIEF	Sonar	Wdbc	Spambase
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304
Tiempo	0,00322333	0,01238967	0,01352137

Tabla 6.5: Resultados obtenidos por el algoritmo AGG-CA en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	77,8846	8,87505000	95,7746	35,399800	80,8696	42,710400
Partición 1-2	76,9231	8,82296000	95,4386	33,997000	81,7391	39,631300
Partición 2-1	83,6538	9,19147000	95,0704	33,764500	84,7826	39,442400
Partición 2-2	88,4615	8,32139000	96,1404	34,547500	80,4348	39,903800
Partición 3-1	76,9231	8,36684000	96,1268	35,249500	82,1739	39,816200
Partición 3-2	83,6538	8,38797000	96,8421	33,479200	83,0435	39,622900
Partición 4-1	86,5385	8,62573000	95,7746	33,165600	86,0870	39,278800
Partición 4-2	80,7692	8,42871000	93,6842	33,641900	82,6087	40,011600
Partición 5-1	85,5769	8,48581000	95,4225	33,458800	82,1739	42,696500
Partición 5-2	88,4615	8,42959000	95,7895	35,445700	82,6087	40,290500
Media	82,8846	8,59355200	95,6064	34,214950	82,6522	40,340440

Los resultados obtenidos son muy similares a los que presenta el AGG-BLX donde la única mejora notable que se obtiene frente a éste es el aumento en los valores máximos (nunca superando los del RELIEF) donde la única conclusión que podemos obtener es que le ocurre lo mismo que a su hermano generacional: el operador de cruce no ha condicionado mejoras notables dado que el punto de partida sabemos que puede llegar a ser el mismo dada la aleatoriedad de la generación de las soluciones, sin embargo al presentar mejores resultados tanto en tiempo de ejecución como tasa de clasificación media, será el operador de cruce seleccionado para los algoritmos meméticos.

Llegamos a la misma conclusión que con el AGG-BLX: este algoritmo no es capaz de hacer frente de forma notable (tiempo de ejecución frente a calidad de soluciones obtenidas) al RELIEF por las mismas razones que su hermano generacional: soluciones iniciales de (posible) baja calidad y conjunto de datos muy pequeño para conseguir mejorar lo suficiente la población de soluciones.

Hemos terminado con el análisis de los AGGs, ahora el siguiente paso será analizar los resultados obtenidos para los algoritmos genéticos estacionarios (AGE-BLX y AGE-CA) frente al RELIEF siguiendo las mismas pautas que hemos seguido hasta ahora.

6.6 AGE-BLX y AGE-CA vs RELIEF

Las tablas de resultados de la ejecución de los dos algoritmos genéticos estacionarios se pueden apreciar a continuación. Justo después analizaremos los resultados con nuestras ya conocidas tablas de máximos, mínimos, medias y tiempos. Los parámetros utilizados para la realización de estos experimentos son los siguientes: probabilidad de cruce =1, probabilidad de mutación =0,001, $\alpha=0.3$.

Tabla 6.6: Resultados obtenidos por el algoritmo AGE-BLX en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	75,0000	8,62090000	95,7746	34,087300	79,1304	39,647400
Partición 1-2	83,6538	8,38082000	96,1404	33,941000	79,1304	39,631600
Partición 2-1	83,6538	8,44260000	95,0704	33,662300	82,6087	39,616100
Partición 2-2	84,6154	8,39822000	95,0877	33,999500	80,4348	39,726200
Partición 3-1	75,9615	8,48062000	96,1268	33,819300	81,3043	39,635100
Partición 3-2	85,5769	8,53754000	97,8947	33,948400	83,0435	39,877100
Partición 4-1	81,7308	8,37636000	95,0704	33,715700	81,7391	39,688200
Partición 4-2	82,6923	8,60657000	93,3333	34,165200	83,4783	39,799400
Partición 5-1	82,6923	8,58618000	93,6620	33,915900	84,3478	39,796800
Partición 5-2	87,5000	8,54761000	96,1404	37,324800	83,9130	39,656800
Media	82,3077	8,49774200	95,4301	34,257940	81,9130	39,707470

Tabla 6.7: Resultados obtenidos por el algoritmo AGE-CA en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	77,8846	9,51809000	95,7746	36,169900	79,1304	42,282700
Partición 1-2	75,9615	9,64244000	95,4386	37,566100	77,8261	40,061700
Partición 2-1	86,5385	9,22493000	95,4225	35,715500	86,5217	39,737600
Partición 2-2	87,5000	8,59058000	95,4386	34,583900	83,0435	40,082700
Partición 3-1	84,6154	8,55318000	96,4789	34,438300	82,1739	40,138800
Partición 3-2	78,8462	8,75988000	97,1930	34,570800	84,7826	39,991600
Partición 4-1	88,4615	8,60534000	96,4789	37,485500	83,4783	40,213800
Partición 4-2	86,5385	8,40715000	94,7368	34,523000	84,7826	41,052300
Partición 5-1	81,7308	8,38949000	94,0141	37,220300	78,6956	39,942300
Partición 5-2	84,6154	8,89087000	95,4386	37,653000	84,7826	40,216800
Media	87,5000	8,85819500	95,6415	35,992630	82,5217	40,372030

AGE-BLX	Sonar	Wdbc	Spambase
Media	82,30768	95,43007	81,91303
Máximo	87,5	97,8947	84,3478
Mínimo	75,9615	93,3333	79,1304
Diferencia	11,5385	4,5614	5,2174
Tiempo	8,49774200	34,257940	39,70747

AGE-CA	Sonar	Wdbc	Spambase
Media	83,26924	95,64146	82,52173
Máximo	84,6154	97,193	86,5217
Mínimo	75,9615	94,0141	77,8261
Diferencia	8,6539	3,1789	8,6956
Tiempo	8,85819500	35,992630	40,37203

RELIEF	Sonar	Wdbc	Spambase
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304
Tiempo	0,00322333	0,01238967	0,01352137

Los dos algoritmos genéticos estacionarios presentan un comportamiento diferente al de sus contrarios generacionales al sustituir únicamente a los dos peores padres de la población en caso de que los hijos generados sean mejores. En cuanto a términos de media para el conjunto más grande de todos (Wdbc) el AGE-CA muestra resultados notablemente mejores que los del RELIEF o su hermano AGE-BLX, dando a entender que el cruce aritmético produce (de media) resultados ligeramente mejores al sustituir únicamente dos de los peores padres de la población.

En cuanto a términos de máximo para el Wdbc el que utiliza el operador BLX explota mejor las soluciones llegando a conseguir casi un 98% de acierto (máximo), que dentro de una cota de error del 2% produce muy buenos resultados, por lo tanto puede preferirse en este caso sacrificar un poco de tiempo de ejecución para llegar a mejores resultados máximos dentro de una conjunto de ejemplos lo suficientemente grande, nunca aplicándose a conjuntos más pequeños, que es donde

vemos que el RELIEF despunta con resultados medios bastante aceptables para el conjunto de datos.

El penúltimo paso de nuestro análisis pasa por comparar los distintos algoritmos meméticos que han sido implementados en base al AGG-CA, dado que fue éste quien arrojo mejores resultados en cuanto a tiempo de ejecución y tasa de clasificación.

6.7 AM -(10,1.0), AM -(10,0.1) y AM -(10,0.1mej) vs RELIEF

Los tres tipos de algoritmos meméticos han sido implementados basándonos en el AGG-CA dado que ha sido el que mejores resultados ha proporcionado. El primero, AM-(10,1.0) aplicará la búsqueda local (BL) a toda la población cada 10 generaciones. El segundo hará lo propio pero para un 10% de la población elegida de forma aleatoria y el último aplicará también cada 10 generaciones la BL pero esta vez a un 10% de los mejores padres dentro de una población de 10 cromosomas para todos los algoritmos meméticos. A continuación detallamos los resultados de los 3 algoritmos con el formato de tablas y dejamos la comparación de éstos con el RELIEF como último punto de este apartado.

Tabla 6.8: Resultados obtenidos por el algoritmo AM-(10,1.0) en el problema del APC

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
Partición 1-1	73,0769	9,66314000	95,0704	34,961600	78,2609	42,072000
Partición 1-2	84,6154	9,44894000	94,3860	35,732800	81,3043	41,187600
Partición 2-1	85,5769	9,49865000	95,7746	35,523200	86,9565	41,597400
Partición 2-2	87,5000	9,59649000	94,3860	36,936000	81,7391	42,416000
Partición 3-1	82,6923	9,27892000	95,7746	34,795800	83,9130	41,446300
Partición 3-2	74,0385	9,69462000	96,1404	34,882600	85,2174	44,829700
Partición 4-1	85,5769	9,97225000	95,4225	39,737100	83,0435	43,368400
Partición 4-2	82,6923	9,63832000	93,6842	35,723900	84,7826	40,972800
Partición 5-1	85,5769	10,18780	93,6620	34,705700	77,3913	40,809700
Partición 5-2	89,4231	9,34356000	95,4386	36,471400	83,4783	40,796400
Media	83,0769	9,63226900	94,9739	35,947010	82,6087	41,949630

Tabla 6.9: Resultados obtenidos por el algoritmo AM-(10,0.1) en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	75,9615	10,64680	96,4789	36,283600	80,0000	44,787500
Partición 1-2	83,6538	10,49600	95,0877	36,625300	79,5652	43,194800
Partición 2-1	86,5385	10,14960	95,4225	35,3652s	82,6087	44,578300
Partición 2-2	90,3846	9,72059000	94,7368	37,469300	83,9130	45,995600
Partición 3-1	75,9615	9,56902000	96,1268	35,870000	77,8261	44,075600
Partición 3-2	90,3846	9,59974000	95,0877	37,395700	81,3043	45,501200
Partición 4-1	88,4615	9,70811000	94,3662	36,562500	85,2174	44,546400
Partición 4-2	83,6538	9,74926000	93,6842	35,617700	84,3478	43,354900
Partición 5-1	81,7308	9,88773000	95,0704	36,229900	77,3913	43,312700
Partición 5-2	88,4615	9,57254000	96,1404	35,408000	82,1739	48,590500
Media	84,5192	9,90993900	95,2202	36,384667	81,4348	44,793750

Tabla 6.10: Resultados obtenidos por el algoritmo AM-(10,0.1mej) en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	76,9231	8,70349000	96,1268	36,051800	84,7826	40,730400
Partición 1-2	82,6923	8,48185000	95,0877	36,376000	83,0435	40,636200
Partición 2-1	87,5000	8,57769000	95,0704	36,099700	90,8696	40,937200
Partición 2-2	85,5769	8,57179000	94,0351	36,192600	84,7826	40,508000
Partición 3-1	90,3846	8,55971000	95,4225	36,668900	81,7391	40,601300
Partición 3-2	89,4231	8,50661000	95,4386	36,040600	83,4783	40,996900
Partición 4-1	80,7692	8,49221000	94,7183	36,275900	79,5652	40,398700
Partición 4-2	82,6923	8,75672000	94,3860	36,360500	86,0870	40,804800
Partición 5-1	83,6538	8,55912000	92,9577	36,244300	75,2174	40,423400
Partición 5-2	79,8077	8,62821000	94,0351	36,163100	82,6087	41,316100
Media	83,9423	8,58374000	94,7278	36,247340	83,2174	40,735300

AM-(10,1.0)	Sonar	Wdbc	Spambase
Media	83,07692	94,97393	82,60869
Máximo	89,4231	96,1404	86,9565
Mínimo	74,0385	93,662	77,3913
Diferencia	15,3846	2,4784	9,5652
Tiempo	9,63226900	35,947010	41,94963

AM-(10,0.1mej)	Sonar	Wdbc	Spambase
Media	83,9423	94,72782	83,2174
Máximo	90,3846	96,1268	90,8696
Mínimo	76,9231	92,9577	75,2174
Diferencia	13,4615	3,1691	15,6522
Tiempo	8,57879889	36,247340	40,7353

AM-(10,0.1)	Sonar	Wdbc	Spambase
Media	84,51921	95,22016	81,43477
Máximo	90,3846	96,4789	85,2174
Mínimo	75,9615	93,6842	77,3913
Diferencia	14,4231	2,7947	7,8261
Tiempo	9,90993900	36,384667	44,793750

RELIEF	Sonar	Wdbc	Spambase
Media	84,13462	95,25477	85,08695
Máximo	91,3462	96,4789	90
Mínimo	77,8846	94,0141	80,8696
Diferencia	13,4616	2,4648	9,1304
Tiempo	0,00322333	0,01238967	0,01352137

Como primer aspecto a considerar en nuestro análisis cabe destacar el hecho de que los tres A.Ms producen resultados muy parecidos entre si, siendo el AM-(10,0.1) el que mejores resultados globales presenta, llegando a superar en una de las bases de datos al RELIEF, mientras que de media para los demás sigue ofreciendo peores soluciones (dado el punto de partida aleatorio). El AM-(10,0.1mej) ha conseguido arroja unos resultados francamente buenos al aplicar la BL sobre el mejor de los padres, pero seguimos con la misma tara de antes: si las soluciones no son de cierta calidad, tardará mucho en converger a buenas soluciones. Por eso el AM-(10,0.1) ofrece mejores soluciones globales, al disponer de aleatoriedad para la selección del 10% de los cromosomas a evaluar. En cuanto a comparación con el RELIEF poco mas hay que decir, no somos capaces de mejorar de forma notable los resultados dado que el conjunto de ejemplos es reducido. Hasta aquí llega nuestro análisis comparativo de los resultados de todos los algoritmos, teniendo una última tabla en la que resaltaremos en azul los mejores resultados de los algoritmos frente a los del RELIEF, aunque en la mayoría de los casos no se lleguen a superar los que ofrece el algoritmo Greedy (resaltado en gris como algoritmo de referencia)

	Sonar		**Wdbc**		**Spambase**	
	%_clas	**T**	***%_clas***	**T**	***%_clas***	**T**
1-NN	83,9423	0,0009	95,8168	0,0028	81,3478	0,0030
RELIEF	84,1346	0,0032	95,2548	0,0124	85,0870	0,0135
BL	83,5577	0,7162	95,1492	1,4099	83,1739	3,2722
AGG-BLX	82,8846	8,5252	95,4652	34,2313	82,5217	40,4036
AGG-CA	82,8846	8,5936	95,6064	34,2150	82,6522	40,3404
AGE-BLX	82,3077	8,4977	95,4301	34,2579	81,9130	39,7075
AGE-CA	83,2692	8,8582	95,6415	35,9926	82,5217	40,3720
AM-(10,1.0)	83,0769	9,6323	94,9739	35,9470	82,6087	41,9496
AM-(10,0.1)	84,5192	9,9099	95,2202	36,3847	81,4348	44,7938
AM-(10,0.1mej)	83,9423	8,5837	94,7278	36,2473	83,2174	40,7353

El AM-(10,0.1) proporciona el mejor resultado medio para la base de datos de Sonar, el AGE-CA hace lo propio para el Wdbc y el algoritmo de comparación RELIEF gana frente a cualquier otro en el Spambase. El genético estacionario se comporta mejor (de media) frente a conjuntos de datos más grandes y con una distribución de los mismos más uniforme. El RELIEF obtiene mejores resultados en un conjunto de datos que es muy difícil de aprender para los demás algoritmos arrojando muy buenos resultados medios, mientras que el memético ha conseguido ajustar mejor que ningún un conjunto reducido de ejemplos (Sonar) a pesar de que el punto de partida no es el más adecuado. Hasta aquí nuestro análisis sobre los algoritmos implementados en la práctica. Hemos intentado aplicar los conocimientos de teoría sobre los algoritmos al análisis realizado aunque en algunas ocasiones los resultados recogidos no se ajusten al 100% al comportamiento esperado por el algoritmo, si bien pueden ser considerados correctos al presentar todos una tasa de clasificación media por encima del 80% para todos los conjuntos de datos.

7. Referencias bibliográficas

La función generarGenAleatorio se ha basado en la función que se encuentra en una entrada en StackOverflow: http://stackoverflow.com/questions/2704521/generate-random-double-numbers-in-c

El resto del contenido es de mi propia autoría y lo he construido en base a los contenedores y funciones de C++ que se encuentran en la página oficial del lenguaje, además de basarme en los seminarios y transparencias de teoría.