Universidad de Granada
E.T.S.I Informática y Telecomunicaciones
Grado en Ingeniería Informática (GII)

Practica 2: Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial para el Problema de Aprendizaje de Pesos en Características

Curso 2016-2017

Problema de Aprendizaje de Pesos en Características

Algoritmos considerados: 1-NN, RELIEF, ES, ILS, DE_Rand, DE_CurrentToBest, LS, AGG-BLX y AM-(10,0.1Mej)

Grupo 2: Viernes 17:30 - 19:30

Arthur Rodríguez Nesterenko - DNI Y1680851W

5 de junio de 2017

Índice

1. Descripción del problema

El objetivo principal de esta práctica trata sobre la utilización de 3 algoritmos principales para la resolución del problema del aprendizaje de pesos en características (APC), en nuestro caso, Enfriamiento Simulado (ES), Búsqueda Local Reiterada (ILS) y Evolución Diferencial (DE). Estos algoritmos serán comparados con algunos de los mejores algoritmos implementados en la práctica anterior con el objetivo de obtener una conclusión acerca de cual ofrece mejores resultados.

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. El mecanismo que emplearemos para resolver estas tareas es el llamado 5-fold cross validation (5fcv), en el que el conjunto de datos se divide en 5 particiones disjuntas al 20 %. Utilizaremos un 80 % de los datos disponibles (4 de 5 particiones) para entrenar nuestro clasificador y posteriormente validaremos el mismo con el 20 % restante de los datos, teniendo un total de 5 particiones posibles al 80-20 %.

El clasificador utilizado será el K-NN (con K=1), que básicamente le asigna a un objeto e' la clasificación *cmin* 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 $W = w1, \ldots, wn$ 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 – algoritmo de comparación), búsqueda local (BL) y búsqueda con metaheurísticas donde entrarán en acción las principales técnicas: Enfriamiento Simulado, Búsqueda Local Reiterada (ILS) y los dos modelos de Evolución Diferencial (DE).

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 algoritmos 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: obviando la lectura de los ficheros de datos (dado que no cambia con respecto a la práctica anterior) pasamos directamente al encapsulamiento de la información y la preparación de la misma para la ejecución de los algoritmos.

2.2. Operadores y operaciones comunes - Pseudocódigo

Una vez leídos los datos (ver Memoria Practica 1) el siguiente paso consiste en encapsularlos dentro de la clase. Primero tenemos que normalizarlos (ver Memoria Práctica 1) y una vez realizada la normalización pasamos a generar las particiones para el ***5-fold cross-validation***, operación sencilla que no precisa ser incluida en pseudocódigo. Llegados a este punto estamos listos para describir los operadores comunes.

2.2.1. Función objetivo: 1-NN

Nuestro objetivo será utilizar el clasificador **1-NN** para maximizar la función objetivo de nuestro problema, considerando un determinado subconjunto de los datos, que llamaremos **subconjunto de entrenamiento (Train)** a partir del cual aprenderemos un clasificador que posteriormente validaremos con el resto del subconjunto de datos, que llamaremos **subconjunto de validación (Test)**, utilizando la técnica del *leave-one-out* cuando sea necesaria. La principal diferencia con la práctica anterior radica en que ahora nuestra función objetivo considera un nuevo criterio: **la minimización del número de características** a usar en el clasificador final. Por lo tanto, nuestro problema se reformula como sigue: $FuncionObjetivo = tasa_{clas} \cdot \alpha + tasa_{red} \cdot (1 - \alpha)$

Donde $\alpha = 0{,}5$ representa la importancia que se le da a cada una de las tasas y la *tasa de clasificación* cuenta el número de instancias bien clasificadas de toda la muestra T(aciertos) sobre la totalidad de la muestra (instancias de T) : $tasa_{clas} = 100 \cdot \frac{aciertosenT}{instanciasdeT}$

y la *tasa de reducción* considera cuántas componentes del vector de pesos, del total de características, son menores que 0.1, esto es: $tasa_{red} = 100 \cdot \frac{||w_i<0{,}1||}{numCaracteristicas}$

A continuación se muestra el pseudocódigo de la función objetivo, donde nuestro clasificador devuelve el valor de la función objetivo (que apartir de ahora llamaremos **Agregado**) teniendo en cuenta los N individuos que forman parte del *Train* y el número de características "irrelevantes "que nuestro clasificador descartará.

Algorithm 1 Clasificador 1-NN: función objetivo

```
Aciertos = 0
for all T_i ∈ Test do
  Inicializar (d_min)
  for all T_j ∈ Train do
    //Leave one out
    if T_i ≠ T_j then
      d_j ← calcularDistanciaEuclidea(T_i, T_j, W_i)
      if d_j < d_min then
        d_min = d_j, c_min = T_j → clase
      end if
    end if
  end for
  if c_min == T_i then
    Aciertos++
  end if
end for
tasa_clas = 100 · Aciertos/instanciasDeT
tasa_red = 100 · ||w_i < 0,1||/numCaracteristicas
Agregado = tasa_clas · α + tasa_red · (1 − α)
Return Agregado
```

2.2.2. 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.

Algorithm 2 Generar Solucion Aleatoria

```
for i=1 to C do
  for j=1 to G do
    cromosoma ← Rand(0, 1)
  end for
  Poblacion ← cromosoma
end for
Return Poblacion
```

2.2.3. Operador de selección de los AG y AM – Torneo Binario

Se eligen de forma aleatoria dos padres y aquel cuya función objetivo sea mayor será seleccionado y pasará a formar parte de los candidatos para la operación de cruce. Así, para el AGG aplicaremos tantos torneos como cromosomas de la población, de forma que se generen la misma cantidad de posibles padres a cruzar. Para el **AM**, dado que el operador de cruce es el **Cruce Aritmético**, se generarán el doble de padres puesto que el cruce genera un único descendiente. En el algoritmo, P representa el número de padres a generar y C el tamaño de la población. El algoritmo devuelve los mejores padres que se considerarán para el cruce.

Algorithm 3 Torneo Binario

```
for i = 1 to P do
  padre_1 ← Rand(0, C)
  padre_2 ← Rand(0, C)
  //Para evitar elegir el mismo padre 2 veces
  while padre_1 == padre_2 do
    padre_2 ← Rand(0, C)
  end while
  if padre_1 → fitness > padre_2 → fitness then
    Padres ← padre_1
  else
    Padres ← padre_2
  end if
end for
Return Padres
```

2.2.4. Operador de cálculo de la Distancia Euclídea

Este operador, imprescindible en nuestra función objetivo, calcula la distancia Euclídea entre dos ejemplos P_1 y P_2. Para el calculo de la distancia se tienen en cuenta las G componentes del vector característico de cada uno de los ejemplos, y el cómputo de la distancia para una característica $i \in [0, G]$ se tendrá en cuenta **siempre y cuando la componente i-esima del vector de pesos W sea mayor que 0.1**, ya que sabemos que cuando esto no ocurre, esta carácterística será descartada y se considerará como irrelevante. El vector de pesos W pondera cada una de las G características del ejemplo en cuestión. Este operador devuelve el cálculo de la distancia Euclídea haciendo uso del desenrollado de bucles para reducir el número de saltos en el cálculo.

Algorithm 4 Calcular Distancia Euclidea

for *i=0 to G* **do**
  if $W_i > 0{,}1$ **then**
    $Distancia = Distancia + W_i * (P1_i - P2_i)^2$
  end if
end for
$Distancia \leftarrow \sqrt{Distancia}$
Return Distancia

2.2.5. Generador de mutación/vecino

Partimos de un generador de números dentro de una distribución normal de media 0 y varianza 0.3. Este operador sirve para generar vecinos (BL) y mutaciones (AG y ES).

Algorithm 5 Generar Vecino/Mutación

$\mu = 0, \sigma = 0{,}3; mutacion = \mathcal{N}(\mu, \sigma)$
$Actualizar(componente_i, mutacion)$
if $Pesos(componente_i) > 1$ **then**
  $Pesos(componente_i) = 1$
else if $Pesos(componente_i) < 0$ **then**
  $Pesos(componente_i) = 0$
end if

2.2.6. Algoritmo de Búsqueda Local

La **Búsqueda Local (BL)** parte de un vector 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** teniendo en cuenta que la función objetivo es ahora el **Agregado** (tasa de clasificación y tasa de reducción, ambas ponderadas) que proporciona nuestro **Clasificador 1-NN**.

El funcionamiento del algoritmo es sencillo: mientras queden **vecinos que generar** (máximo de $20 \cdot N$ vecinos, con N igual al número de características) y no hayamos superado la **cantidad máxima de evaluaciones** (llamadas a la función objetivo), realizaremos una permutación aleatoria de un vector de índices que representa las N características, seleccionando **uno de estos índices** en cada iteración del bucle interno y a partir de éste generaremos un **vecino** siguiendo la función descrita anteriormente y se **evaluará la función objetivo** del vecino generado: si el vecino generado es **mejor que la solución actual**, éste sustituirá a la solución actual. El algoritmo devuelve la solución **W** que mejores resultados ha conseguido. NOTA: La generación de la solución aleatoria de la que parte el algoritmo se realiza en la función *main*, pero se incluye dentro del pseudocodigo del algoritmo dado que es un paso previo a éste.

Algorithm 6 Búsqueda Local (LS)

```
vecinos = evaluaciones = 0
//Generar una solucion aleatoria de N caracteristicas y evaluar fitness
W ← GenerarSolucion(1, N)
clasificacion ← 1NN(Train, W)
//Inicio del algoritmo: condiciones de parada
while vecinos < 20N && evaluaciones < 15000 do
  //Permutacion aleatoria de los indices de W
  indices ← PermutacionAleatoria(1 : N)
  //Exploración del entorno con esquema del primer mejor
  for all N_i ∈ N && !hayMejora do
    //Generar un nuevo vecino (mutando la característica i-esima) y evaluar su fitness
    W_i ← GenerarVecino(W, N_i), vecinos++
    fitness_i ← 1NN(Train, W_i), evaluaciones++
    //El primer vecino que mejore a la solucion actual hará que termine el bucle interno
    if clasificacion_i > clasificacion then
      clasificacion = clasificacion_i , W = W_i
      hayMejora = TRUE
    end if
  end for
end while
//Se devuelve la mejor solucion encontrada durante el proceso
Return W
```

3. Estructura del método de búsqueda y operaciones relevantes de los algoritmos: ES, ILS y DE.

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 proceso del **cálculo de la temperatura inicial** y el **esquema de enfriamiento** del algoritmo de **ES**, con su posterior pseudocódigo. Una vez explicado todo este procedimiento pasaremos con la **ILS** donde nos centraremos principalmente en el **operador de mutación (mutación fuerte)** que utiliza este algoritmo. Finalmente, realizaremos una descripción detallada y concisa de los algoritmos de **DE**, donde para cada uno de los modelos considerados abordaremos la **recombinación** y el **reemplazamiento** de las soluciones como principales novedades, así como cualquier otro procedimiento que ayude a comprender de mejor forma su funcionamiento. Sin más preámbulos, comenzamos.

3.1. Algoritmo de Enfriamiento Simulado (ES)

El algoritmo de **Enfriamiento Simulado** es un método que permite realizar la búsqueda por entornos y se caracteriza principalmente por hacer uso de un **criterio de aceptación de soluciones adaptable** a lo largo de toda la ejecución del algoritmo. ¿Cómo se adapta este criterio de aceptación? Nuestro algoritmo, tal y como su nombre indica, simula el proceso de enfriamiento de partículas y sus cambios energéticos conforme la **temperatura** decrece hasta converger a una situación estable.

Según el **trabajo de Metrópolis**, las leyes de la termodinámica dicen que existe una distribución de probabilidad que intenta explicar la probabilidad de que a una temperatura **t** se dé un incremento de temperatura de magnitud δE. Por tanto nuestro algoritmo hace uso de una variable llamada **Temperatura** que permitirá dedicidr en que medida aceptar o rechazar soluciones vecinas que sean **peores que la actual**, dado que el objetivo principal es evitar estancarnos en un óptimo local.

A grosso modo, el algoritmo genera en cada iteración un número preestablecido de vecinos (en nuestro caso) y justo después entra en juego el criterio de aceptación: si la solución es mejor, se acepta. Pero si es peor, existe una probabilidad de aceptación $P_{acept} = exp(\frac{-\delta}{T})$ que permitirá al algoritmo aceptar soluciones peores que la actual pero permitiendole salir de óptimos locales. En nuestro caso, δ representa la diferencia de costes entre la solución actual y la vecina, y T la temperatura actual; por tanto a mayor temperatura, mayor probabilidad de aceptación de peores soluciones, lo que indica que durante las primeras fases, el algoritmo acepta soluciones peores que la actual y por tanto **explora** es espacio de soluciones. Conforme se va enfriando la temperatura, se aceptarán menos soluciones peores y se **explotará** más la mejor solución. Por tanto este algoritmo nos permitirá aprender un clasificador con la particion de **Train** y luego validar el mismo con la partición de **Test**.

3.1.1. Esquema de enfriamiento y cálculo de tempearatura inicial

El esquema de enfriamiento empleado será el esquema de Cauchy modificado, que controla el enfriamiento de la temperatura en función del número de enfriamientos que queramos realizar. El cálculo de la temperatura en cada uno de los procesos de enfriamientos y el valor del parámetro β se puede ver a continuación.

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k}; \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

La temperatura inicial vendrá dada por la siguiente fórmula:

$$T_0 = \frac{\mu \cdot C(S_0)}{-ln(\phi)}$$

donde $C(S_0)$ representa el coste de la solucion inicial y $\phi \in [0,1]$ es la probabilidad de aceptar una solución un μ por 1 peor que la inicial. Los valores considerados para $\mu = \phi = 0{,}3$ y la temperatura final será 10^{-3}. Cabe destacar además que el número **maximo de vecinos** que se generarán en cada iteración será de **10*N** (con N igual al número de características) y el **número máximo de éxitos** que se considera en cada proceso de búsquda es de **0.1*max_vecinos**. A continuación se muestra el pseudocódigo del algoritmo con todas las consideraciones anteriores. NOTA: al igual que en la BL, partimos de una solución inicial generada en el **main**, pero se incluye como parte del algoritmo en el pseudocódigo.

Algorithm 7 Enfriamiento Simulado (ES)

enfriamientos = 0 , numExitos =1
//Generación y evaluación de la solucion inicial
$S_0 \leftarrow GenerarSolucion(1, N)$, $C(S_0) = 1NN(Train, S_0)$
//Actualizacion de las soluciones: actual y mejor
$S_{act} = S_{best} = S_0$, $C(S_{act}) = C(S_{best}) = C(S_0)$
//Temperatura inicial y β
$T_0 = \frac{\mu \cdot C(S_0)}{-ln(\phi)}$, $\beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$, $T_k = T_0$
//Bucle principal del algoritmo
while $enfriamientos < M$ && $numExitos > 0$ **do**
  $numExitos = 0$
  //Bucle de exploración (primeras fases) y explotación (fases finales)
  for *i=0 to max_vecinos && numExitos <maxExitos* **do**
    //Generacion y evaluación de vecino
    $i = Rand(0 : N)$
    $S' = GenerarVecino(S_{act}, i)$
    $C(S') = 1NN(Train, S')$
    //Proceso de aceptación/rechazo de soluciones
    $\Delta f = C(S') - C(S_{act})$
    if $\Delta f > 0$ || $Rand(0,1) \leq exp(\frac{-\Delta f}{T})$ **then**
      //Se actualiza la solucion actual
      $S_{act} = S'$, $C(S_{act}) = C(S')$, $numExitos++$
      //Se comprueba si cambia la mejor solucion
      if $C(S_{act}) > C(S_{best})$ **then**
        $S_{best} = S'$, $C(S_{best}) = C(S')$
      end if
    end if
  end for
  //Enfriamos la temperatura
  $T_k = \frac{T_k}{1 + \beta \cdot T_k}$, $enfriamientos++$
end while
Return S_{best}

3.2. Algoritmo de Búsqueda Local Reiterada (ILS)

El esquema de una Búsqueda Local Reiterada consiste en **aplicar un algoritmo de BL de forma repetida** sobre una solución inicial a la que se le aplica una mutación fuerte. Este operador de mutación cambia el peso de $t = 0{,}1 \cdot N$ características del vector de pesos **W** de forma aleatoria y además la distribución Normal a partir de la cual se generan los valores aleatorios tiene una $\sigma = 0{,}4$.

Algorithm 8 Mutación Fuerte ILS

*//**Indices de las componentes a mutar***
$indices \leftarrow (0 : size(W))$
for *i=0 to t* **do**
*//**Eleccion aleatoria de la componente i-esima a mutar. Posteriormente se elimina para no considerarla nuevamente***
$mutar = Rand(0 : size(W))$, $indices \leftarrow (indices - mutar)$
$W \leftarrow GenerarVecino(W, mutar)$
end for
Return W

La **BL se aplicará 15 veces** y el límite de llamadas a la **función objetivo** = 1000.

Algorithm 9 Búsqueda Local Reiterada

$t = 0{,}1 \cdot N$
*//**Generación de solución inicial y aplicación de BL sobre ésta***
$S_{act} = GenerarSolucion(1, N)$, $S_{act} = BL(Train, S_{act})$
*//**Actualización de todas las soluciones***
$S' = S_{act}$, $S_{best} = S_{act}, C_{best} = 0$
*//**Comienzo del algoritmo de ILS***
for *i=1 to 15* **do**
*//**Aplicacion de la mutación fuerte sobre S' y posterior BL***
$S' \leftarrow MutacionFuerte(S', t)$; $S' \leftarrow BL(Train, S')$
*//**Criterio de aceptación: la mejor de las soluciones entre** S_{act} **y S'***
if $S' \rightarrow fitness < S_{act} \rightarrow fitness$ **then**
$S' = S_{act}$, $S' \rightarrow fitness = S_{act} \rightarrow fitness$
else
$S_{act} = S'$, $S_{act} \rightarrow fitness = S' \rightarrow fitness$
end if
*//**Guardamos la mejor solucion en caso de mejorar la actual***
if $S' \rightarrow fitness > C_{best}$ **then**
$S_{best} = S'$, $C_{best} = S' \rightarrow fitness$
end if
end for
Return S_{best}

3.3. Algoritmos de Evolución Diferencial: DE_Rand y DE_CurrentToBest

El algoritmo de **Evolución Diferencial** es un modelo evolutivo que enfatiza en el proceso de **mutación**, realizando posteriormente una operación de cruce completamente distinta a cualquier mecanismo visto hasta ahora. El algoritmo trabaja con una población de soluciones iniciales $P_{x,0}$ de tamaño **N**, cada una con **C características** y va generando una población nueva en cada iteración del algoritmo siguiendo un mecanismo de **Mutación Diferencial**.

Con respecto de cada uno de los vectores $X_{i,g}$ de la población en la generación **g**, este mecanismo genera un nuevo vector mutado que llamaremos $V_{i,g}$, añadiendo un vector diferencia que se ha generado con un conjunto de muestras aleatorias y que además se escala siguiendo un factor de escalado $F \in [0, 1]$. Por lo tanto, al final de la generación **g-ésima** tendremos otra población $P_{u,g}$ de la misma dimensión que la población inicial pero sus componentes serán estos N vectores mutados de C características.

¿Cuál es el mecanismo de recombinación seguida para generar cada uno de los N vectores? El mecanismo seguido será la **Recombinación Binomial**, en la que cada vector $U_{j,i,g}$ donde **j** indica el la **componente j-ésima del vector** que se está generando, i el **índice del vector en generación** y g la generación actual, se construye siguiendo las condiciones descritas a continuación:

$$u_{j,i,g} = \begin{cases} v_{j,i,g} & if \qquad Rand(0,1) \leq CR || j = j_{rand} \\ \\ x_{j,i,g} & otherwise \end{cases}$$

Finalmente, cuando ya disponemos de ambas poblaciones, llega la hora de efectuar el **reemplazamiento**, que será **uno a uno**. Esto quiere decir que si el vector $u_{i,g}$ tiene un mayor valor de función fitness que el vector objetivo $x_{i,g}$, reemplazará al mismo para la generación $g + 1$. En caso contrario, el vector objetivo se mantiene para la siguiente generación.

Comenzaremos con el algoritmo DE_Rand y posteriomente explicaremos el DE_Current To Best. Como últimas consideraciones, ambas versiones del algoritmo terminarán tras 15000 evaluaciones de la función objetivo, con un tamaño de la población $N = 50$, una probabilidad de cruce $CR = 0{,}5$ y un factor de escalado $F = 0{,}5$.

3.3.1. Algoritmo DE_Rand

El esquema de mutación diferencial empleado por este algoritmo viene dada por la siguiente expresión: $V_{i,g} = X_{r1,G} + F \cdot (X_{r2,G} - X_{r3,G})$ donde $X_{r1,G}$, $X_{r2,G}$ y $X_{r3,G}$ indican 3 padres seleccionados de forma aleatoria de la población en la generación G. Siguiendo el proceso de **recombinación binomial** y todas las restricciones a considerar, se presenta el pseudocódigo del algoritmo, donde se aprende el clasificador con la partición de *Train*.

Algorithm 10 DE_Rand1

$evaluaciones = 0$, $S_{best0} = 0$, $C_{best0} = 0$
$indices \leftarrow (0 : C)$
//Generar población inicial (N vectores de C características) y evaluar
$P_0 = GenerarSolucion(N, C)$
for all $X_i \in P_0$ **do**
  $evaluaciones_0 \leftarrow 1NN(Train, X_i)$
  //Guardar la mejor solucion de esta poblacion
  if $X_i \rightarrow fitness > C_{best0}$ **then**
    $S_{best0} = X_i$, $C_{best0} = X_i \rightarrow fitness$
  end if
end for
//Comienzo del algoritmo DE
while $evaluaciones < 15000$ **do**
  $S_{bestAct} = 0$, $C_{bestAct} = 0$
  for *i=0 to N* **do**
    //Seleccion de 3 padres de forma aleatoria
    $index \leftarrow indices - i$, $RandomShuffle(index)$
    $padre1 \leftarrow index[0]$, $padre2 \leftarrow index[1]$, $padre3 \leftarrow index[2]$
    //Proceso de mutación y recombinación binomial
    for *j=0 to C* **do**
      if $Rand(0,1) \leq CR$ || $Rand(0, C) == j$ **then**
        $offspring_{j,i} = padre1_j + F \cdot (padre2_j - padre3_j)$
      else
        $offspring_{j,i} = P_0 \rightarrow X_{j,i}$
      end if
    end for
    //Evaluacion del offspring
    $evaluaciones_{act} \leftarrow 1NN(Train, offspring_j)$
    //Proceso de reemplazamiento uno a uno
    //El offspring es mejor que el vector objetivo
    if $C_{off} > X_i \rightarrow fitness$ **then**
      $P_{actual} \leftarrow P_{actual} \cup offspring_j$
    else
      $P_{actual} \leftarrow P_{actual} \cup X_j$
    end if
    //Guardamos la mejor solucion de la generacion actual
    if $evaluaciones_{act,j} \rightarrow fitness > C_{bestAct}$ **then**
      $S_{bestAct} = X_j$, $C_{bestAct} = evaluaciones_{act,j} \rightarrow fitness$
    end if
  end for
  //Nos quedamos con la mejor solucion de las dos poblaciones
  if $C_{bestAct} > C_{best0}$ **then**
    $S_{best0} = S_{bestAct}$, $C_{best0} = C_{bestAct}$
  end if
  //Reemplazamiento de la poblacion
  $P_0 = P_{act}$, $evaluaciones_0 = evaluaciones_{actual}$
end while
Return S_{best0}

3.3.2. Algoritmo DE_CurrentToBest

El esquema de mutación diferencial empleado por este algorimto viene dada por la siguiente expresión: $V_{i,g} = X_{i,G} + F \cdot (X_{best,G} - X_{i,G}) + F \cdot (X_{r1,G} - X_{r2,G})$ donde $X_{r1,G}$ y $X_{r2,G}$ son 2 padres seleccionados de forma aleatoria de la población en la generación G, $X_{i,G}$ es el vector objetivo y $X_{best,G}$ la mejor solución de la generación G. Se ha dividido el algoritmo por partes para evitar ocupar más paginas de las necesarias. Los cambios de este algoritmo frente a su hermano DE_Rand se muestran en color rojo.

Algorithm 11 DE_CurrentToBest

$evaluaciones = 0$, $S_{best0} = 0$, $C_{best0} = 0$
$indices \leftarrow (0 : C)$
//Generar población inicial (N vectores de C características) y evaluar
$P_0 = GenerarSolucion(N, C)$
for all $X_i \in P_0$ **do**
  $evaluaciones_0 \leftarrow 1NN(Train, X_i)$
  //Guardar la mejor solucion de esta poblacion
  if $X_i \rightarrow fitness > C_{best0}$ **then**
    $S_{best0} = X_i$, $C_{best0} = X_i \rightarrow fitness$
  end if
end for

Los principales cambios con respecto al primer algoritmo de DE es que solo se seleccionan 2 padres de forma aleatoria antes de comenzar a generar el offspring, además de que el mecanismo de Mutación Diferencial tiene en cuenta el vector objetivo en la iteración i-ésima, además de la mejor solución de la generación actual. Como bien hemos dicho anteriormente, estos principales cambios se desctacan en color rojo, quedando el resto del algoritmo sin cambio alguno.

Algorithm 12 DE_CurrentToBest

*//**Comienzo del algoritmo DE***
while $evaluaciones < 15000$ **do**
  $S_{bestAct} = 0$, $C_{bestAct} = 0$
  for i=*0 to* N **do**
    //Seleccion de 2 padres de forma aleatoria
    $index \leftarrow indices - i$, $RandomShuffle(index)$
    $padre1 \leftarrow index[0]$, $padre2 \leftarrow index[1]$
    *//**Proceso de mutación y recombinación binomial***
    for j=*0 to* C **do**
      if $Rand(0,1) \leq CR$ || $Rand(0,C) == j$ **then**
        $offspring_{j,i} = X_{j,i} + F \cdot (S_{best0} - X_{j,i}) + F \cdot (padre1_j - padre2_j)$
      else
        $offspring_{j,i} = P_0 \rightarrow X_{j,i}$
      end if
    end for
    *//**Evaluacion del offspring***
    $evaluaciones_{act} \leftarrow 1NN(Train, offspring_j)$
    *//**Proceso de reemplazamiento uno a uno***
    *//**El offspring es mejor que el vector objetivo***
    if $C_{off} > X_i \rightarrow fitness$ **then**
      $P_{actual} \leftarrow P_{actual} \cup offspring_j$
    else
      $P_{actual} \leftarrow P_{actual} \cup X_j$
    end if
    *//**Guardamos la mejor solucion de la generacion actual***
    if $evaluaciones_{act,j} \rightarrow fitness > C_{bestAct}$ **then**
      $S_{bestAct} = X_j$, $C_{bestAct} = evaluaciones_{act,j} \rightarrow fitness$
    end if
  end for
  *//**Nos quedamos con la mejor solucion de las dos poblaciones***
  if $C_{bestAct} > C_{best0}$ **then**
    $S_{best0} = S_{bestAct}$, $C_{best0} = C_{bestAct}$
  end if
  *//**Reemplazamiento de la poblacion***
  $P_0 = P_{act}$, $evaluaciones_0 = evaluaciones_{actual}$
end while
Return S_{best0}

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 esa misma partición. 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 su pseudocódigo se puede ver en la memoria de la Práctica 1, por lo que a continuación definiremos el funcionamiento del RELIEF en forma de pseudocódigo.

Algorithm 13 RELIEF

```
//Recorremos cada ejemplo y calculamos su amigo y enemigo mas cercano
for all Tr_i ∈ Train do
    distancia_amigo ← CalcularAmigoMasCercano(Tr_i, Train)
    distancia_enemigo ← CalcularAmigoMasCercano(Tr_i, Train)
    //Actualizamos los pesos con en funcion de las distancias al amigo y
    enemigo mas cercano de cada ejemplo
    for all W_i ∈ W do
        W_i ← W_i + distancia_enemigo − distancia_amigo
    end for
end for
//Calculamos el mayor de los pesos y normalizamos
W_max = MAX(W)
for all W_j ∈ W do
    if W_j < 0 then
        W_j = 0
    else
        W_j = W_j / W_max
    end if
end for
Return W
```

Hasta aquí todo lo referente al pseudocódigo de los algoritmos. 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. A lo largo de la práctica hemos hecho uso de 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.

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 que se ha seguido para el desarollo de la misma.

5.1. 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 y reducción, amigo y enemigo más cercano, generación de soluciones aleatorias y la generación de vecino/mutación normal con distintas distribuciones de probabilidad.
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 del **Enfriemiento Simulado (ES):** utilizando operador de generación de solución aleatoria y de vecino por mutación normal.
6. Implementación de la BL para la **Búsqueda Local Reiterada (ILS)**: realizar cambios sobre la BL de la práctica anterior e implementación del código de la ILS a partir de la BL y la generación de vecinos por mutación normal con distintos parámetros μ y σ.
7. Implementación de los **dos modelos de Evolución Diferencial (DE)**: valiéndonos de los operadores de generación de soluciones aleatorias y siguiendo el esquema de Mutación Diferencial (diferente para cada modelo), recombinación binomial y reemplazamiento uno a uno.
8. Implementación de los **operadores comunes de los AG y AM**: los operadores de cruce BLX-α y CA, así como el Torneo Binario y la generación de un gen en un intervalo determinado.
9. Implementación del AGG-BLX y AM(10,0.1Mej): recuperación del código de éstos de la práctica 1 y adaptación a la nueva función objetivo.

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 9 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**. En la Práctica 1 se consideraba un esquema de **Aprendizaje y Validación** distinto al considerado en ésta práctica, al igual que la **función objetivo**. Para ésta práctica sin embargo, el esquema de Aprendizaje y Validación empleado nos permitirá ajustar un mejor clasificador al entrenar el mismo con un 80 % de la totalidad de los datos. Dado que la función objetivo también ha sido modificada y ahora tiene doble propósito, la comparación que realizaremos de cada uno de los algoritmos para escoger el mejor de forma global es la siguiente: un algoritmo será mejor en función de la **tasa de clasificación** que es capaz de proporcionar siempre y cuando la **simplicidad** del mismo sea la **mayor posible**, esto es, que la **cantidad de características relevantes sea mínima** (y por tanto, la tasa de reducción sea cuanto más alta, mejor).

Como aclaración, las tablas de resultados propuestos se han organizado de forma un tanto distinta a la práctica anterior: al contener más campos y para evitar problemas con los márgenes y demás, hemos separado cada tabla del algoritmo en 3 subtablas, una para cada base de datos. El color naranja en la última fila de cada tabla indica la media de cada uno de los valores.

6.1. Clasificador 1-NN

El algoritmo 1NN no se considera un algoritmo de aprendizaje en sí, sino que nos sirve para evaluar la calidad de nuestro clasificador siguiendo la función objetivo descrita en la sección 2.2.1. Las evaluaciones que realiza el 1NN, sin estar asociado a ningún otro algoritmo, parten de un vector de pesos unitario **W** (todas las componentes a 1) e intentan clasificar cada ejemplo del **Test** en función del vecino más cercano con respecto de la partición de **Train**. Aquellos ejemplos que son de la misma clase tienden a organizarse de forma similar en el espacio N-dimensional del problema considerado y ésta es una de las condiciones que aprovecha nuestro "algoritmo "para ofrecer una tasa de clasificación de alta calidad. Pero, ¿que pasa con la tasa de reducción? Veamos los resultados a continuación para cada una de las bases de datos.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	82,926800	0,000000	41,463400	0,00097
Partición 2	82,926800	0,000000	41,463400	0,00109
Partición 3	87,804900	0,000000	43,902400	0,00087
Partición 4	95,121900	0,000000	47,561000	0,00086
Partición 5	84,090900	0,000000	42,045500	0,00083
Media	86,5743	0,0000	43,2871	0,00092

Figura 6.1: Tabla 6.1: Resultados del algoritmo 1-NN en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
81,521700	0,000000	40,760900	0,0043
82,608700	0,000000	41,304400	0,0031
86,956500	0,000000	43,478300	0,0031
80,434800	0,000000	40,217400	0,0029
83,695700	0,000000	41,847800	0,0026
83,0435	0,0000	41,5218	0,0032

Figura 6.2: Tabla 6.1: Resultados del algoritmo 1-NN en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
98,230100	0,000000	49,115000	0,0034
95,575200	0,000000	47,787600	0,0027
96,460200	0,000000	48,230100	0,0030
95,575200	0,000000	47,787600	0,0026
92,307700	0,000000	46,153900	0,0025
95,6297	0,0000	47,8148	0,0029

Figura 6.3: Tabla 6.1: Resultados del algoritmo 1-NN en el problema del APC. Wdbc

Como era de esperar, nuestro algoritmo 1NN no es capaz de realizar reducción alguna sobre la cantidad de componentes utilizadas por el clasificador, y por tanto su tasa de

reducción será siempre igual a 0. Para los siguientes algoritmos seguiremos un esquema similar, comentando en cada casa como afectan las distintas técnicas de búsqueda que implementa cada uno y la utilización de poblaciones de soluciones frente a soluciones únicas en el resultado final: nuestro clasificador.

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. Como bien sabemos, nuestro algoritmo se vale del cálculo de las distancias entre el ejemplo más cercano y más lejano de su misma clase, de forma que estas distancias sean las que ponderen la importancia de cada una de las N características de nuestro clasificador.

No es descabellado pensar que este algoritmo proporcionará buenos resultados en cuanto a porcentaje de clasificación de las muestras de entrenamiento, dado que nos basamos en el mismo principio de organización de ejemplos de la misma clase en zonas del espacio muy cercanas, pero nuevamente nos encontramos con que no será capaz de cumplir la función objetivo en su totalidad, ya que **no es capaz de explotar el espacio de soluciones lo suficiente como para reducir el peso de un número considrable de carácterísticas** por debajo de 0.1. Por lo tanto, tendremos un clasificador un tanto por ciento mejor que el 1NN primitivo pero la complejidad del mismo seguirá siendo alta. Veamos los resultados que avalan nuestro análisis.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	78,048800	13,333300	45,691100	0,00966
Partición 2	80,487800	15,000000	47,743900	0,00773
Partición 3	90,243900	11,666700	50,955300	0,00681
Partición 4	97,561000	13,333300	55,447200	0,00687
Partición 5	84,090900	10,000000	47,045500	0,00915
Media	86,0865	12,6667	49,3766	0,00804

Figura 6.4: Tabla 6.2: Resultados del algoritmo RELIEF en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
82,608700	10,526300	46,567500	0,0395
86,956500	19,298200	53,127400	0,0369
86,956500	22,807000	54,881800	0,0357
85,869600	28,070200	56,969900	0,0369
82,608700	26,315800	54,462200	0,0337
85,0000	21,4035	53,2018	0,0365

Figura 6.5: Tabla 6.2: Resultados del algoritmo RELIEF en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
97,345100	3,333330	50,339200	0,0338
95,575200	6,666670	51,120900	0,0310
97,345100	3,333330	50,339200	0,0297
96,460200	13,333300	54,896800	0,0305
94,017100	0,000000	47,008500	0,0329
96,1485	5,3333	50,7409	0,0316

Figura 6.6: Tabla 6.2: Resultados del algoritmo RELIEF en el problema del APC. Wdbc

La distribución de las particiones, al ser de forma aleatoria y no balanceada, puede llevarnos a resultados como los que se ven en las tablas. Nuestro algoritmo RELIEF parte de un vector de pesos iniciales a 0 y en función de la distribución de los ejemplos de las clases en las distintas particiones, calcula la función objetivo. En bases de datos "fáciles de aprender "como lo son Sonar y Wdbc, ofrece buenos resultados de clasificación pero muy pobres en reducción, ya que intenta ponderar todas las carácterísticas por igual debido al mecanismo inherente del algoritmo. Sin embargo para Spambase, ofrece resultados intermedios y que ya permiten avistar una reducción importante en el número de carácterísticas que utiliza el clasificador, pero aún así sigue siendo un modelo complejo que intentaremos mejorar con los siguientes 3 algoritmos propuestos.

6.3. Enfriamiento Simulado (ES) vs RELIEF

El algoritmo de **Enfriamiento Simulado** ofrece una propuesta distinta a cualquier otra vista hasta el momento. Su objetivo es partir de una solución aleatoria de pesos W y durante las primeras fases del algoritmos realizar una **exploración exhaustiva del espacio de soluciones** mediante una distribución de probabilidad que le permite escapar de óptimos locales, aceptando soluciones en ocasiones mucho peores que la actual.

En cuanto la **temperatura empieza a caer**, el algoritmo **cambia exploración por explotación** y se centra en optimizar la solución mediante la generación de vecinos por mutación normal durante un periodo de tiempo preestablecido, marcado por un número de vecinos que se le permite generar y un número máximo de aciertos permitidos. El balance entre exploración y explotación permite cumplir los dos objetivos de nuestro clasificador: potencia y complejidad reducida. Veamos la calidad de los resultados que ofrece nuestro algoritmo y comparemos con nuestro algoritmo RELIEF a través de las tablas de Máximos, Minimos y Medias que se introdujeron en la Práctica 1.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	85,365900	31,666700	58,516300	3,02839
Partición 2	82,926800	30,000000	56,463400	3,04835
Partición 3	87,804900	36,666700	62,235800	2,98901
Partición 4	92,682900	40,000000	66,341500	2,90783
Partición 5	86,363600	40,000000	63,181800	2,88503
Media	87,0288	35,6667	61,3478	2,97172

Figura 6.7: Tabla 6.3: Resultados del algoritmo ES en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
85,869600	29,824600	57,847100	13,9688
83,695700	33,333300	58,514500	14,0195
81,521700	33,333300	57,427500	14,0636
83,695700	36,842100	60,268900	13,8594
70,652200	42,105300	56,378700	13,4573
81,0870	35,0877	58,0873	13,8737

Figura 6.8: Tabla 6.3: Resultados del algoritmo ES en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
96,460200	36,666700	66,563400	12,2167
97,345100	40,000000	68,672600	12,0749
98,230100	40,000000	69,115000	12,3316
95,575200	40,000000	67,787600	11,8053
93,162400	43,333300	68,247900	11,6992
96,1546	40,0000	68,0773	12,0255

Figura 6.9: Tabla 6.3: Resultados del algoritmo ES en el problema del APC. Wdbc

Para realizar las comparaciones pertinentes entre RELIEF y ES vamos utilizar las tablas de Máximos, Minimos y Medias, donde cada valor propuesto corresponde la función objetivo conjunta: tasa de clasificación y reducción.

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.10: Tabla de Máximos, Minimos y Media para RELIEF

ES	Sonar	Wdbc	Spambase
Media	61,34776	68,0773	58,08734
Máximo	66,3415	69,115	60,2689
Mínimo	56,4634	66,5634	56,3787
Diferencia	9,8781	2,5516	3,8902

Figura 6.11: Tabla de Máximos, Minimos y Media para ES

El algoritmo de ES ofrece mejoras considerables con respecto a nuestro algoritmo de comparación por el simple hecho de no realizar una búsqueda de tipo voraz, sino centrada en el balance entre exploración y explotación del espacio de soluciones. Si bien es cierto que el porcentaje de reducción en cada base da datos es de media a penas un tercio del

total de características, se puede vislumbrar el camino que seguiremos a partir de ahora con el resto de algoritmos con los que vamos a comparar. El Enfriamiento Simulado es capaz de potenciar la calidad del clasificador mediante la simulación del enfriamiento de las partículas y es claro vencedor en su pugna con el RELIEF. Todos los algoritmos serán capaces de clasificar con cierta calidad pero la diferencia la marcará aquel que sea capaz de encontrar buenas soluciones y que sean además de baja complejidad.

6.4. Algoritmo de Búsqueda Local Reiterada (ILS vs RELIEF)

La ILS es un algoritmo simple pero eficaz que permite obtener buenos resultados en un tiempo relativamente corto. ¿Cuál es la base de su éxito? Cuando se aplica la BL sobre una solución de cierta calidad, lo normal es que ésta mejore. La ILS es un algoritmo que **aplica de forma continuada la BL sobre una solución de cierta calidad** y que además sufre, en cada iteración, una **mutación fuerte**. Esta mutación fuerte será la que le permita salir de óptimos locales y por tanto permita al algorimto explorar el espacio de soluciones pero con menor libertad de movimiento a través de éste, ya que solo se modifican un 10 % de las características.

Lo que nuestro algoritmo busca realmente es, partiendo de que soluciones de cierta calidad también se encuentras distribuidas en zonas cercanas en el espacio, explorar el espacio cercano de esas soluciones y una vez encontrada una solución de buena calidad, explotarla para conseguir el mayor porcentaje de acierto a la vez que se reduce el número de características del clasificador. Veamos los resultados del algoritmo que posteriormente compararemos con nuestro algoritmo RELIEF.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	82,926800	46,666700	64,796700	30,31280
Partición 2	78,048800	48,333300	63,191100	30,21910
Partición 3	85,365900	43,333300	64,349600	30,41390
Partición 4	87,804900	45,000000	66,402400	30,38600
Partición 5	84,090900	43,333300	63,712100	29,86930
Media	83,6475	45,3333	64,4904	30,24022

Figura 6.12: Tabla 6.4: Resultados del algoritmo ILS en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
78,260900	47,368400	62,814600	141,7930
76,087000	54,386000	65,236500	141,4750
84,782600	42,105300	63,443900	143,4880
85,869600	45,614000	65,741800	143,7580
75,000000	42,105300	58,552600	143,0780
80,0000	46,3158	63,1579	142,7184

Figura 6.13: Tabla 6.4: Resultados del algoritmo ILS en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
95,575200	50,000000	72,787600	71,5109
93,805300	53,333300	73,569300	71,4073
98,230100	60,000000	79,115000	71,1314
96,460200	60,000000	78,230100	72,5793
94,017100	56,666700	75,341900	70,1782
95,6176	56,0000	75,8088	71,3614

Figura 6.14: Tabla 6.4: Resultados del algoritmo ILS en el problema del APC. Wdbc

Aunque el balance entre exploración y exploración sea menor con respecto al algoritmo de ES, la potencia de la propia BL permite reducir de forma notable la cantidad de características utilizadas en el clasificador, y por tanto aumentando los valores de nuestra función objetivo de forma que se posicione como el mejor algoritmo hasta el momento, a la espera de los DE. ¿Qué tan bueno es con respecto al algoritmo de compración RELIEF?

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.15: Tabla de Máximos, Minimos y Media para RELIEF

ILS	Sonar	Wdbc	Spambase
Media	64,49038	75,80878	63,15788
Máximo	66,4024	79,115	65,7418
Mínimo	63,1911	72,7876	58,5526
Diferencia	3,2113	6,3274	7,1892

Figura 6.16: Tabla de Máximos, Minimos y Media para ILS

Poniendo en una balanza el desempeño de RELIEF frente a la ILS, podemos comprobar como el segundo es mucho más efectivo en términos de que no existen diferencias tan grandes entre máximos y mínimos de los valores de la función objetivo, poniendo de manifiesto la capacidad del algoritmo de realizar una **explotación intensiva de soluciones de buena calidad** para obetener aún mejores soluciones.

Mejora la bondad de nuestro clasificador en todas las bases de datos y aunque sea ligeramente peor en cuanto a términos de clasificación, siguiendo el principio de la Navaja de Occam, son **preferibles soluciones mucho más simples para un mismo problema** y la ILS aporta este tipo de soluciones en las que solo se utilizan, de media, el 50 % de las características, mostrando un modelo mucho más sencillo de interpretar y utilizar.

6.5. Algoritmos de Evolución Diferencial (DE) vs RELIEF

Los algoritmos de Evolución Diferencial son de los más potentes de la literatura actual y sus buenos resultados se deben a que son algoritmos de **caracter poblacional** que utilizan un mecanismo de mutación basado en las **diferencias entre ejemplos** (escogidos de forma aleatoria) de la población y que sufre un proceso de escalado para potenciar la calidad de las soluciones. El proceso mediante el cual se **recombinan las soluciones** para generar descendientes potencialmente mejores que los actuales juega un papel muy importante en nuestro algoritmo y también permite la exploración del espacio de soluciones durante gran parte de la ejecución del algoritmo.

Pero lo que realmente permite a estos algoritmos despuntar frente a otros es la sencillez del **modelo de mutación** previamente comentado: al elegir individuos de la población de forma aleatoria , independientemente de su calidad, se potencia la convergencia de la mayoría de éstos en soluciones de cierta calidad conforme pasan las generaciones. Esto incurre en tener un **conjunto de soluciones diversas** pero todas de cierta calidad, donde finalmente triunfará aquella que haya conseguido abrirse paso entre la diversidad y la calidad, permitiendo al algoritmo converger en una solución que permita tener un modelo lo más efectivo y sencillo posible. Veamos los resultados.

6.5.1. Algoritmo de DE_Rand

Este algoritmo de DE selecciona 3 padres de forma aleatoria para generara un descendiente siguiendo la mutación que se puede ver en la expresión 3.3.1. Los resultados de este algoritmo son considerablemente mejores que cualquiera de los anteriores.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	73,170700	88,333300	80,752000	24,43520
Partición 2	75,609800	91,666700	83,638200	24,30380
Partición 3	68,292700	91,666700	79,979700	24,33670
Partición 4	80,487800	90,000000	85,243900	24,11590
Partición 5	72,727300	90,000000	81,363600	23,38920
Media	74,0577	90,3333	82,1955	24,11616

Figura 6.17: Tabla 6.5: Resultados del algoritmo DERand en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
88,043500	92,982500	90,513000	115,0430
83,695700	94,736800	89,216200	114,8990
91,304400	94,736800	93,020600	117,4420
82,608700	92,982500	87,795600	113,6450
90,217400	92,982500	91,599900	113,7030
87,1739	93,6842	90,4291	114,9464

Figura 6.18: Tabla 6.5: Resultados del algoritmo DERand en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
92,920300	93,333300	93,126800	93,8471
94,690300	93,333300	94,011800	94,5482
92,035400	93,333300	92,684400	93,6221
92,920300	93,333300	93,126800	94,3485
87,179500	93,333300	90,256400	93,0546
91,9492	93,3333	92,6412	93,8841

Figura 6.19: Tabla 6.5: Resultados del algoritmo DERand en el problema del APC. Wdbc

6.5.2. Algoritmo de DE_CurrentToBest

El DE_CurrentToBest selecciona solo 2 padres de forma aleatoria para generara un descendiente, y se vale además de la mejor solución obtenida hasta el momento y del vector objetivo de la generación G, siguiendo la expresión 3.3.2

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	70,731700	63,333300	67,032500	27,39460
Partición 2	82,926800	51,666700	67,296700	28,13610
Partición 3	85,365900	53,333300	69,349600	28,21010
Partición 4	87,804900	61,666700	74,735800	27,76120
Partición 5	81,818200	53,333300	67,575800	26,93540
Media	81,7295	56,6667	69,1981	27,68748

Figura 6.20: Tabla 6.6: Resultados del algoritmo DE-CTB en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
85,869600	59,649100	72,759300	126,8970
88,043500	61,403500	74,723500	124,3190
73,913000	71,929800	72,921400	125,5230
83,695700	66,666700	75,181200	127,2770
65,217400	70,175400	67,696400	125,5910
79,3478	65,9649	72,6564	125,9214

Figura 6.21: Tabla 6.6: Resultados del algoritmo DE-CTB en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
93,805300	76,666700	85,236000	101,9020
95,575200	66,666700	81,120900	103,9460
95,575200	80,000000	87,787600	100,6500
94,690300	80,000000	87,345100	100,7490
88,888900	83,333300	86,111100	103,3390
93,7070	77,3333	85,5201	102,1172

Figura 6.22: Tabla 6.6: Resultados del algoritmo DE-CTB en el problema del APC. Wdbc

¿Por qué DE_Rand ofrece mejores resultados que la versión Current To Best? Existe una razón de peso que avala estos resultados y la misma radica en el proceso de **mutación de las soluciones**. En DE_Current To Best los offspring son generados basándose en 2 padres aleatorios, la mejor solución y el vector objetivo, por lo que **una mejor solución estancada en un óptimo local condicionaría gravemente los descendientes** generados a partir de ésta, además de que **el vector objetivo puede no ser de suficiente calidad** y por tanto este condicionando el offspring generado en cada momento.

Por el contrario DE_Rand hace uso de 3 padres aleatorios e intenta mejorarlos a través de la mutación. Estos 3 padres aleatorios serán seleccionados posteriormente para generar offspring a partir de ellos, ofreciendo la **posibilidad de que éstos también mejoren de cara a la generación siguiente** (sin condicionarlos con posibles óptimos locales) y que por tanto toda la población converja en una calidad razonablemente buena.

Una vez visto cual algoritmo presenta mejores soluciones, vamos a compararlo con el RELIEF.

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.23: Tabla de Máximos, Minimos y Media para RELIEF

DE_RAND	Sonar	Wdbc	Spambase
Media	82,19548	92,64124	90,42906
Máximo	85,2439	94,0118	93,0206
Mínimo	79,9797	90,2564	87,7956
Diferencia	5,2642	3,7554	5,225

Figura 6.24: Tabla de Máximos, Minimos y Media para DE_RAND

Las mejoras son abismales para todas las bases de datos: los valores medios de función objetivo son muy altos, lo que indica que este algoritmo ofrece soluciones de alta calidad y complejidad reducida en la que apenas utilizamos de media un 10 % del total de características. Las posibilidades que ofrecen los algoritmos de DE al considerar los mecanismos de mutación y recombinación, a pesar de su simpleza, son muy potentes y un algoritmo de caracter Greedy no es capaz de hacer frente a los resultados que este ofrece.

Además, trabajar a nivel de poblaciones no hace sino sumar ventajas a la hora de obtener soluciones de calidad: tenemos un **mayor espacio de soluciones** que considerar, cada una de esas soluciones son candidatas a ofrecer los mejores resultados, la **capacidad de exploración se ve aumentada** considerablemente y la **explotación** de las mejores soluciones se da **sobre toda la población**. Es por eso que los algoritmos de DE son tan potentes y como bien muestran los resultados de las tablas, para tener un clasificador potente y a la vez sencillo no hace falta invertir demasiado tiempo, lo que es un valor añadido a un algoritmo que ya de por sí es una de las mejores alternativas que hemos encontrado para el problema del APC.

6.6. BL vs RELIEF

Los 3 algoritmos siguientes se recuperan de la práctica anterior pero adaptados para considerar la nueva función objetivo. Cada uno parte de las mismas condiciones que en la Práctica 1 por lo que serán comparados con el RELIEF y con el mejor de los algoritmos obtenidos hasta ahora.

La 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. Veamos los resultados obtenidos.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	78,048800	35,000000	56,524400	2,52281
Partición 2	80,487800	23,333300	51,910600	2,47641
Partición 3	80,487800	28,333300	54,410600	2,49200
Partición 4	90,243900	31,666700	60,955300	2,43867
Partición 5	79,545500	33,333300	56,439400	2,41701
Media	81,7628	30,3333	56,0481	2,46938

Figura 6.25: Tabla 6.7: Resultados del algoritmo BLen el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
82,608700	29,824600	56,216600	11,0660
83,695700	26,315800	55,005700	10,8227
84,782600	28,070200	56,426400	11,0494
81,521700	29,824600	55,673200	10,7861
83,695700	35,087700	59,391700	11,4036
83,2609	29,8246	56,5427	11,0256

Figura 6.26: Tabla 6.7: Resultados del algoritmo BL en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
95,575200	43,333300	69,454300	4,8696
95,575200	36,666700	66,120900	4,8466
95,575200	30,000000	62,787600	5,2365
94,690300	30,000000	62,345100	4,9615
93,162400	33,333300	63,247900	4,8679
94,9157	34,6667	64,7912	4,9564

Figura 6.27: Tabla 6.7: Resultados del algoritmo BL en el problema del APC. Wdbc

La BL, al ser un algoritmo que se aplica sobre una única solución (aleatoria), dependerá de la calidad de la misma por lo que se esperaba que los resultados ni siquiera se acercasen a los del mejor algoritmo hasta el momento, el DE_Rand. Si queremos comparar la mejora que ofrece este algoritmo frente al de comparación RELIEF, atendamos a las tablas siguientes.

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.28: Tabla de Máximos, Minimos y Media para RELIEF

BL	Sonar	Wdbc	Spambase
Media	56,04806	64,79116	56,54272
Máximo	60,9553	69,4543	59,3917
Mínimo	51,9106	62,3451	55,0057
Diferencia	9,0447	7,1092	4,386

Figura 6.29: Tabla de Máximos, Minimos y Media para BL

Contrario a lo comentado con respecto de la práctica 1, ahora la BL sí que supera al algoritmo de comparación, dado que al aplicarse un nuevo método de **Aprendizaje y Validación** se permite explotar mucho más las soluciones de buena calidad, además de que estamos entrenando nuestro clasificador con mayor volúmen de datos, lo que conlleva a una mejora sustancial del mismo debido a la robustez con la que la BL permite explotar soluciones de cierta calidad.

6.7. AGG-BLX vs RELIEF

El algoritmo genético generacional con cruce $BLX-\alpha$ se recupera intacto con respecto a la práctica anterior, pero considerando el nuevo método de **Aprendizaje y Validación** además de la función objetivo que busca un equilibrio entre potencia de clasificación y simplicidad del clasificador. Con los mismos parámetros que en la práctica anterior se muestran los resultados.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	78,048800	21,666700	49,857700	31,56580
Partición 2	80,487800	21,666700	51,077200	31,52600
Partición 3	85,365900	23,333300	54,349600	31,53480
Partición 4	90,243900	25,000000	57,621900	31,50100
Partición 5	84,090900	20,000000	52,045500	30,38700
Media	83,6475	22,3333	52,9904	31,30292

Figura 6.30: Tabla 6.8: Resultados del algoritmo AGG-BLX en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
90,217400	24,561400	57,389400	147,8100
86,956500	22,807000	54,881800	144,7460
82,608700	24,561400	53,585100	147,2420
84,782600	24,561400	54,672000	146,4380
82,608700	22,807000	52,707900	146,5470
85,4348	23,8596	54,6472	146,5566

Figura 6.31: Tabla 6.8: Resultados del algoritmo AGG-BLX en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
97,345100	30,000000	63,672600	125,1330
93,805300	30,000000	61,902700	124,2480
95,575200	33,333300	64,454300	125,1820
94,690300	30,000000	62,345100	124,7590
93,162400	30,000000	61,581200	124,8710
94,9157	30,6667	62,7912	124,8386

Figura 6.32: Tabla 6.8: Resultados del algoritmo AGG-BLX en el problema del APC. Wdbc

Si bien los resultados en cuanto a clasificación son relativamente buenos, la tasa de reducción se ve notablemente perjudicada. ¿Por qué razón? El algoritmo genético considera el cruce de dos padres aleatorios con una capacidad mínima en la exploración y

con una mutación de apenas 1 o 2 individuos por generación, lo que **limita considerablemente que la calidad de las soluciones mejore** de cara a la reducción del número de carácterísticas de nuestro clasificador. Además, su tiempo de ejecución se dispara notablemente y comparandolo con el RELIEF, no merece la pena utilizar este algoritmo para nuestro problema: invertimos más tiempo para obtener soluciones muy parecidas.

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.33: Tabla de Máximos, Minimos y Media para RELIEF

AGG-BLX	Sonar	Wdbc	Spambase
Media	52,99038	62,79118	54,64724
Máximo	57,6219	64,4543	57,3894
Mínimo	49,8577	61,5812	52,7079
Diferencia	7,7642	2,8731	4,6815

Figura 6.34: Tabla de Máximos, Minimos y Media para AGG-BLX

6.8. AM-(10,0.1Mej) vs RELIEF

El Algoritmo Memético de la práctica anterior se basaba en el cruce aritmético, ya que fue el que mejores resultados dio. En esta ocasión, se recupera teniendo en cuenta que la función objetivo ha cambiado y los resultados son relativamente buenos, teniendo en cuenta que parte de un algoritmo genético que, como hemos visto anteriormente, no ofrecía muy buenos resultados.

	Sonar			
	%_clas	*%_red*	*Agr*	T
Partición 1	73,170700	50,000000	61,585400	28,09790
Partición 2	85,365900	56,666700	71,016300	27,91910
Partición 3	85,365900	53,333300	69,349600	28,37400
Partición 4	90,243900	51,666700	70,955300	28,41780
Partición 5	81,818200	51,666700	66,742400	27,43580
Media	83,1929	52,6667	67,9298	28,04892

Figura 6.35: Tabla 6.9: Resultados del algoritmo AMMej en el problema del APC. Sonar

Spambase			
%_clas	*%_red*	*Agr*	T
83,695700	63,157900	73,426800	127,2390
84,782600	54,386000	69,584300	129,6040
84,782600	57,894700	71,338700	130,8770
82,608700	52,631600	67,620100	129,4680
75,000000	50,877200	62,938600	133,1660
82,1739	55,7895	68,9817	130,0708

Figura 6.36: Tabla 6.9: Resultados del algoritmo AMMej en el problema del APC. Spam

Wdbc			
%_clas	*%_red*	*Agr*	T
93,805300	46,666700	70,236000	114,3330
94,690300	33,333300	64,011800	114,7730
96,460200	60,000000	78,230100	108,9960
97,345100	36,666700	67,005900	117,8900
93,162400	66,666700	79,914500	102,3670
95,0927	48,6667	71,8797	111,6718

Figura 6.37: Tabla 6.9: Resultados del algoritmo AMMej en el problema del APC. Wdbc

En cuanto a qué tan bueno es respecto del algoritmo de comparación cabe destacar que la BL que se aplica tras un número determinado de generaciones contribuye notablemente a la mejora de un algoritmo que sin ésta, sería un simple algoritmo genético. Las mejoras se aprecian mayormente sobre aquellas bases de datos con más ejemplos.

RELIEF	Sonar	Wdbc	Spambase
Media	49,3766	50,74092	53,20176
Máximo	55,4472	54,8968	56,9699
Mínimo	45,6911	47,0085	46,5675
Diferencia	9,7561	7,8883	10,4024

Figura 6.38: Tabla de Máximos, Minimos y Media para RELIEF

AM-Mej	Sonar	Wdbc	Spambase
Media	67,9298	71,87966	68,9817
Máximo	71,0163	79,9145	73,4268
Mínimo	61,5854	64,0118	62,9386
Diferencia	9,4309	15,9027	10,4882

Figura 6.39: Tabla de Máximos, Minimos y Media para AM

7. Conclusiones finales

Para abordar este apartado vamos a valernos de la tabla de resultados globales obtenida tras todas las experimentaciones y de un conjunto de gráficas que ayudarán a visualizar la calidad de los algoritmos frente a cada una de las bases de datos.

	Sonar				Wdbc				Spambase			
	%_clas	*%_red*	*Agr*	T	*%_clas*	*%_red*	*Agr*	T	*%_clas*	*%_red*	*Agr*	T
1-NN	86,5743	0,0000	43,2871	0,0009	95,6297	0,0000	47,8148	0,0029	83,0435	0,0000	41,5218	0,0032
RELIEF	86,0865	12,6667	49,3766	0,0080	96,1485	5,3333	50,7409	0,0316	85,0000	21,4035	53,2018	0,0365
ES	87,0288	35,6667	61,3478	2,9717	96,1546	40,0000	68,0773	12,0255	81,0870	35,0877	58,0873	13,8737
ILS	83,6475	45,3333	64,4904	30,2402	95,6176	56,0000	75,8088	71,3614	80,0000	46,3158	63,1579	142,7184
DE_rand	74,0577	90,3333	82,1955	24,1162	91,9492	93,3333	92,6412	93,8841	87,1739	93,6842	90,4291	114,9464
DE_current_to_best	81,7295	56,6667	69,1981	27,6875	93,7070	77,3333	85,5201	102,1172	79,3478	65,9649	72,6564	125,9214
LS	81,7628	30,3333	56,0481	2,4694	94,9157	34,6667	64,7912	4,9564	83,2609	29,8246	56,5427	11,0256
AGG-BLX	83,6475	22,3333	52,9904	31,3029	94,9157	30,6667	62,7912	124,8386	85,4348	23,8596	54,6472	146,5566
AM-(10,0.1mej)	83,1929	52,6667	67,9298	28,0489	95,0927	48,6667	71,8797	111,6718	82,1739	55,7895	68,9817	130,0708

Figura 7.1: Tabla de resultados globales para el problema del APC

A primera vista se puede ver como para todas las bases de datos el mejor algoritmo de todos es el **DE_Rand**, quizas no siempre en términos de clasificación pero siempre

mejorando a cualquier algoritmo en términos de reducción y función objetivo. Como no queda suficientemente claro vamos a elaborar un conjunto de gráficas, una por cada base de datos, para esclarecer cuales son los 3 mejores algoritmos a nivel global y las razones de su éxito.

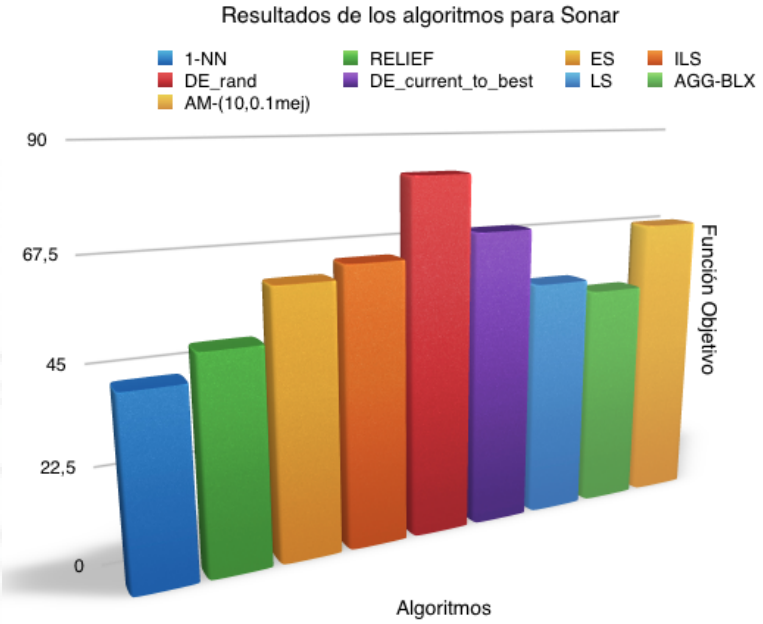

Figura 7.2: Gráfica de resultados: Sonar

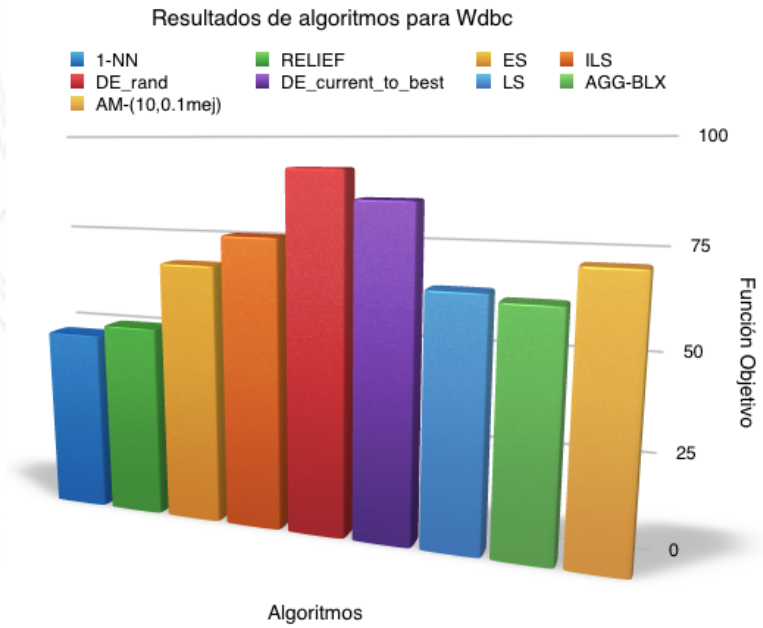

Figura 7.3: Gráfica de resultados: WDBC

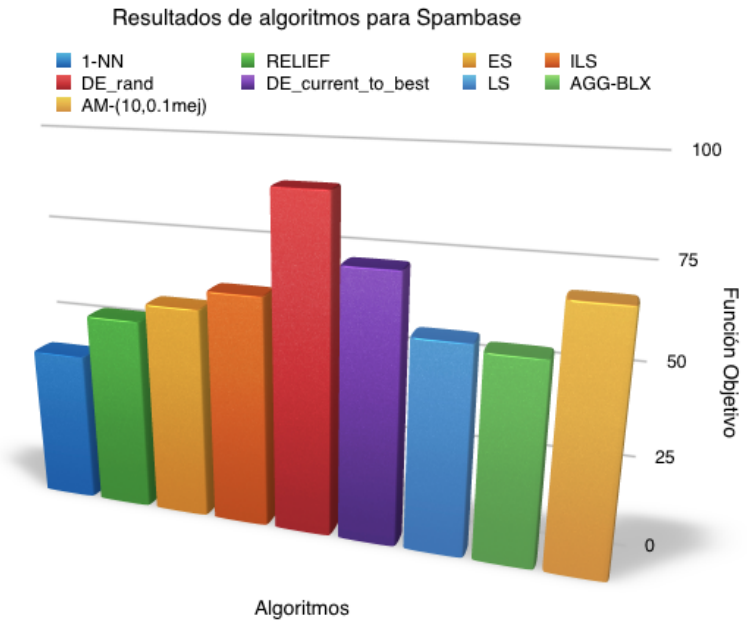

Figura 7.4: Gráfica de resultados: Spambase

Sin duda alguna, el mejor algoritmo de todos es el DE_Rand, el segundo mejor su hermano DE_CurrentToBest y el tercero mejor (en dos de las 3 bases de datos) es el AM-(10,0.1Mej). Las razones del exito de cada uno son las que se explican a lo largo de las comparaciones: los DE tienen un esquema de mutación y recombinación sencillo pero potente que permite obtener muy buenas soluciones y el AM tiene la BL que optimiza las soluciones de cierta calidad, por lo que estos serían los 3 algoritmos con los que deberíamos afrontar problemas de este tipo con bases de datos de este estilo, ganando en calidad del ajuste, en simplicidad del clasificador y en eficiencia en cuanto a tiempo.