METAHEURíSTICAS

Práctica 1

Universidad de Granada

CURSO 2016/2017

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

Contenido
Enfriamiento Simulado
Búsqueda Local Reiterada
Evolución Diferencial

Autor:
Juan Luis Suárez Díaz
77148642-H
jlsuarezdiaz@correo.ugr.es
GRUPO 2 (VIERNES)
Cuarto Curso del DGIIM

4de junio de2017

Índice

Descripción del problema

Estamos ante un problema de aprendizaje automático, en concreto un problema de clasificación, en el que se pretende optimizar el rendimiento del clasificador 1NN. Este clasificador, dada una muestra de datos y un nuevo dato a clasificar, obtiene la clase para el nuevo dato como aquella correspondiente a la del dato más cercano en la muestra. Con esta descripción, el clasificador obtendrá el vecino más cercano ponderando en la misma medida todas las características de los datos que manejamos, lo que en principio puede darnos peores resultados, puesto que es posible que no todas las características consideradas tengan la misma relevancia a la hora de realizar la clasificación.

Mediante el Aprendizaje de Pesos en Características se pretende, a partir de la muestra de entrenamiento, obtener un vector de pesos asociado al conjunto de características, de forma que la distancia para obtener el vecino más cercano se calcule ponderando cada componente con el peso obtenido. Si el aprendizaje es efectivo, el vector de pesos nos permitirá aumentar la tasa de acierto a la hora de clasificar nuevos datos. En las siguientes secciones estudiaremos distintas heurísticas con las que afrontar este problema y veremos en qué medida permiten mejorar el rendimiento del clasificador 1NN.

Consideraciones comunes

Esquemas de representación

Trabajaremos en concreto con 3 conjuntos de datos: `sonar`, `Wdbc` y `Spambase`. Estos conjuntos están formados por un conjunto de ejemplos, cada uno con un número fijo de características y una clase asociada. Los ejemplos junto con sus características los representaremos en una matriz, donde cada fila es un ejemplo y cada columna una característica. Además, para dar igual importancia a todos los atributos, la matriz estará normalizada por columnas (características), utilizando como criterio de normalización los valores máximo y mínimo encontrados para cada característica en la matriz.

Más adelante tendremos que hacer particiones de los datos, de forma que en cada partición haya un subconjunto de ejemplos. Para representar las particiones utilizaremos un vector de índices `v`, de forma que, si `p` es la partición considerada y `m` es la matriz de ejemplos del problema, se tiene que `p[i] = m[v[i]]`, es decir, el ejemplo i-ésimo en la partición es el ejemplo en el problema dado por el elemento i-ésimo del vector de índices.

Las soluciones con las que trabajaremos serán vectores reales de pesos, de tamaño el número de características del problema (las columnas de la matriz). Las soluciones tomarán siempre valores en $[0, 1]$.

Función objetivo

Para evaluar el rendimiento del clasificador 1NN con un algoritmo determinado, utilizaremos la técnica de validación cruzada 5-Fold. Para ello, haremos 5 particiones distintas de los datos, y para cada partición, aprenderemos el vector de pesos con el contenido de su partición y lo evaluaremos con las restantes. El valor de rendimiento promedio será la media de estas 5 evaluaciones.

```
for i from 1 to 5
    solucion = algoritmo(particion[i])
    fitness[i] = f_objetivo(datos-particion[i], solucion)

end

return media(fitness)
```

Para evaluar una solución sobre la partición de entrenamiento, seguiremos dos criterios: por un lado, obtendremos el porcentaje de los datos que quedan bien clasificados al evaluarlo sobre el resto de la partición. Por otro lado, se considerará que una solución es mejor cuanto mayor número de características haya conseguido eliminar, entendiendo por eliminable a cada característica para la que el peso obtenido sea cercano a cero (concretamente, menor que 0.1). Ambos criterios ponderarán en la misma medida dentro de la función objetivo. Como consecuencia del criterio de reducción, al calcular la distancia en el criterio de clasificación tampoco se tendrán en cuenta los atributos con pesos menores que 0.1.

La tasa de clasificación de una solución sobre una partición de entrenamiento, se realizará aplicando el clasificador sobre cada dato de la partición y viendo si la clase obtenida por el clasificador coincide con la clase del dato. Para evitar que el vecino más cercano proporcionado por el clasificador sea el mismo dato, el dato a clasificar se aparta de la muestra, siguiendo el procedimiento *Leave One Out*. El porcentaje de aciertos será la medida de evaluación sobre la partición.

```
def tasa_clas(particion, solucion)
    aciertos = 0
```

```
    for dato in particion
      c = clasificar_1NN(particion, solucion, dato)
      if c = dato.clase
          aciertos++
      end
    end

    return (100 * aciertos)/particion.tamaño
end
```

Finalmente, se describe el pseudocódigo del clasificador 1NN (para un dato en la partición de entrenamiento con *Leave One Out*).

```
def clasificar_1NN(particion,solucion,dato)
    # Para evitar asignar el dato a clasificar
    if particion.primero != dato then clase_min = particion.primero.clase
    else clase_min = particion.segundo.clase

    if particion.primero != dato then dist_min = sqDist(particion.primero, dato)
    else dist_min = sqDist(particion.segundo, dato)

    for dato_test in particion
        if dato_test != dato # Dejamos fuera el dato a clasificar
            if sqDist(dato_test, dato) < dist_min
                clase_min = dato_test.clase
                dist_min = sqDist(dato_test, dato)
            end
        end
    end

    return clase_min
end

# Función distancia euclidea al cuadrado ponderada con los pesos de la solución
def sqDist(dato1, dato2, solucion)
    suma = 0
    for i from 1 to solucion.size
        if solucion[i] >= 0.1  # Considerando la reducción
            suma = suma + solucion[i]*(dato1[i]-dato2[i])^2
        end
    end
end
```

La tasa de reducción consistirá en contar simplemente los pesos en la solución que valgan menos que 0.1:

```
def tasa_red(solucion)
    num_reds = 0
    for peso in solucion
        if peso < 0.1 then num_reds++
    end
    return 100 * (num_reds)/solucion.size
end
```

Finalmente, como ya se ha comentado, el valor final de la función objetivo se obtiene combinando ambas tasas: $f = \alpha\ tasa\_clas + (1 - \alpha) tasa\_red$. En este caso las ponderamos igual, es decir, $\alpha = 0,5$.

La evaluación de la partición test se realizará aplicando este mismo procedimiento usando la partición aprendida como el conjunto sobre el que se busca el vecino más cercano. En este caso, la tasa de clasificación se obtendrá sin *Leave One Out*, mientras que evidentemente la tasa de reducción coincidirá con la obtenida durante el aprendizaje.

Un aspecto importante a destacar en la función objetivo que se ha tenido en cuenta durante la implementación es que el orden de eficiencia es de $O(P \times P \times S)$, donde P es el tamaño de la partición y S el tamaño de la solución. Es un orden considerable y la función se llamará gran cantidad de veces a lo largo de los distintos algoritmos, por lo que es bueno considerar cualquier posible mejora de esta. Para ello, en la implementación se ha optado por modificar ligeramente la estructura del algoritmo, sin modificar el resultado. Se ha tenido en cuenta que la función distancia ponderada es simétrica respecto de los datos para una solución prefijada. De esta forma, la modificación considerada para la implementación consiste en la inicialización al principio del algoritmo de una matriz triangular con las distancias entre todos los datos de la partición, y la obtención de las distancias durante la clasificación se reduce a acceder a una posición de la matriz ya creada. Aunque la clase de complejidad sigue siendo la misma, el uso de una matriz solamente triangular para calcular las distancias permite reducir las iteraciones a la mitad.

Generación de vecinos

Consideraremos que una solución es vecina de una solución si se diferencian en una única componente en un valor que sigue una distribución normal centrada en 0 y con desviación 0.3. De esta forma, para generar un vecino de una solución, dada una componente, le sumaremos un valor extraído de la distribucuón normal anterior. Para cumplir con las restricciones del problema, si la suma supera el valor 1 o alcanza un valor negativo, se truncará a 1 o a 0, respectivamente.

```
def mov(solucion,i,sigma)
    solucion[i] = solucion[i] + normal(0,sigma).nuevoNumero()
end
```

Generación de soluciones aleatorias

Para generar una solución aleatoria, a cada componente le asignaremos un valor uniformemente distribuido entre 0 y 1.

```
def solucionAleatoria(problema)
    for i in 1 to problema.numeroAtributos
        solucion[i] = uniforme(0,1).nuevoNumero()
    end
    return solucion
end
```

Búsqueda local

Se mantiene el mismo algoritmo de búsqueda local implementado en la práctica anterior. El algoritmo de búsqueda local utilizado sigue el modelo del primer mejor. Con el operador de generación de

vecinos indicado previamente se generan nuevas soluciones, y en cuanto una mejora a la solución actual, se actualiza como nueva solución. El procedimiento se repite mientras no se verifique ninguna de las condiciones de parada. En este caso, las condiciones de parada vienen dadas por un número máximo de evaluaciones de la función objetivo, o bien, por un número máximo de vecinos generados sin obtener mejora.

En cuanto a aspectos de implementación, se considera durante todo el proceso una única solución que va siendo modificada, y en caso de que no haya mejora, se devuelve la componente modificada a su estado anterior. Esto mejora la eficiencia al evitar copiar soluciones, aunque el cuello de botella está en la llamada a la función objetivo.

Finalmente, la selección de la componente a modificar de la solución se hace de forma aleatoria, pero recorriendo todas las componentes antes de dar una nueva pasada al vector. Para eso se utiliza una permutación que se va barajando cada vez que se recorre.

El pseudocódigo es el siguiente:

```
def BusquedaLocal(part_train, solucion, max_evals, max_no_mej)
    num_evals = 0
    no_mejoras = 0

    fitness = f_objetivo(part_train,solucion)
    permutacion = [1,...,solucion.size]

    # Mientras no condiciones de parada
    while num_evals < max_evals and no_mejoras < max_no_mej
        shuffle(permutacion)
        for indice in permutacion
            peso_actual = solucion[indice] # Para deshacer la mutación
            mov(solucion,indice,0.3)       # Generamos vecino
            newfit = f_objetivo(part_train,solucion)
            num_evals++             # Nueva evaluación de la función objetivo

            if newfit > fit
                 # Hay mejora, actualizamos fitness y resteamos no_mejoras
                fitness = newfit
                no_mejoras = 0
            else
                # No hay mejora, deshacemos la mutación e incrementamos no_mejoras
                solucion[indice] = peso_actual
                no_mejoras++
            end
        end
    end
    return solucion
end
```

Métodos de búsqueda

Enfriamiento simulado

El algoritmo de enfriamiento simulado se inspira en los procesos de calentamiento de algunos materiales. En este caso, la aceptación de soluciones depende en gran medida de una variable temperatura. La variable temperatura determinará una probabilidad con la que el algoritmo de exploración podrá aceptar soluciones peores a la actual. Análogamente a su inspiración física, a mayor temperatura, habrá más movimiento y se explorará el espacio de búsqueda con una mayor amplitud. Cuando la temperatura vaya disminuyendo, se intensificará la explotación de las soluciones obtenidas. La generación de nuevas soluciones seguirá el esquema de vecinos habitual, con la novedad del criterio de aceptación basado en la temperatura. El esquema de búsqueda se resume en el siguiente pseudocódigo:

```
def SimulatedAnnealing(particion)

    inicializarTemperaturas()

    best_sol = s = solucionAleatoria()
    fit = best_fit = fitness(particion,s)

    while not condicionesParada()

        num_neighbours = 0
        num_success = 0
        permutacion = generarPermutacion(s.size())

        while num_neighbours < max_neighbours and num_success < max_success

            # Para ir mutando todas las componentes
            if(permutacion.empty()) permutacion = generarPermutacion(s.size())
            rnd = permutacion.extraer()

            s_i = s[rnd] # Para deshacer la mutación después sin copiar

            s.mov(rnd,0.3) # Generamos vecino
            newfit = fitness(particion,s)
            num_neighbours++

            diff = (fit - newfit)/100.0 # Diferencia de costes

            # Criterio de aceptación
            # Aceptamos si es mejor, o si es peor según la probabilidad siguiente
            if diff != 0 and (diff < 0 or AleatorioUniforme(0,1) <= exp(-diff/temp))

                # Actualizamos
                fit = newfit
                if fit > best_fit # Actualizamos mejor solución si es necesario
                    best_sol = s
                    bestfit = fit
                end

                num_success++
            else
              s[rnd] = s_i # Deshacemos mutación
```

```
            end
        end

        enfriar()
    end

    return best_sol
```

El criterio de parada utilizado viene dado por un máximo de evaluaciones de la función objetivo, por una parte, o por otra, que en alguno de los bucles internos (los de temperatura fija) no se haya aceptado ninguna solución, es decir, se llega a una temperatura a la que el algoritmo no se presta a aceptar peores soluciones y tampoco a mejorar por generación de vecinos.

En cuanto a la inicialización de temperaturas, el procedimiento seguido para determinar la temperatura inicial es el siguiente:

$$T_0 = \frac{\mu C(S_0)}{\log(\phi)}$$

Donde se han tomado $\mu = \phi = 0,3$ y $C(S_0)$ es el coste de la solución inicial.

Finalmente, se utiliza el esquema de enfriamiento de Cauchy:

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

Donde $\beta = \frac{T_0 - T_f}{M T_0 T_f}$ y $M = \frac{max\_evaluaciones}{max\_vecinos}$. El criterio para la llamada al esquema de enfriamiento se basa en un número máximo de vecinos generados $max\_vecinos$ y un número máximo de éxitos (soluciones que son aceptadas por el criterio de la temperatura) $max\_success$, como se mostraba en el pseudocódigo anterior.

Búsqueda Local Reiterada

El esquema de búsqueda empleado en la búsqueda local reiterada es el siguiente: partimos de una solución inicial optimizada con búsqueda local, y durante repetidas veces a la solución que tenemos le aplicamos una mutación brusca, y a dicha mutación le aplicamos búsqueda local. Nos quedamos con la que tenga un mayor valor de la función objetivo y repetimos el proceso.

```
def ILS(particion)
    # Solución inicial+BL
    s = solucionAleatoria()
    busquedaLocal(s,particion)
    fit = f_objetivo(particion,s)

    while not condicionesParada()
        s' = mutacionBrusca(s)
        busquedaLocal(s',particion)
        newfit = f_objetivo(particion,s')

        if newfit > fit
            s = s'
            fit = newfit
        end
    end
```

```
end
```

Finalmente, el operador de mutación brusca de la solución consiste en un conjunto de mutaciones simples (la generación de vecinos usual) con una mayor desviación (0.4 en lugar del usual 0.3). El número de mutaciones simples realizado ha sido un 10 % del total de atributos. A la hora de mutar, se puede optar tanto por seleccionar los atributos completamente al azar como por controlar que no mute siempre el mismo atributo. En la implementación se ha optado por esta segunda opción, utilizando vectores de permutaciones de los atributos.

```
def mutacionBrusca(solucion)
    s = solucion.copia()

    num_mutaciones = 0.1 * solucion.size
    # Obtenemos los índices a mutar (por cualquiera de los
    # procedimientos aleatorios indicados previamente)
    indices_mutacion = obtenerAleatorios(min = 0, max = solucion.size - 1,
                        num = num_mutations)

    for i in indices_mutacion
        s.mov(i,0.4) # Operador de mutación simple
    end

    return s
end
```

Evolución Diferencial

La evolución diferencial es un algoritmo evolutivo en el que la población de soluciones avanza de acuerdo a las reglas de combinación que se muestran más adelante. Como algoritmo evolutivo, la estrategia de resolución se puede resumir en el siguiente pseudocódigo:

```
# Genera soluciones uniformememnte
# distribuidas en [0,1]
iniciarPoblacionAleatoria()

while not condiciones_parada()
    nuevaGeneracion()
end

return poblacion.mejorSolucion()
```

Las condiciones de parada serán, en general, el número de evaluaciones de la función objetivo, aunque también podrían considerarse otras posibilidades, como el número de generaciones desarrolladas. El esquema de desarrollo de nuevas generaciones, para los distintos algoritmos de evolución diferencial, puede descomponerse en las siguientes fases:

```
def nuevaGeneracion()
    seleccion()     # Selección de padres
```

```
    recombinacion() # Obtención de hijos
    reemplazo()     # Sustitución de la población
end
```

El proceso de selección consiste en extraer grupos de padres de la población. Se extraerán tres padres por cada individuo de la población, que serán los que intervengan posteriormente en las recombinaciones para cada hijo (el tercer padre podría no intervenir según el operador de cruce utilizado, pero por simplicidad se extrae siempre).

```
def seleccion()
    for i from 1 to poblacion.size()
        [r1,r2,r3] = aleatoriosDistintosEntre(0,poblacion.size()-1)
        padres.añadir([poblacion[r1],poblacion[r2],poblacion[r3]])
    end
end
```

Para la recombinación, utilizaremos el operador de cruce correspondiente según el modelo de evolución diferencial escogido. En cada recombinación pueden intervenir hasta tres padres escogidos al azar en la población, además de la mejor solución de cada generación. La aleatoriedad con la que se escogieron los padres en el proceso de selección nos permite ahora recorrer la lista de padres e ir pasándoselos de tres en tres al operador de cruce.

```
def cruce()
    for i from 0 to poblacion..size()-1
        hijos.añadir(operadorCruce(padres[3*i],padres[3*i+1],
                    padres[3*i+2],poblacion[i],mejor_solucion))
    end
end
```

A continuación, describimos ambos operadores de cruce. El cruce aleatorio obtiene como nueva solución a uno de los padres trasladado en la dirección de los otros dos un factor F, bajo la recombinación binomial, es decir, dicha recombinación tendrá lugar solo bajo cierta probabilidad CR (por gen). Se han tomado $F = CR = 0,5$. También hay que tener en cuenta que la recombinación podría exceder el intervalo $[0, 1]$, por lo que puede ser necesario truncar.

```
#pi: Padre i-ésimo
#x: Individuo i-ésimo de la población
#best: Mejor solución
def DE_Rand(p1,p2,p3,x,best)
    s=solucion(tam = p1.size())

    for k from 1 to solucion.size()
        if aleatorioUniforme(0,1) < CR  # Recombinación
            s[k] = p1[k] + F * (p2[k] - p3[k])
            if s[k] < 0.0 then s[k] = 0.0
            if s[k] > 0.0 then s[k] = 1.0
        else  # Mantenemos atributo
            s[k] = x[k]
        end
    end
end
```

EL cruce *Current-to-best* se obtiene de trasladar el individuo de la población, primero en la dirección hacia la mejor solución, y luego en la dirección entre los dos padres que intervienen en este caso, ambas una cantidad F. De nuevo consideramos recombinación binomial, con $F = CR = 0,5$, y la necesidad de truncar si un atributo excede los límites.

```
#pi: Padre i-ésimo
#x: Individuo i-ésimo de la población
#best: Mejor solución
def DE_Rand(p1,p2,p3,x,best)
    s=solucion(tam = p1.size())

    for k from 1 to solucion.size()
        if aleatorioUniforme(0,1) < CR  # Recombinación
            s[k] = x[k] + F * (best[k] - x[k]) + F*(p1[k]-p2[k])
            if s[k] < 0.0 then s[k] = 0.0
            if s[k] > 0.0 then s[k] = 1.0
        else  # Mantenemos atributo
            s[k] = x[k]
        end
    end
end
```

Finalmente, el esquema de reemplazamiento utilizado es uno a uno, es decir, para cada individuo de la población, se compara con el hijo obtenido en esa misma posición y en la siguiente generación se queda el mejor de los dos. En este momento también nos aseguramos de que se actualiza la mejor solución de la población.

```
def reemplazo()
    for i from 1 to poblacion.size()
        if hijo[i].fitness > poblacion[i].fitness
            poblacion[i] = hijo[i]
            if hijo[i].fitness > mejorSolucion.fitness
                mejorSolucion = hijo[i]
            end
        end
    end
end
```

Por último, sobre las llamadas a la función objetivo, se realizan durante la fase de recombinación, tantas como hijos se hayan creado.

Algoritmo de comparación: RELIEF

El algoritmo utilizado para comparar con las heurísticas es el greedy RELIEF, con algunas modificaciones para satisfacer las restricciones de la solución. Este algoritmo parte de un vector inicial de pesos inicializado a 0, y para cada dato en la partición, busca los datos más cercanos a él de su misma clase (amigo más cercano) y de clase distinta (enemigo más cercano), respectivamente. Después, actualiza el vector de pesos sumando las distancias componente a componente con el enemigo más cercano y restando las distancias componente a componente con el amigo más cercano. Con esto se pretende dar mayor relevancia a las características que mejor separan los datos de clases distintas, y disminuir la importancia de las características que separan datos de la misma clase. Una vez actualizado el vector con todos los datos, para satisfacer las restricciones de la solución, las componentes negativas se hacen cero y se normaliza el vector con la norma del máximo. El pseudocódigo queda como sigue:

```
def RELIEF(particion)
    w = [0,...,0]
    for dato in particion
        amigo = amigoMasCercano(dato,particion)
        enemigo = enemigoMasCercano(dato,particion)

        for i from 1 to w.size()
            w[i] = w[i] - |dato[i] - amigo[i]| + |dato[i] - enemigo[i]|
        end
    end

    if w[i] < 0 then w[i] = 0 for i from 1 to w.size

    normalizar_max(w)

    return(w)

end
```

Los algoritmos para el amigo y el enemigo más cercanos son análogos, con la única diferencia de que el amigo más cercano no puede compararse con el propio dato. A continuación se muestran los pseudocódigos:

```
def amigoMasCercano(dato, particion)
    dist_mas_cercano = INF
    for elem in particion
        if elem != dato and elem.clase == dato.clase
            if dist(elem,dato) < dist_mas_cercano
                dist_mas_cercano = dist(elem,dato)
                amigo = elem
            end
        end
    end

    return amigo
end

def enemigoMasCercano(dato, particion)
    dist_mas_cercano = INF
    for elem in particion
```

```
        if elem.clase != dato.clase
            if dist(elem,dato) < dist_mas_cercano
                dist_mas_cercano = dist(elem,dato)
                enemigo = elem
            end
        end
    end

    return enemigo
end
```

Procedimiento considerado para desarrollar la práctica

Para el desarrollo de los distintos algoritmos de la práctica se ha elaborado un código propio en `C++`. Para la lectura de los ficheros arff se ha incorporado y arreglado un código C++ disponible en GitHub [2]. Los códigos disponen de un `makefile` que permite compilar todos los módulos automáticamente y de varios scripts de bash para tomar resultados. El ejecutable generado es `./bin/apc`. Los distintos problemas se encuentran en la carpeta `data`.

Para ejecutar el programa desde la línea de comandos se utiliza la sintaxis `./bin/apc [archivo del problema] [opciones]`. Las opciones disponibles son:

- `-a <algoritmo>` (**Necesaria**). Especifica el algoritmo a utilizar. Los algoritmos disponibles son:
  - `1NN`: Evalua el clasificador 1NN. Por defecto, sobre una solución constante 1. Se puede especificar otra solución con la opción `-w`.
  - `RANDOM`: Genera y evalúa soluciones aleatorias uniformemente distribuidas sobre [0,1].
  - `RELIEF`: Obtiene soluciones con el algoritmo RELIEF.
  - `RANDOM+LS`: Aplica la búsqueda local sobre soluciones iniciales aleatorias.
  - `RELIEF+LS`: Aplica la búsqueda local sobre soluciones iniciales RELIEF.
  - `AGG-BLX`: Obtiene soluciones con el AGG con operador de cruce BLX-0.3.
  - `AGG-CA`: Obtiene soluciones con el AGG con operador de cruce aritmético.
  - `AGE-BLX`: Obtiene soluciones con el AGE con operador de cruce BLX-0.3.
  - `AGE-BLX`: Obtiene soluciones con el AGE con operador de cruce aritmético.
  - `AM-10-1.0`: Algoritmo memético que aplica BL a todos los individuos cada 10 generaciones.
  - `AM-10-0.1`: Algoritmo memético que aplica BL a un 10 % de la población al azar cada 10 generaciones.
  - `AM-10-0.1mej`: Algoritmo memético que aplica BL al 10 % mejor de la población cada 10 generaciones.
  - `SA`: Enfriamiento simulado con esquema de Cauchy,
  - `ILS`: Obtiene soluciones con Búsqueda Local Reiterada.
  - `DE-RAND`: Algoritmo de evolución diferencial con cruce `Rand`.
  - `DE-CURRENTTOBEST`: Algoritmo de evolución diferencial con cruce `Current-To-Best`.
- `-m <modo>`: Modo de evaluación y particionado. Por defecto, `5FOLD`. Los modos disponibles son:
  - `5x2`: Evaluación con `1NN` y validación cruzada `5x2` (práctica 1).
  - `5FOLD`: Evaluación media de `1NN` y reducción, con validación cruzada 5-Fold (práctica 2).
- `-o <nombre salida>`: Especifica un nombre para los ficheros de salida con resultados que se crearán. Es necesario para utilizar las opciones `-p` y `-t`. Dependiendo de estas opciones, se crearán distintos ficheros con el nombre indicado y distintas extensiones añadidas por el programa.
- `-p <string>`: Indica qué datos serán imprimidos en ficheros. Cada carácter en `string` indica un tipo de dato a imprimir. Los datos admitidos son:
  - `f`: Se imprimirá un archivo con los fitness obtenidos (`.fit`).
  - `p`: Se imprimirá un archivo con los índices de cada partición utilizada (`.part`).
  - `t`: Se imprimirá un archivo con los tiempos obtenidos (`.time`).
  - `i`: Se imprimirá un archivo con los fitness obtenidos sobre la partición de entrenamiento (`.trfit`).
  - `s`: Se imprimirá un archivo con las soluciones obtenidas (`.sol`).
- `-s <semilla>`: Especifica una semilla para generar números aleatorios con la que ejecutar el programa.
- `-t <string>`: Se creará una tabla (`.table`) con los datos indicados en `string`. Cada carácter en `string` indica un tipo de dato a imprimir. Los datos admitidos son los mismos que en la

opción `-p`, a excepción de `s` y `p`.

- `-w <fichero solucion>`: Especifica un fichero donde hay almacenada una solución para clasificar con ella. Solo se tendrá en cuenta si el algoritmo es 1NN.

En la práctica, el uso del programa se reduce a la llamada `./bin/apc <nombre problema> -a <algoritmo> -s <semilla>`. Un ejemplo de uso del programa para tomar resultados es `./bin/apc ./data/sonar.arff -a RELIEF -s 3 -o ./sol/RELIEF_sonar_3 -t fti -p sp`.

Finalmente, se proporcionan los siguientes scripts de bash:

- `./sh/exec.sh`. Dado un directorio, pasado como argumento, ejecuta todos los algoritmos con todos los problemas y guarda los resultados en el directorio. Se puede modificar para ejecutar solo determinados problemas con determinados algoritmos, y para las semillas que se deseen.
- `./sh/calcAvg.sh`. Dado un directorio, pasado como argumento, lee las soluciones encontradas en ese directorio y genera ficheros con las tablas de datos medios obtenidos para cada algoritmo en cada problema. Los ficheros resultantes tienen la forma `means_$SEMILLA.table`.
- `./sh/start.sh`. Genera un nuevo directorio basado en la fecha de la ejecución y llama a los dos scripts anteriores para tomar resultados.

Experimentos y análisis de resultados

Resultados obtenidos.

Todas las ejecuciones que se muestran de ahora en adelante se han realizado sobre un ordenador HP con las siguientes características:

- Procesador Intel(R) Core(TM) i7
- Frecuencia del procesador: 2.8 GHz
- 4 procesadores principales, 8 procesadores lógicos
- 16 GB de RAM.

Las ejecuciones se han realizado sobre el sistema operativo Windows 7, a través de la herramienta `Cygwin` [1], que proporciona funcionalidades para Windows similares a las de las distribuciones de Linux. Finamente, el código `C++` utilizado ha sido compilado con optimización `-O2`.

Para la toma de resultados se ha utilizado el script `start.sh` mencionado en la sección anterior, y la semilla utilizada ha sido 3141592.

Para cada problema y cada algoritmo se han tomado los siguientes datos: fitness sobre la muestra de entrenamiento, fitness sobre los datos test, tasas de clasificación y reducción sobre el test y tiempo de ejecución. Los resultados obtenidos son:

1NN	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	84.6154	-	0.0020	95.7747	-	0.0070	78.2609	-	0.0070
PARTITION (0,1)	84.6154	-	0.0020	94.7368	-	0.0060	84.7826	-	0.0070
PARTITION (1,0)	81.7308	-	0.0010	95.0704	-	0.0060	87.3913	-	0.0080
PARTITION (1,1)	87.5000	-	0.0020	96.4912	-	0.0060	82.6087	-	0.0070
PARTITION (2,0)	86.5385	-	0.0020	95.0704	-	0.0070	82.6087	-	0.0070
PARTITION (2,1)	83.6538	-	0.0020	94.0351	-	0.0070	84.3478	-	0.0070
PARTITION (3,0)	85.5769	-	0.0010	94.7183	-	0.0060	77.8261	-	0.0070
PARTITION (3,1)	82.6923	-	0.0010	97.1930	-	0.0060	88.2609	-	0.0070
PARTITION (4,0)	86.5385	-	0.0010	96.1268	-	0.0070	86.0870	-	0.0060
PARTITION (4,1)	80.7692	-	0.0020	93.6842	-	0.0060	78.6957	-	0.0070
MEAN	**84.4231**	-	**0.0016**	**95.2901**	-	**0.0064**	**83.0870**	-	**0.0070**
STDEV	*2.0978*	-	*0.0005*	*1.0440*	-	*0.0005*	*3.5982*	-	*0.0004*

Figura 1: Resultados de la ejecución del clasificador 1NN con pesos 1.

RANDOM	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	86.5385	83.6538	0.0000	95.0704	95.4386	0.0000	79.5652	81.3043	0.0000
PARTITION (0,1)	86.5385	86.5385	0.0000	95.4386	95.4225	0.0000	83.0435	78.6957	0.0000
PARTITION (1,0)	82.6923	88.4615	0.0000	94.7183	96.1404	0.0000	87.8261	85.2174	0.0000
PARTITION (1,1)	85.5769	81.7308	0.0000	95.4386	95.0704	0.0000	86.0870	85.6522	0.0000
PARTITION (2,0)	87.5000	82.6923	0.0000	95.4225	94.0351	0.0000	83.0435	82.1739	0.0000
PARTITION (2,1)	82.6923	88.4615	0.0000	93.6842	94.7183	0.0000	84.7826	83.4783	0.0000
PARTITION (3,0)	85.5769	83.6538	0.0000	94.7183	98.5965	0.0000	79.5652	85.6522	0.0000
PARTITION (3,1)	88.4615	85.5769	0.0000	98.2456	94.7183	0.0000	85.2174	77.3913	0.0000
PARTITION (4,0)	84.6154	80.7692	0.0000	96.1268	93.3333	0.0000	86.0870	83.4783	0.0000
PARTITION (4,1)	79.8077	83.6538	0.0000	94.7368	96.1268	0.0000	81.7391	85.6522	0.0000
MEAN	**85.0000**	**84.5192**	**0.0000**	**95.3600**	**95.3600**	**0.0000**	**83.6957**	**82.8696**	**0.0000**
STDEV	*2.4777*	*2.5237*	*0.0000*	*1.1414*	*1.3578*	*0.0000*	*2.6607*	*2.8258*	*0.0000*

Figura 2: Resultados de la ejecución del clasificador 1NN con soluciones aleatorias.

RANDOM+LS	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	85.5769	93.2692	2.8232	95.4225	95.7895	3.9642	80.8696	94.7826	11.4529
PARTITION (0,1)	84.6154	89.4231	2.1021	95.0877	96.1268	3.9182	83.0435	87.8261	18.2042
PARTITION (1,0)	82.6923	91.3462	1.9061	94.7183	97.8947	6.2130	89.1304	93.9130	11.1998
PARTITION (1,1)	84.6154	87.5000	2.5421	96.4912	96.4789	3.9072	85.2174	90.8696	16.2349
PARTITION (2,0)	86.5385	87.5000	2.5401	95.4225	96.4912	5.9803	83.9130	88.6957	11.5367
PARTITION (2,1)	80.7692	92.3077	2.1731	94.3860	97.1831	10.4228	81.3043	90.4348	10.7248
PARTITION (3,0)	85.5769	88.4615	1.9071	95.4225	98.9474	5.2383	77.3913	91.7391	15.9139
PARTITION (3,1)	88.4615	88.4615	2.0851	97.8947	96.4789	6.1093	87.3913	89.1304	10.3066
PARTITION (4,0)	84.6154	87.5000	1.9951	96.1268	95.4386	4.7023	83.0435	91.7391	12.3069
PARTITION (4,1)	79.8077	86.5385	1.9947	94.7368	97.1831	4.8633	83.4783	90.8696	12.4165
MEAN	**84.3269**	**89.2308**	**2.2069**	**95.5709**	**96.8012**	**5.5319**	**83.4783**	**91.0000**	**13.0297**
STDEV	*2.4723*	*2.1843*	*0.3001*	*0.9843*	*0.9867*	*1.8396*	*3.1535*	*2.0764*	*2.5896*

Figura 3: Resultados de la ejecución de la búsqueda local partiendo de soluciones aleatorias.

RELIEF	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	81.7308	83.6538	0.0020	95.4225	94.7368	0.0110	82.1739	89.1304	0.0130
PARTITION (0,1)	82.6923	92.3077	0.0030	95.4386	95.4225	0.0120	87.3913	85.6522	0.0130
PARTITION (1,0)	77.8846	88.4615	0.0030	95.0704	97.5439	0.0120	88.6957	86.9565	0.0120
PARTITION (1,1)	88.4615	86.5385	0.0030	95.7895	96.4789	0.0130	83.9130	87.8261	0.0130
PARTITION (2,0)	85.5769	82.6923	0.0020	95.4225	94.7368	0.0130	86.9565	87.8261	0.0130
PARTITION (2,1)	80.7692	89.4231	0.0030	94.3860	95.0704	0.0120	84.3478	86.9565	0.0140
PARTITION (3,0)	81.7308	90.3846	0.0030	95.7747	98.5965	0.0120	80.4348	90.8696	0.0130
PARTITION (3,1)	86.5385	84.6154	0.0030	97.5439	96.1268	0.0120	90.0000	85.2174	0.0130
PARTITION (4,0)	84.6154	83.6538	0.0030	96.1268	93.6842	0.0130	86.9565	84.7826	0.0130
PARTITION (4,1)	79.8077	89.4231	0.0030	94.7368	96.1268	0.0120	82.1739	90.0000	0.0130
MEAN	**82.9808**	**87.1154**	**0.0028**	**95.5712**	**95.8524**	**0.0122**	**85.3043**	**87.5217**	**0.0130**
STDEV	*3.1024*	*3.1657*	*0.0004*	*0.8185*	*1.3754*	*0.0006*	*2.9922*	*1.9251*	*0.0004*

Figura 4: Resultados de la ejecución del greedy RELIEF.

RELIEF+LS	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	84.6154	89.4231	3.4172	95.0704	97.5439	5.3803	80.0000	93.4783	12.3333
PARTITION (0,1)	82.6923	93.2692	1.9071	95.4386	96.8310	3.9622	86.9565	92.6087	13.1448
PARTITION (1,0)	77.8846	93.2692	2.8422	95.7747	98.5965	4.3462	86.9565	95.6522	17.9072
PARTITION (1,1)	86.5385	90.3846	2.0301	95.7895	96.4789	3.9222	85.2174	95.6522	16.2541
PARTITION (2,0)	85.5769	91.3462	2.7822	95.0704	95.7895	4.1162	88.2609	91.7391	9.1325
PARTITION (2,1)	82.6923	93.2692	1.9631	94.3860	96.4789	4.0806	85.2174	94.3478	15.4667
PARTITION (3,0)	85.5769	93.2692	2.1281	94.7183	99.6491	5.1413	80.4348	95.2174	11.5189
PARTITION (3,1)	83.6538	90.3846	3.3502	97.1930	97.1831	4.7029	86.0870	93.0435	15.3581
PARTITION (4,0)	84.6154	92.3077	2.8492	97.1831	96.1404	4.4053	86.9565	95.6522	13.0017
PARTITION (4,1)	78.8462	92.3077	2.1751	95.0877	96.8310	5.1279	80.0000	93.4783	10.2636
MEAN	**83.2692**	**91.9231**	**2.5444**	**95.5712**	**97.1522**	**4.5185**	**84.6087**	**94.0870**	**13.4381**
STDEV	*2.7264*	*1.3732*	*0.5447*	*0.9057*	*1.1172*	*0.5096*	*3.0447*	*1.3496*	*2.6363*

Figura 5: Resultados de la ejecución de la búsqueda local partiendo de soluciones RELIEF.

AGG-BLX	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	87.5000	92.3077	22.6123	95.7747	97.1930	93.5869	79.5652	93.4783	106.1087
PARTITION (0,1)	85.5769	95.1923	22.7039	94.7368	97.8873	93.1515	84.3478	91.3043	105.1876
PARTITION (1,0)	79.8077	94.2308	22.6945	95.7747	98.5965	93.8880	86.9565	93.9130	105.5034
PARTITION (1,1)	82.6923	89.4231	22.6489	95.4386	96.8310	92.3849	86.0870	95.2174	105.5512
PARTITION (2,0)	86.5385	92.3077	22.6425	96.1268	96.8421	93.2345	83.0435	93.9130	104.7712
PARTITION (2,1)	80.7692	93.2692	22.7259	94.7368	97.1831	92.4831	83.9130	90.4348	105.6676
PARTITION (3,0)	80.7692	94.2308	22.6971	92.6056	99.2982	93.2855	80.8696	94.7826	104.7234
PARTITION (3,1)	84.6154	93.2692	22.8133	97.8947	97.1831	93.0517	90.8696	93.4783	105.1096
PARTITION (4,0)	82.6923	91.3462	22.7509	96.4789	97.8947	93.4761	82.6087	92.6087	104.4376
PARTITION (4,1)	80.7692	92.3077	22.7483	94.3860	97.5352	93.2667	81.7391	93.0435	104.3486
MEAN	**83.1731**	**92.7885**	**22.7038**	**95.3954**	**97.6444**	**93.1809**	**84.0000**	**93.2174**	**105.1409**
STDEV	*2.5889*	*1.5650*	*0.0574*	*1.3387*	*0.7565*	*0.4375*	*3.1280*	*1.3912*	*0.5431*

Figura 6: Resultados de la ejecución del algoritmo AGG-BLX.

AGG-CA	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	88.4615	91.3462	22.4405	95.7747	97.1930	92.9089	80.8696	91.7391	104.4944
PARTITION (0,1)	86.5385	91.3462	22.5593	95.4386	97.8873	92.1147	79.1304	89.1304	103.8085
PARTITION (1,0)	80.7692	91.3462	22.5705	95.0704	97.8947	92.8999	86.5217	93.0435	104.4700
PARTITION (1,1)	82.6923	91.3462	22.5653	95.7895	97.1831	92.1055	81.7391	92.6087	103.9589
PARTITION (2,0)	85.5769	88.4615	22.4945	95.7747	96.1404	92.8001	83.4783	92.1739	104.0267
PARTITION (2,1)	83.6538	92.3077	22.5983	94.3860	97.1831	92.1977	86.9565	91.3043	104.2716
PARTITION (3,0)	87.5000	91.3462	22.5965	94.7183	98.5965	92.9017	77.8261	93.9130	103.9753
PARTITION (3,1)	85.5769	91.3462	22.6115	96.4912	97.5352	92.1139	88.2609	89.1304	103.8397
PARTITION (4,0)	84.6154	87.5000	22.6023	96.4789	96.8421	92.9637	83.9130	90.4348	103.7347
PARTITION (4,1)	79.8077	89.4231	22.4487	95.0877	97.5352	92.1009	82.1739	90.8696	103.9061
MEAN	**84.5192**	**90.5769**	**22.5487**	**95.5010**	**97.3991**	**92.5107**	**83.0870**	**91.4348**	**104.0486**
STDEV	*2.6663*	*1.4774*	*0.0610*	*0.6614*	*0.6312*	*0.3865*	*3.2445*	*1.5068*	*0.2558*

Figura 7: Resultados de la ejecución del algoritmo AGG-CA.

AGE-BLX	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	86.5385	93.2692	22.5713	95.0704	98.2456	96.3215	82.1739	94.3478	104.0203
PARTITION (0,1)	84.6154	96.1538	22.6819	94.3860	98.5916	95.5692	83.0435	90.8696	103.6053
PARTITION (1,0)	83.6538	93.2692	22.6673	95.0704	98.5965	96.4183	84.3478	94.3478	103.9345
PARTITION (1,1)	85.5769	89.4231	22.5983	95.7895	97.5352	95.3526	85.6522	94.7826	104.1557
PARTITION (2,0)	88.4615	92.3077	22.4833	94.7183	97.1930	96.1473	84.3478	93.4783	104.3246
PARTITION (2,1)	82.6923	93.2692	22.4803	95.4386	97.5352	95.6195	85.6522	93.0435	104.0135
PARTITION (3,0)	86.5385	94.2308	22.5017	92.9577	99.2982	95.9973	80.8696	95.2174	104.3396
PARTITION (3,1)	87.5000	92.3077	22.5091	96.8421	98.2394	95.8049	88.2609	90.0000	104.0411
PARTITION (4,0)	84.6154	89.4231	22.5435	97.5352	96.8421	96.1561	89.1304	90.8696	104.1037
PARTITION (4,1)	77.8846	94.2308	22.5399	93.3333	97.5352	95.7157	86.0870	93.0435	103.9285
MEAN	**84.8077**	**92.7885**	**22.5576**	**95.1142**	**97.9612**	**95.9102**	**84.9565**	**93.0000**	**104.0467**
STDEV	*2.8459*	*1.9821*	*0.0683*	*1.3387*	*0.7167*	*0.3351*	*2.4376*	*1.7332*	*0.2025*

Figura 8: Resultados de la ejecución del algoritmo AGE-BLX.

AGE-CA	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	88.4615	91.3462	22.6333	95.7747	97.1930	93.1073	80.8696	91.3043	104.3914
PARTITION (0,1)	85.5769	94.2308	22.5447	95.4386	97.8873	92.8477	87.8261	87.8261	104.2015
PARTITION (1,0)	81.7308	92.3077	22.4591	95.0704	97.8947	93.1789	86.9565	92.1739	104.2979
PARTITION (1,1)	87.5000	85.5769	22.4105	96.1404	96.8310	92.5959	86.9565	93.4783	103.9761
PARTITION (2,0)	85.5769	87.5000	22.4425	94.7183	96.8421	93.1849	82.6087	90.4348	103.7227
PARTITION (2,1)	82.6923	91.3462	22.3773	94.0351	96.4789	92.6787	84.7826	92.1739	103.8217
PARTITION (3,0)	84.6154	89.4231	22.4513	94.7183	99.2982	93.3817	81.3043	93.9130	103.6739
PARTITION (3,1)	82.6923	88.4615	22.4715	96.8421	97.1831	92.2869	83.9130	87.8261	103.5569
PARTITION (4,0)	83.6538	88.4615	22.5023	96.8310	96.8421	93.1377	85.6522	89.1304	103.8259
PARTITION (4,1)	76.9231	94.2308	22.5309	94.3860	97.1831	92.5539	82.1739	91.3043	103.7019
MEAN	**83.9423**	**90.2885**	**22.4823**	**95.3955**	**97.3634**	**92.8954**	**84.3043**	**90.9565**	**103.9170**
STDEV	*3.1024*	*2.7347*	*0.0703*	*0.9354*	*0.7750*	*0.3351*	*2.3851*	*2.0375*	*0.2724*

Figura 9: Resultados de la ejecución del algoritmo AGE-CA.

AM-10-1.0	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	91.3462	91.3462	23.3043	94.3662	97.1930	99.0971	77.8261	91.7391	111.4976
PARTITION (0,1)	80.7692	91.3462	23.3325	94.3860	97.5352	97.9992	83.9130	91.7391	111.2420
PARTITION (1,0)	79.8077	94.2308	23.6554	95.0704	97.8947	98.7500	87.3913	92.6087	111.5864
PARTITION (1,1)	89.4231	88.4615	23.7456	96.8421	97.5352	97.8230	84.7826	94.7826	111.3590
PARTITION (2,0)	82.6923	88.4615	23.6495	95.7747	96.1404	98.6662	84.3478	93.0435	111.7230
PARTITION (2,1)	83.6538	93.2692	23.4703	93.6842	96.8310	98.0182	82.1739	90.8696	111.1750
PARTITION (3,0)	86.5385	92.3077	23.4387	94.0141	98.5965	98.5492	78.2609	93.9130	110.8261
PARTITION (3,1)	77.8846	91.3462	23.3883	96.8421	96.8310	98.3356	86.9565	92.1739	111.1239
PARTITION (4,0)	82.6923	91.3462	23.4265	96.1268	97.1930	98.8547	87.3913	93.4783	111.0587
PARTITION (4,1)	82.6923	94.2308	23.4625	94.0351	98.2394	98.4934	78.6957	92.6087	111.5810
MEAN	**83.7500**	**91.6346**	**23.4874**	**95.1142**	**97.3989**	**98.4587**	**83.1739**	**92.6957**	**111.3173**
STDEV	*3.9982*	*1.9256*	*0.1400*	*1.1379*	*0.6875*	*0.3903*	*3.5804*	*1.0960*	*0.2652*

Figura 10: Resultados de la ejecución del algoritmo AM-10-1.0.

AM-10-0.1	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	91.3462	91.3462	23.0137	94.3662	97.1930	94.3202	77.8261	92.1739	104.4748
PARTITION (0,1)	84.6154	93.2692	23.1109	93.6842	98.2394	96.4659	84.3478	91.3043	104.5540
PARTITION (1,0)	81.7308	92.3077	23.3279	96.4789	98.2456	96.1431	88.2609	93.4783	104.6570
PARTITION (1,1)	84.6154	90.3846	22.9519	95.7895	97.1831	93.2429	87.3913	95.6522	104.4356
PARTITION (2,0)	85.5769	89.4231	22.9709	95.4225	97.1930	93.6291	84.7826	92.1739	104.4254
PARTITION (2,1)	86.5385	93.2692	23.0279	95.7895	96.8310	92.9773	82.6087	91.7391	104.4736
PARTITION (3,0)	83.6538	92.3077	23.0555	93.3099	99.2982	93.7988	77.8261	95.2174	104.7174
PARTITION (3,1)	83.6538	93.2692	23.1197	97.1930	96.4789	93.1879	86.0870	91.7391	104.4266
PARTITION (4,0)	83.6538	87.5000	22.9983	95.7747	96.1404	94.0910	83.0435	91.7391	104.6020
PARTITION (4,1)	79.8077	91.3462	22.8473	94.0351	97.1831	93.1179	83.9130	92.1739	104.2620
MEAN	**84.5192**	**91.4423**	**23.0424**	**95.1843**	**97.3986**	**94.0974**	**83.6087**	**92.7391**	**104.5028**
STDEV	*2.8989*	*1.7965*	*0.1210*	*1.2079*	*0.8927*	*1.1801*	*3.3512*	*1.4561*	*0.1250*

Figura 11: Resultados de la ejecución del algoritmo AM-10-0.1.

AM-10-0.1mej	SONAR			WDBC			SPAMBASE		
	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME	FITNESS	TRAIN FIT	TIME
PARTITION (0,0)	91.3462	91.3462	22.8071	94.3662	97.1930	94.1414	77.3913	91.3043	104.7236
PARTITION (0,1)	83.6538	92.3077	22.8595	95.4386	97.8873	93.7273	83.9130	90.4348	104.2103
PARTITION (1,0)	84.6154	94.2308	22.8755	94.3662	97.8947	94.4024	87.3913	92.6087	104.8000
PARTITION (1,1)	80.7692	88.4615	22.9075	95.7895	96.8310	93.8053	86.0870	94.7826	105.7682
PARTITION (2,0)	86.5385	88.4615	22.8423	95.4225	97.5439	94.5738	83.0435	91.7391	104.2218
PARTITION (2,1)	83.6538	92.3077	22.8663	96.1404	97.5352	93.6591	83.9130	91.7391	104.4366
PARTITION (3,0)	81.7308	94.2308	22.8215	93.3099	99.2982	94.4306	82.1739	93.0435	104.5856
PARTITION (3,1)	83.6538	92.3077	22.8883	97.1930	97.5352	93.6405	88.6957	87.8261	104.3864
PARTITION (4,0)	87.5000	87.5000	22.8003	95.4225	97.1930	94.4080	85.6522	91.3043	104.2167
PARTITION (4,1)	75.9615	92.3077	22.8159	95.0877	97.8873	93.7040	83.0435	93.0435	104.5244
MEAN	**83.9423**	**91.3462**	**22.8484**	**95.2536**	**97.6799**	**94.0492**	**84.1304**	**91.7826**	**104.5874**
STDEV	*3.9188*	*2.2755*	*0.0358*	*1.0150*	*0.6328*	*0.3590*	*2.9887*	*1.7550*	*0.4386*

Figura 12: Resultados de la ejecución del algoritmo AM-10-0.1mej.

SEED-3141592	sonar			wdbc			spambase-460		
ALGORITHMS	**FITNESS**	**TRAIN**	**TIME**	**FITNESS**	**TRAIN**	**TIME**	**FITNESS**	**TRAIN**	**TIME**
1NN	84.4231	-	0.0016	95.2901	-	0.0064	83.0870	-	0.0070
RANDOM	85.0000	84.5192	0.0000	95.3600	95.3600	0.0000	83.6957	82.8696	0.0000
RELIEF	82.9808	87.1154	0.0028	95.5712	95.8524	0.0122	85.3043	87.5217	0.0130
RELIEF+LS	83.2692	91.9231	2.5444	95.5712	97.1522	4.5185	84.6087	94.0870	13.4381
RANDOM+LS	84.3269	89.2308	2.2069	95.5709	96.8012	5.5319	83.4783	91.0000	13.0297
AGG-BLX	83.1731	92.7885	22.7038	95.3954	97.6444	93.1809	84.0000	93.2174	105.1409
AGG-CA	84.5192	90.5769	22.5487	95.5010	97.3991	92.5107	83.0870	91.4348	104.0486
AGE-BLX	84.8077	92.7885	22.5576	95.1142	97.9612	95.9102	84.9565	93.0000	104.0467
AGE-CA	83.9423	90.2885	22.4823	95.3955	97.3634	92.8954	84.3043	90.9565	103.9170
AM-10-1.0	83.7500	91.6346	23.4874	95.1142	97.3989	98.4587	83.1739	92.6957	111.3173
AM-10-0.1	84.5192	91.4423	23.0424	95.1843	97.3986	94.0974	83.6087	92.7391	104.5028
AM-10-0.1mej	83.9423	91.3462	22.8484	95.2536	97.6799	94.0492	84.1304	91.7826	104.5874

Figura 13: Resultados medios y comparación conjunta de todos los algoritmos.

Valoración general de los resultados.

En primer lugar discutimos las diferencias en lo respectivo al aprendizaje con el modelo de validación de la práctica anterior. En esta ocasión si se aprecian variaciones mayores entre los distintos modelos en lo que respecta a las tasas sobre los datos test. Además, estas tasas, en general, son parecidas a las obtenidas sobre los datos de entrenamiento. Podemos concluir que se ha conseguido reducir el sobreaprendizaje. Esto se debe en parte a que el modelo de validación 5-Fold presenta menos varianza que las particiones al 50 % realizadas con 5x2, y en parte también a que la tasa de reducción influye bastante y permite eliminar muchas características, en general, haciendo que el clasificador aprenda con menos características y en consecuencia reduciendo la capacidad de ajuste a los datos. Aun así, en los algoritmos que consiguen alcanzar valores por encima del 85-90 % sobre la muestra de entrenamiento, se aprecia (sobre todo en `sonar`) que las evaluaciones de los datos test no consiguen subir tanto, quedándose estancadas en torno al 80-83 %. Es decir, se sigue produciendo algo de sobreaprendizaje, especialmente en `sonar`, que tienen menor número de datos que los otros dos problemas. De hecho, comparando las diferencias en las tasas de clasificación las diferencias son aún mayores (las tasas de clasificación sobre train no se muestran por espacio, pero como la tasa de reducción es fija, se puede deducir como $clas\_train = 2 \cdot train\_fit - tasa\_red$). En este caso, en sonar vemos que, por ejemplo, en enfriamiento simulado, la tasa de clasificación en train supera el 94 %, quedándose en 80 % en test. Con ILS y evolución diferencial ocurre igual. En `spambase` estos mismos algoritmos también reducen la tasa de clasificación sobre test, pero las diferencias no son tan abultadas.

En cuanto a los conjuntos de datos, podemos sacar las mismas conclusiones que en la práctica anterior. Se observa que el conjunto que mejores resultados da es `wdbc`. Los otros dos proporcionan

resultados similares, y bastante inferiores. Esto puede deberse al menor número de características en `wdbc` en parte, y también a que el origen de los distitntos conjuntos de datos puede producir más o menos ruidos. Por ejemplo, en `sonar` los datos clasificados se obtienen mediante señales obtenidas sobre distintos materiales, y estos pueden verse afectados por numerosas condiciones externas. En `spambase` puede introducirse un ruido similar, ya que un mismo correo podría considerarse spam o no según el destinatario. En `wdbc` puede haber menos ruido debido a que los datos se toman de imágenes de células, en las que en principio puede ser menos conflicitvo medir sus características.

En lo relativo a los tiempos, el algoritmo greedy sigue siendo el más rápido, como era de esperar, y el tiempo obtenido por el clasificador greedy nos da una medida del tiempo que tarda en evaluarse cada llamada a la función objetivo. Fijadas las 15000 iteraciones que utilizan los distintos algoritmos, vemos que, salvo búsqueda local y enfriamiento simulado, el resto requieren de tiempos parecidos para evaluarse, y esto es debido a que prácticamente todo el tiempo de estos algoritmos es consumido evaluando la función objetivo. La búsqueda local tarda menos tiempo porque termina antes de las 15000 iteraciones por el criterio de vecinos encontrados sin éxito. En cambio, el enfriamiento simulado termina antes porque el criterio de enfriamiento acepta muchas soluciones, y el algoritmo termina alcanzando la temperatura final algo antes de las 15000 iteraciones, con el esquema de Cauchy. Por otra parte, al añadir al criterio de aceptación la condición de ignorar soluciones que no han variado su coste, el número de evaluaciones realizadas no es en ninguno de los problemas menor que la mitad del esperado. Sin esta condición se aceptan muchas más soluciones, haciendo que en ‘sonar, por ejemplo, apenas se tarden 2 segundos.

Finalmente, en cuanto a la capacidad de aprendizaje de cada algoritmo, vemos que, en general, las tasas de clasificación del RELIEF y el 1NN ya son elevadas de por sí, y solo algunos algoritmos la superan ligeramente, según el problema. Sin embargo, apenas consiguen reducción. Esto es obvio para el 1NN y, en el caso del RELIEF, es un algoritmo determinista y que no está ideado para reducir, luego su tasa de reducción es muy baja. La poca diferencia entre estos dos algoritmos y los demás en cuanto a clasificación nos indica que los algoritmos heurísticos se centran sobre todo en aumentar la tasa de reducción. Vemos que, salvo `DE-RAND`, los algoritmos no evolutivos `ILS` y `SA` son los que consiguen mayores reducciones. Esto puede deberse a que implícitamente tienen una componente de búsqueda local, luego los resultados que consiguen son al menos los de la búsqueda local, y estos resultados superan en reducción a los evolutivos salvo `DE-RAND`. El memético se encuentra en esta misma situación, aunque sin llegar a los resultados de `ILS` y `SA`. Finalmente, el que claramente consigue reducir más es la evolución diferencial con cruce aleatorio, aunque también suele clasificar ligeramente peor a los demás sobre el test. Aunque en tasas de clasificación hay pocas diferencias y los resultados varían con el problema. En agregado, `DE-RAND` proporciona mejores resultados en todos los problemas gracias a su reducción.

En los siguientes análisis, si no se indica lo contrario, todas las ejecuciones realizadas han sido sobre el problema `sonar` con la semilla `3141592`.

Análisis de la evolución del enfriamiento simulado.

Lo que distingue al enfriamieno simulado de los demás algoritmos es su capacidad de poder ir aceptando soluciones peores a la actual sobre las que seguir trabajando. También este número de aceptaciones disminuye conforme se avanza en el algoritmo (va disminuyendo la temperatura). Esto se puede observar gráficamente. En la siguiente gráfica se muestra la evolución de las tasas de la solución que va modificando el algoritmo (en esta gráfica y en las sucesivas, en rojo se muestra la tasa de clasificación, en azul la de reducción y en morado el agregado durante el aprendizaje, es decir, sobre el train).

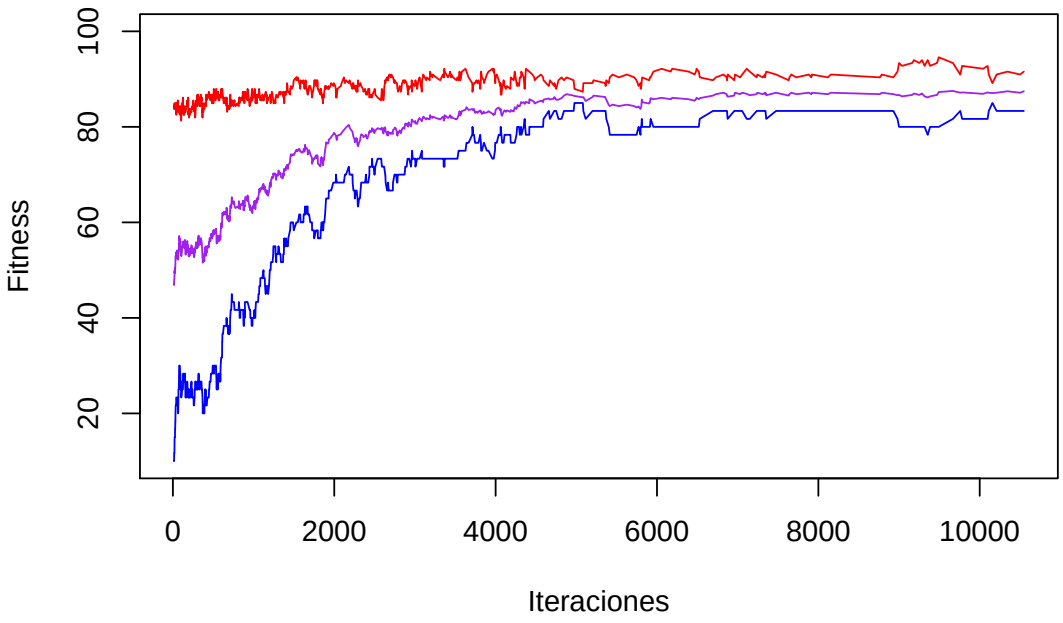


Figura 14: Evolución de la función objetivo en función del número de iteraciones en enfriamiento simulado.

Vemos como, durante las primeras iteraciones, se aprecian muchas subidas y bajadas en el agregado. La diferencia en estas subidas y bajadas se va suavizando conforma avanzan las iteraciones, hasta que se acaba estabilizando en las iteraciones finales, cuando ya la temperatura es muy baja. También vemos que bajadas grandes luego permiten subir una mayor cantidad, lo que justifica el uso de este algoritmo: aceptar soluciones peores para poder explorar distintas zonas del espacion de búsqueda y evitar óptimos locales. También se observa que, aunque se estabilice el agregado por el final, las otras tasas sí que presentan alguna variación importante, pero lo que se está optimizando es el agregado, así que es algo razonable.

También podemos ver que, mientras que la tasa de clasificación apenas varía en un pequeño intervalo, la tasa de reducción sube una gran cantidad a lo largo del algoritmo. Esto se puede apreciar en todas las gráficas que se verán más adelante, lo que nos confirma que los algoritmos heurísticos se centran en aumentar la reducción.

Análisis de la evolución en ILS

La principal característica del ILS es que, cada cierto número de iteraciones, la solución con la que se trabaja es mutada con un cambio brusco. Mientras, el resto del tiempo se está haciendo búsqueda local. El cambio brusco es el que nos permite explorar nuevas zonas en el espacio de búsqueda, y a priori no nos garantizan que mejoren la solución actual. De hecho, lo normal es que no ocurra, pero la búsqueda local sobre esta nueva solución nos puede conducir a mejores soluciones en zonas distintas del espacio de búsqueda. En la siguiente gráfica se muestra la evolución del ILS a lo largo del número de iteraciones.

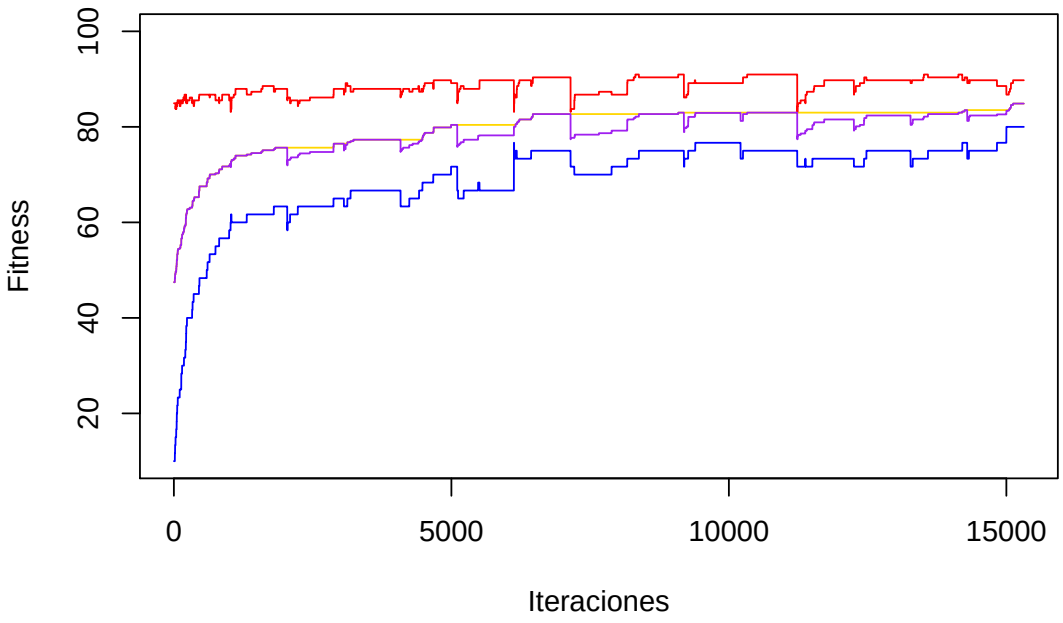


Figura 15: Evolución de la función objetivo en función del número de iteraciones en ILS

Se aprecia claramente el comportamiento que habíamos indicado. el algoritmo va aplicando búsqueda local, que va aumentando el valor de la función objetivo, pero cada cierto número de iteraciones hay una bajada. Esta se debe a la mutación brusca. En ocasiones, la búsqueda local sobre la mutación no consigue mejorar la solución actual, por lo que se vuelve a ella y se prueba con otra mutación, pero otras veces sí consigue superar el valor de la solución anterior (marcado en amarillo), y obteniendo así una nueva solución más prometedora sobre la que trabajar.

Análisis de la evolución diferencial.

En las siguientes gráficas se muestra la evolución de las tasas en función de las generaciones en ambos algoritmos de evolución diferencial. Se observa que con la recombinación `RAND` hay una gran subida de reducción y se aprecia mucha variabilidad hasta casi el final del algoritmo, mientras que en `CurrentToBest` la solución se estabiliza enseguida.

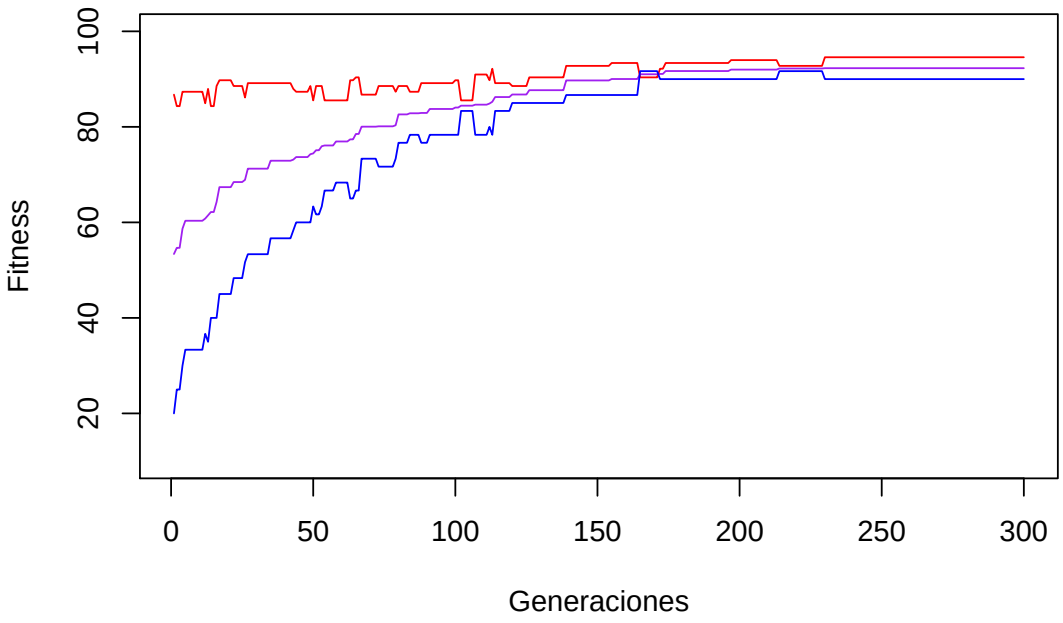


Figura 16: Evolución de la función objetivo en función del número de generaciones en DE-RAND

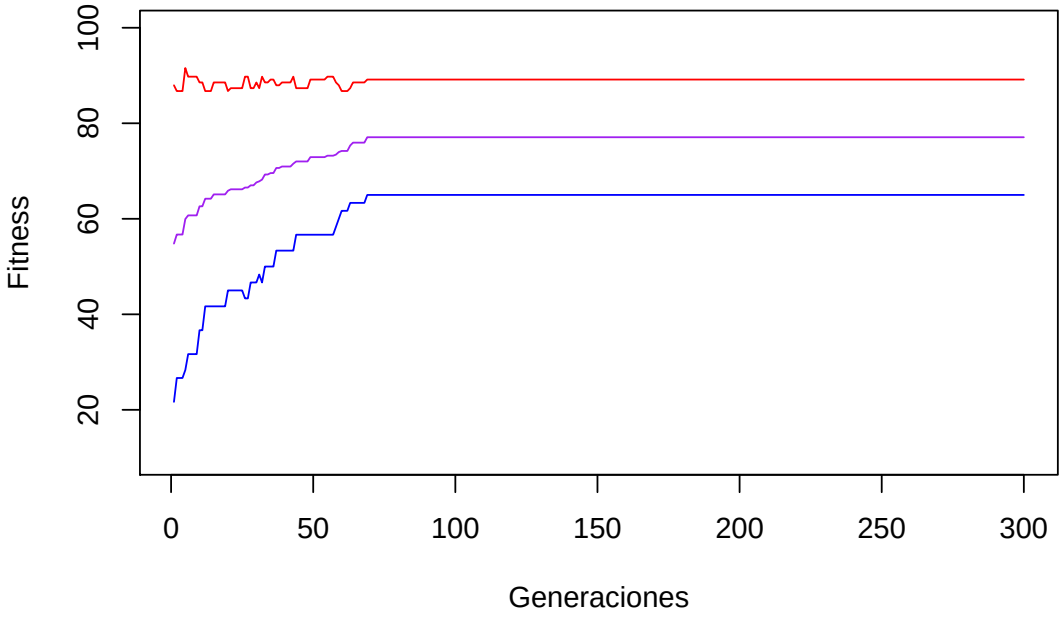


Figura 17: Evolución de la función objetivo en función del número de generaciones en DE-CurrentToBest

Comparación de los operadores de recombinación

Para estudiar las diferencias entre ambos algoritmos de evolución diferencial, estudiamos lo que realmente los distingue, el operador de recombinación. Para ello, ejecutamos el algoritmo de evolución diferencial y miramos cómo evoluciona la mejor solución en cada generación. En las siguientes gráficas se muestran, por superposición y por separado, la evolución de las mejores soluciones con el paso de las generaciones en `DE-RAND` (se muestra solo un número pequeño de mejoras, para que sea representable).

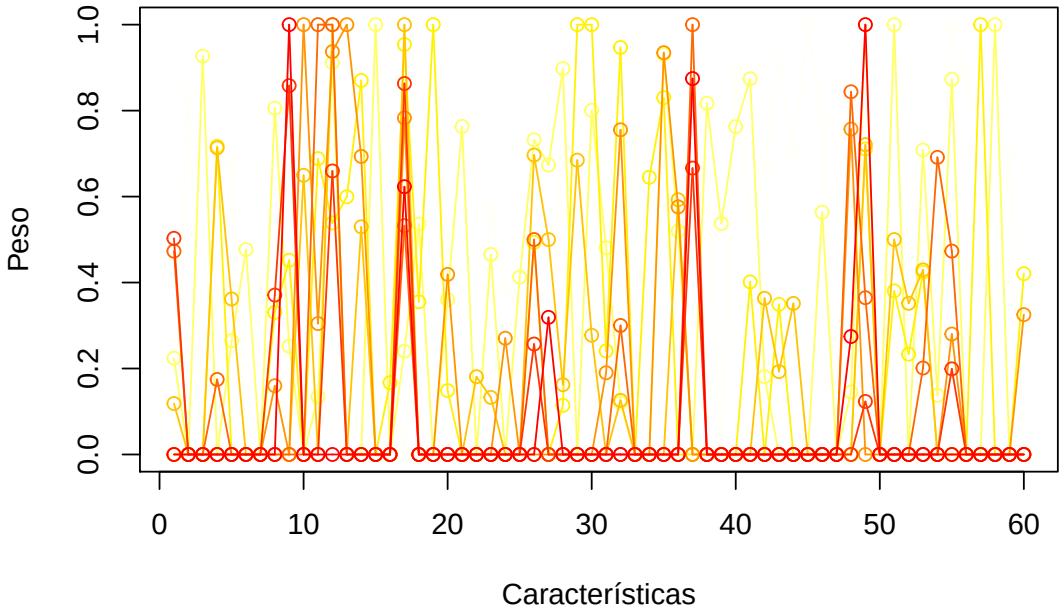
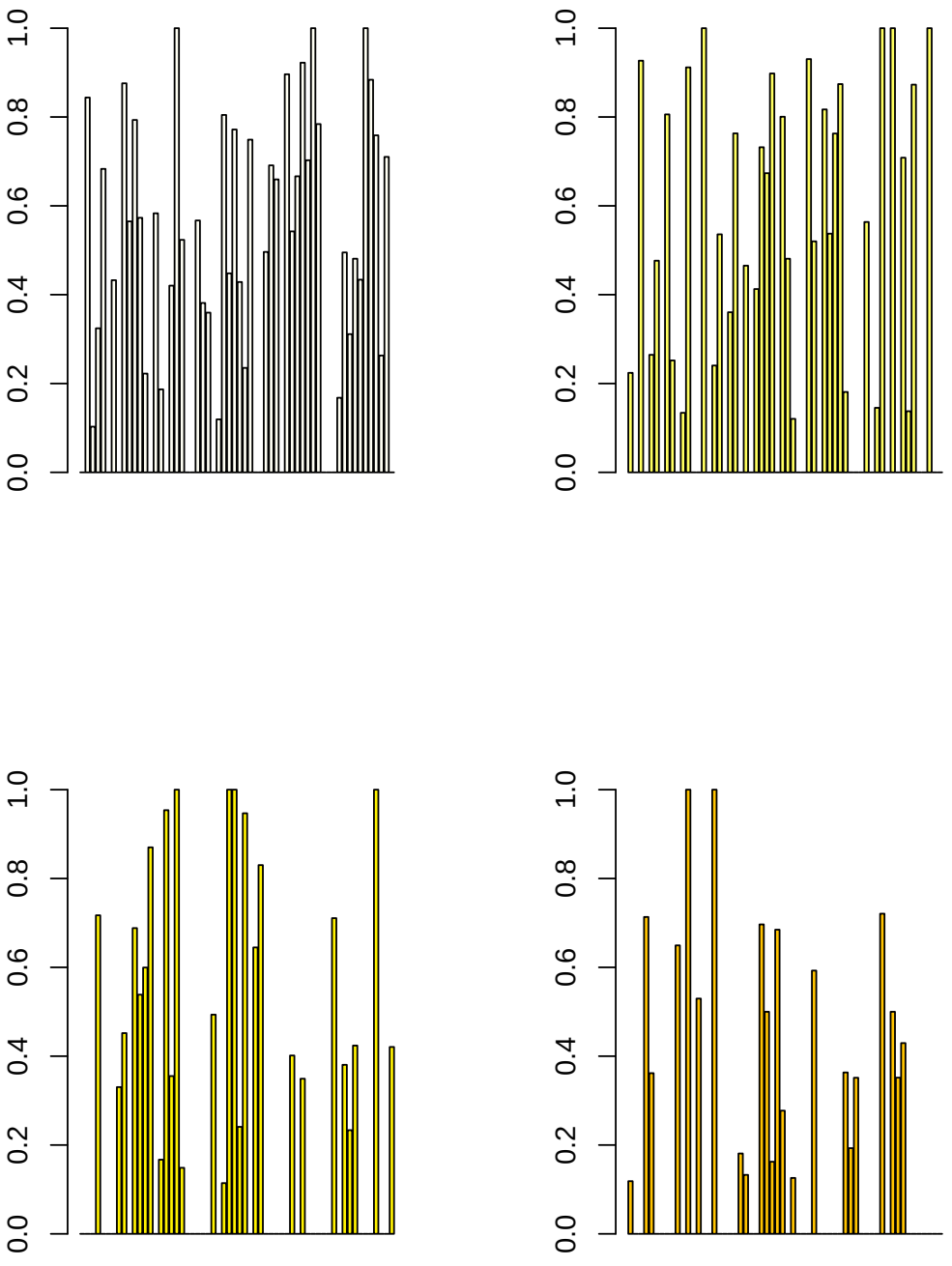


Figura 18: Evolución superpuesta de las mejores soluciones con el operador de cruce RAND en evolución diferencial (los colores más cálidos representan soluciones más evolucionadas).


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

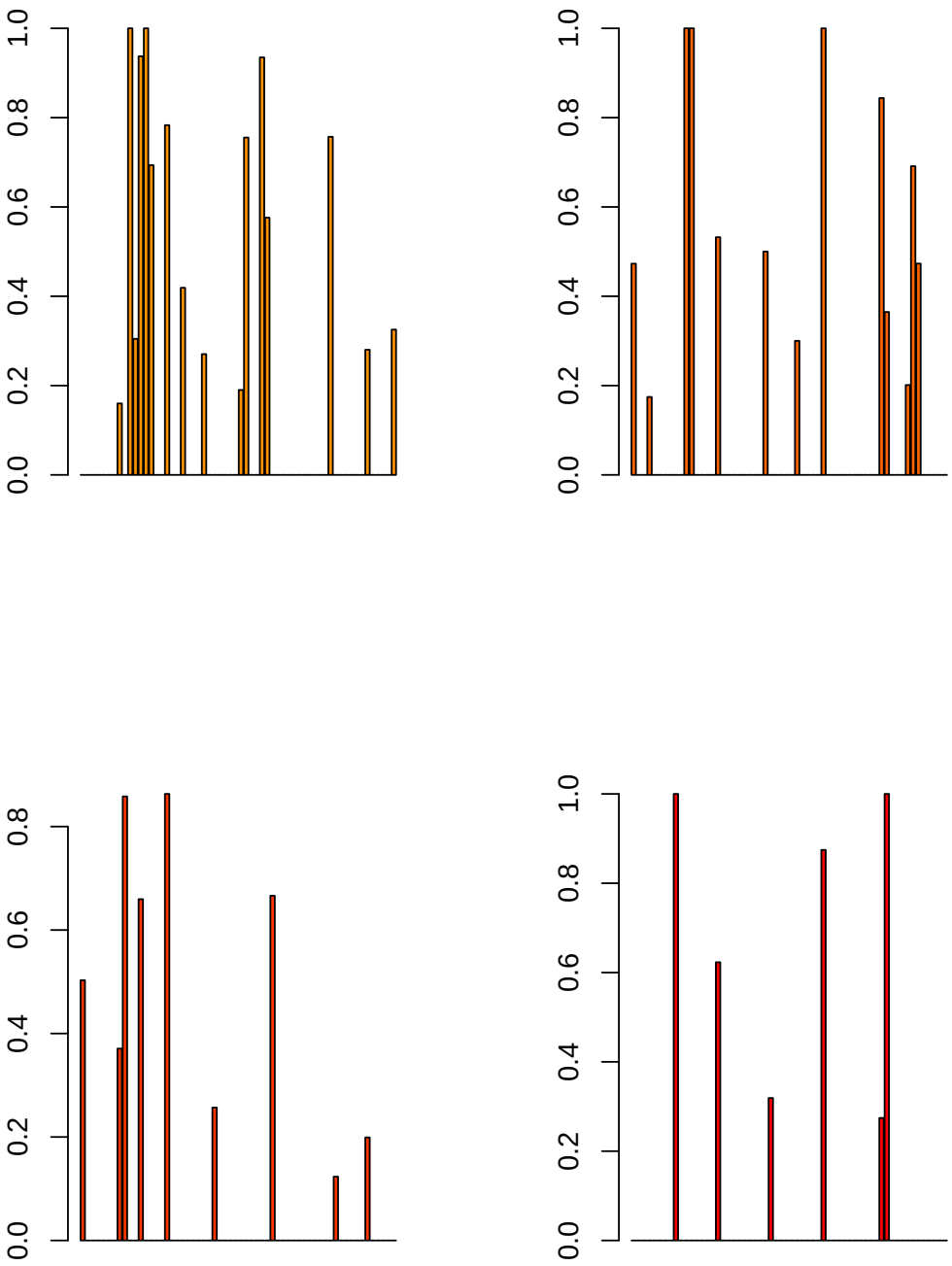


Figura 19: Evolución de las mejores soluciones con el operador de cruce RAND en evolución diferencial.

En la gráfica de soluciones superpuestas vemos cómo hay una gran variabilidad en las mejores soluciones que se van obteniendo. Esto se debe a que en este algoritmo el cruce se basa en padres escogidos completamente al azar entre la población, por lo que la componente aleatoria es importante. También el gran número de cambios muestra que esta forma de recombinación, a pesar de ser

aleatoria es muy efectiva, y además reduce enormemente los pesos, como se observa en la segunda gráfica.

A continuación realizamos el mismo proceso con `DE-CurrentToBest`.

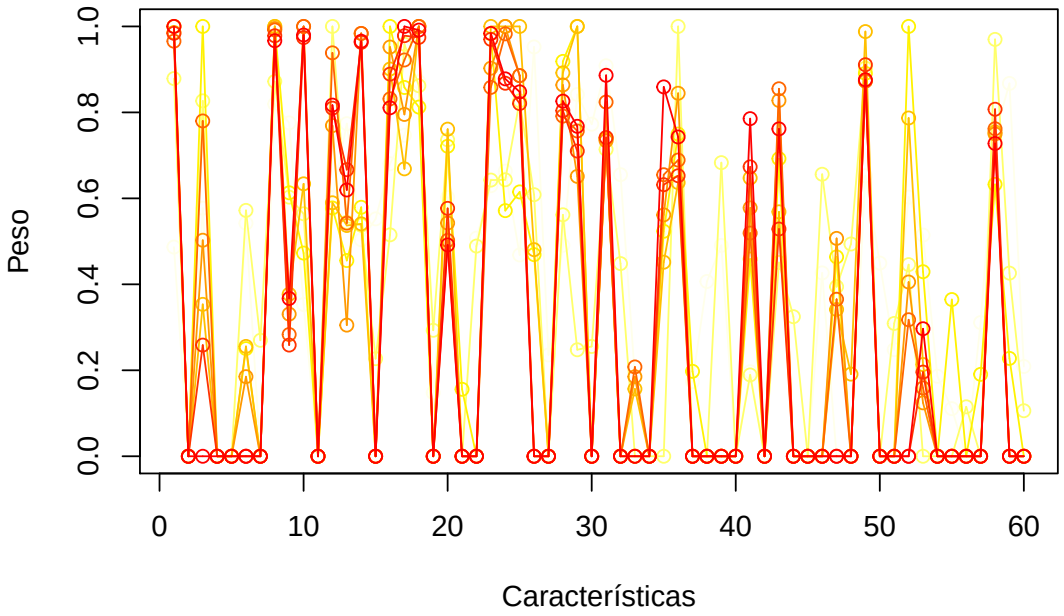
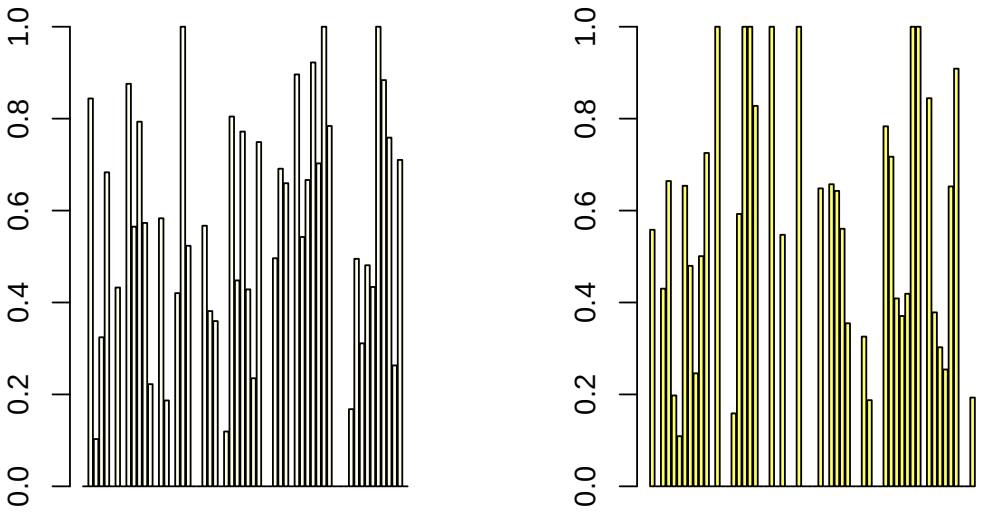
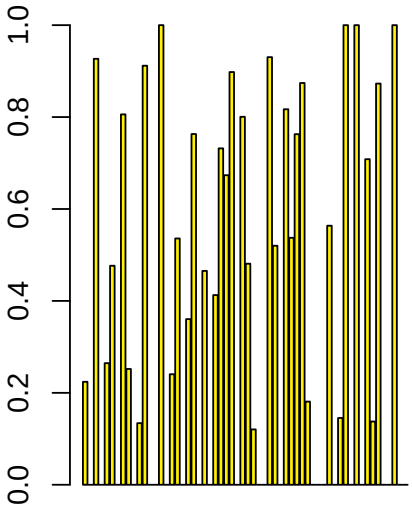
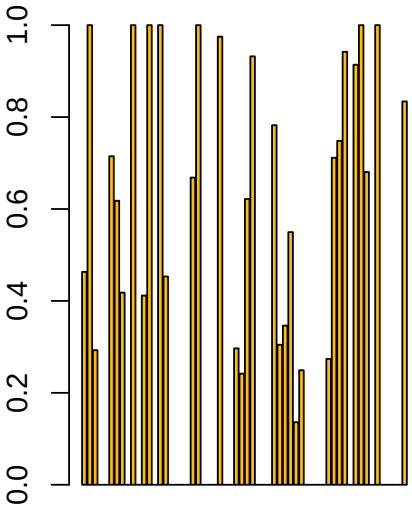
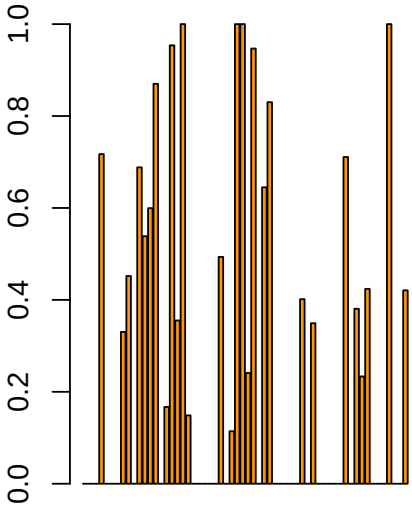


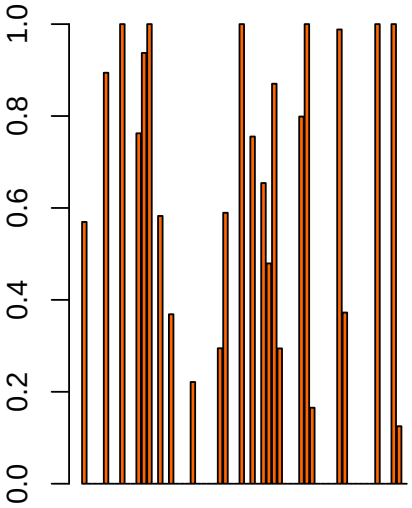
Figura 20: Evolución de las mejores soluciones superpuestas con el operador de cruce CurrentToBest en evolución diferencial (los colores más cálidos representan soluciones más evolucionadas)




0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0


0.0
0.2
0.4
0.6
0.8
1.0

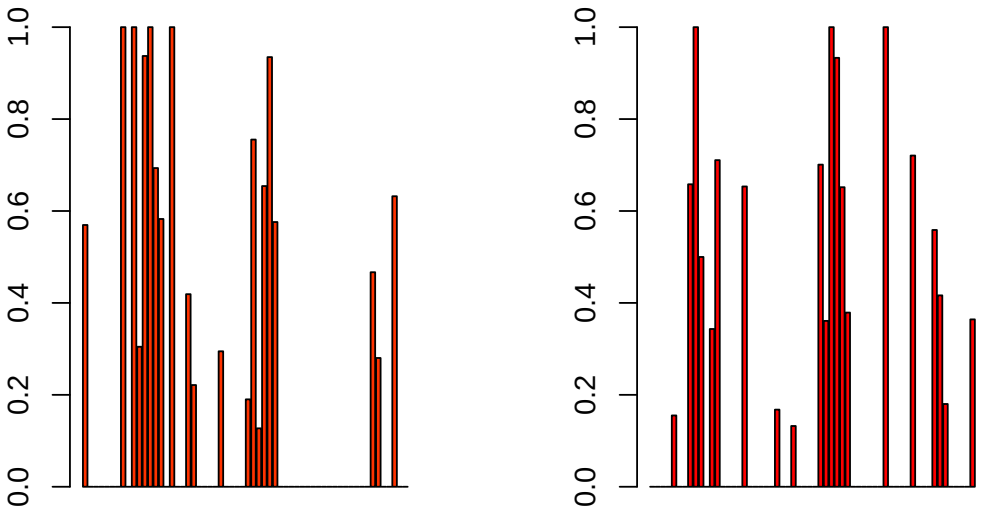
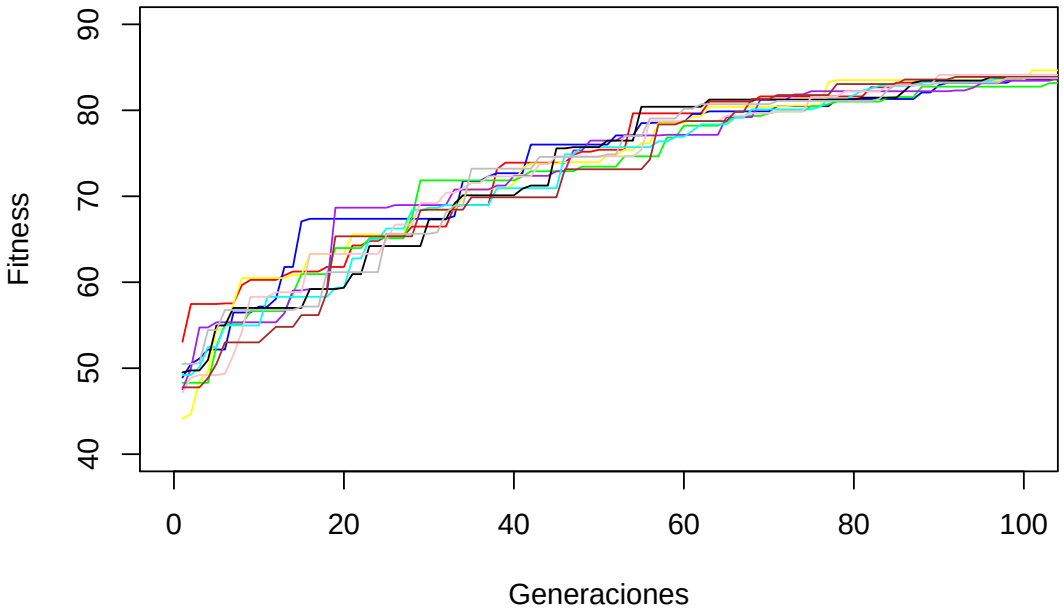


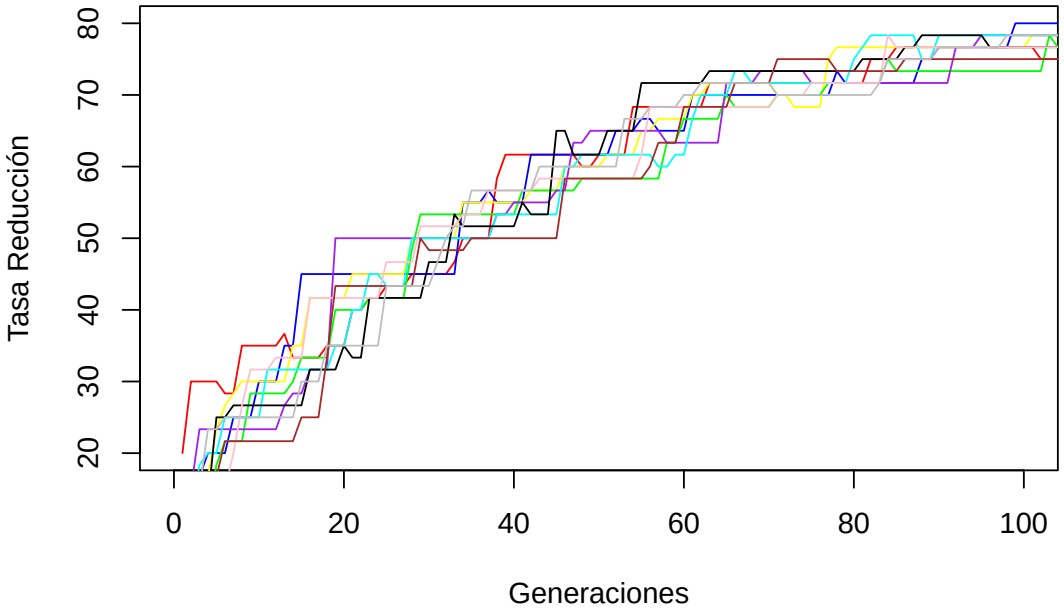
Figura 21: Evolución de las mejores soluciones superpuestas con el operador de cruce CurrentToBest en evolución diferencial

En este caso vemos en la gráfica de soluciones superpuestas que los pesos se mueven mucho menos y tienden a estabilizarse en torno a una solución. Esto puede deberse a que en este caso, aunque también influye el movimiento de padres escogidoa al azar, el operador de cruce tiene una componente no aleatoria que tiende a mover las soluciones en la dirección de la mejor solución. De esta forma, la variabilidad es menor. Además, también disminuye bastante la reducción de pesos porque el acercamiento a la mejor solución, que inicialmente no suele tener pesos bajos, dificulta que los pesos en los hijos puedan bajar.

Evolución de una población completa en DE-RAND

A continuación vamos a ver cómo evoluciona una población de individuos mediante la evolución diferencial con el cruce `RAND`. Para ello, ejecutamos el algoritmo con una población de 10 individuos y vemos cómo evolucionan sus tasas con el número de generaciones. El menor número de individuos impide que se alcancen las mismas tasas que con 50, pero aun así se puede apreciar la tendencia evolutiva de la población.


Fitness
40
50
60
70
80
90
0
20
40
60
80
100
Generaciones


Tasa Reducción
20
30
40
50
60
70
80
0
20
40
60
80
100
Generaciones

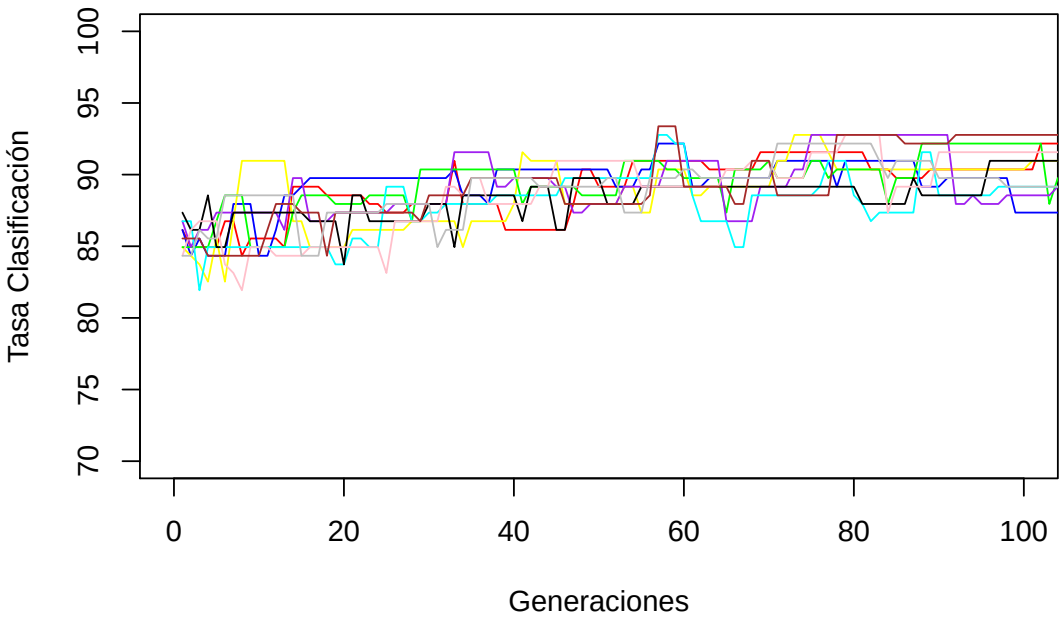


Figura 22: Evolución de fitness, tasa de clasificación y tasa de reducción en DE-RAND para una población de 10 individuos. Cada color representa a un individuo.

Podemos apreciar claramente en primer lugar, en la gráfica de fitness, el elitismo del algoritmo y el reemplazamiento uno a uno. Como consecuencia del reeemplazamiento uno a uno, cada individuo nunca va a empeorar su solución, luego todos los fitness son funciones crecientes. Además, esto nos asegura el elitismo, ya que la mejor solución siempre se va a mantener, salvo que sea reemplazada por una solución mejor.

En las tasas de clasificación y reducción no se puede decir lo mismo, y es que a veces las soluciones bajan su valor en estas gráficas o incluso se pierde el mejor valor para una de las tasas. Pero como lo que el algoritmo optimiza el agregado de las tasas, estas bajadas implican una subida en la tasa complementaria, y esto permite conservar el crecimiento constante de la función objetivo. También vemos que las tasas de clasificación oscilan en torno a los mismos valores a lo largo del algoritmo, mientras que las de reducción suben bastante rápido, como ha estado ocurriendo a lo largo de los distintos algoritmos heurísticos.

Evolución de los algoritmos de la práctica 1

Finalmente analizamos los dos algoritmos genéticos incorporados de la práctica 1.

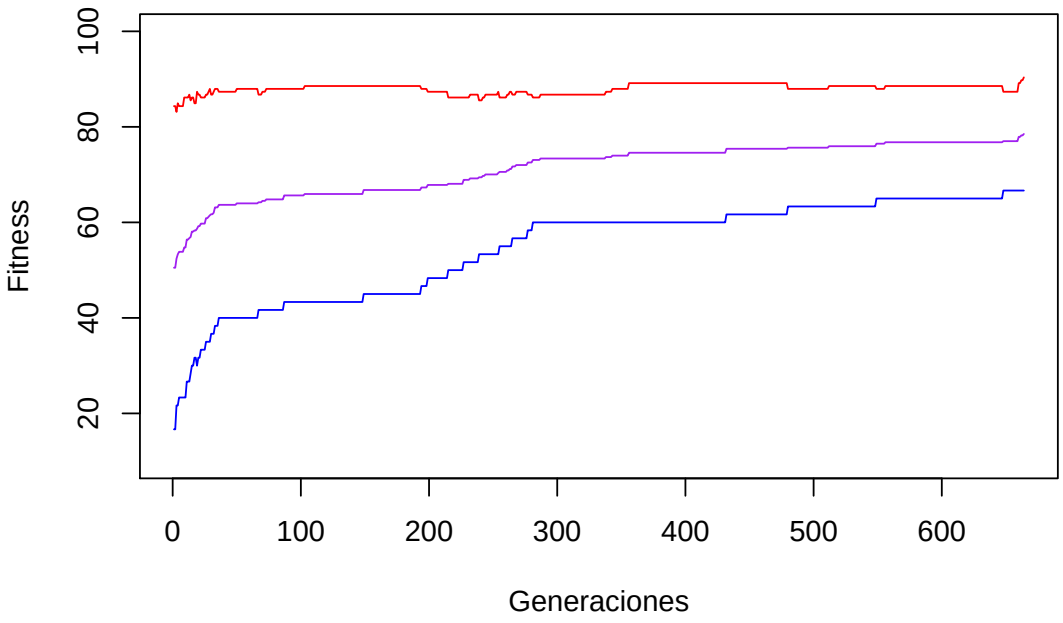


Figura 23: Evolución de la función objetivo en función del número de generaciones en AGG-BLX

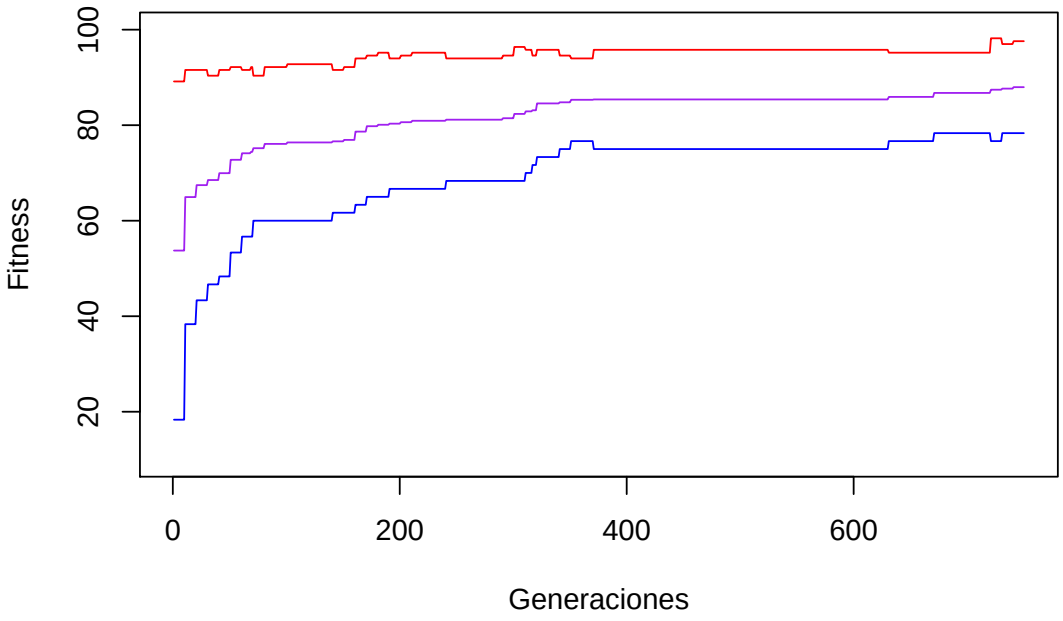


Figura 24: Evolución de la función objetivo en función del número de generaciones en AM-10-0.1mej

En el caso del AGG-BLX podemos observar un crecimiento lento pero constante, principalmente

debido a la reducción. El operador de cruce BLX favorece la diversificación y en consecuencia va permitiendo el aumento de reducciones siempre que sigan manteniendo una buena tasa de clasificación, lo que permite este crecimiento pausado.

En cuanto al algoritmo memético, se aprecia un comportamiento similar al del AGG, pero con algunos saltos más marcados, sobre todo al principio, debidos posiblemente a que en las generaciones de búsqueda local la solución a la que se le ha aplicado dicha búsqueda ha conseguido dar un salto de calidad.

Referencias

[1] Cygwin, https://www.cygwin.com/.

[2] teju85. Arff formated file reader in c++, https://github.com/teju85/ARFF.