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

5de 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	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	42.8571	85.7143	0.0000	0.0011	-	46.4912	92.9825	0.0000	0.0046	-	41.3043	82.6087	0.0000	0.0059	-
PARTITION 2	39.2857	78.5714	0.0000	0.0011	-	48.2456	96.4912	0.0000	0.0042	-	40.7609	81.5217	0.0000	0.0050	-
PARTITION 3	47.6190	95.2381	0.0000	0.0011	-	47.8070	95.6140	0.0000	0.0042	-	45.1087	90.2174	0.0000	0.0050	-
PARTITION 4	42.6829	85.3659	0.0000	0.0011	-	47.3684	94.7368	0.0000	0.0042	-	39.6739	79.3478	0.0000	0.0056	-
PARTITION 5	46.3415	92.6829	0.0000	0.0011	-	46.9027	93.8053	0.0000	0.0050	-	42.3913	84.7826	0.0000	0.0052	-
MEAN	**43.7573**	**87.5145**	**0.0000**	**0.0011**	-	**47.3630**	**94.7260**	**0.0000**	**0.0044**	-	**41.8478**	**83.6957**	**0.0000**	**0.0053**	-
STDEV	*2.9513*	*5.9026*	*0.0000*	*0.0000*	-	*0.6240*	*1.2480*	*0.0000*	*0.0003*	-	*1.8510*	*3.7019*	*0.0000*	*0.0003*	-

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

RELIEF	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	48.3333	83.3333	13.3333	0.0021	49.1365	48.5965	93.8596	3.3333	0.0064	49.4689	52.6697	91.3043	14.0351	0.0079	50.6317
PARTITION 2	45.1190	78.5714	11.6667	0.0018	49.8092	48.2456	96.4912	0.0000	0.0063	47.4725	54.0046	86.9565	21.0526	0.0078	55.3633
PARTITION 3	50.9524	95.2381	6.6667	0.0018	43.3936	46.4912	92.9825	0.0000	0.0065	48.1319	58.6003	89.1304	28.0702	0.0078	57.9210
PARTITION 4	48.5163	85.3659	11.6667	0.0018	50.4441	47.8070	95.6140	0.0000	0.0062	47.6923	56.3024	88.0435	24.5614	0.0078	55.6231
PARTITION 5	54.2886	90.2439	18.3333	0.0018	52.5798	53.2301	96.4602	10.0000	0.0065	52.5877	54.0046	86.9565	21.0526	0.0086	55.0915
MEAN	**49.4419**	**86.5505**	**12.3333**	**0.0018**	**49.0727**	**48.8741**	**95.0815**	**2.6667**	**0.0064**	**49.0707**	**55.1163**	**88.4783**	**21.7544**	**0.0080**	**54.9261**
STDEV	*3.0510*	*5.7365*	*3.7417*	*0.0001*	*3.0652*	*2.2919*	*1.4194*	*3.8873*	*0.0001*	*1.8902*	*2.0978*	*1.6271*	*4.6549*	*0.0003*	*2.3713*

Figura 2: Resultados de la ejecución del greedy RELIEF.

RANDOM+LS	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	79.1667	83.3333	75.0000	3.6433	83.8855	90.6140	91.2281	90.0000	8.6327	93.0220	80.8638	88.0435	73.6842	10.4363	80.8638
PARTITION 2	77.0238	85.7143	68.3333	3.6765	81.1546	89.2982	88.5965	90.0000	5.4241	91.0440	78.0225	85.8696	70.1754	21.1528	80.8758
PARTITION 3	67.2619	92.8571	41.6667	1.8791	65.1104	90.6140	91.2281	90.0000	9.7570	93.0220	81.5313	85.8696	77.1930	24.7188	83.5693
PARTITION 4	80.9553	90.2439	71.6667	3.4494	81.9411	80.2632	93.8596	66.6667	4.8975	81.3553	86.6705	89.1304	84.2105	29.7477	87.8933
PARTITION 5	78.9024	87.8049	70.0000	4.6077	81.4072	84.3510	92.0354	76.6667	7.7362	85.1535	87.2140	90.2174	84.2105	29.8601	87.2140
MEAN	**76.6620**	**87.9907**	**65.3333**	**3.4512**	**78.6998**	**87.0281**	**91.3895**	**82.6667**	**7.2895**	**88.7193**	**82.8604**	**87.8261**	**77.8947**	**23.1831**	**84.0832**
STDEV	*4.8624*	*3.3367*	*12.0370*	*0.8828*	*6.8618*	*4.0940*	*1.6953*	*9.5219*	*1.8598*	*4.6770*	*3.5391*	*1.7391*	*5.6140*	*7.1643*	*3.0079*

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

AGG-BLX	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	73.8095	80.9524	66.6667	14.9841	78.5141	87.8070	95.6140	80.0000	56.3108	88.0220	84.5824	90.2174	78.9474	66.7655	82.5443
PARTITION 2	67.6190	78.5714	56.6667	14.9873	75.3213	83.2456	96.4912	70.0000	54.4912	82.6923	75.9725	78.2609	73.6842	67.2988	82.3584
PARTITION 3	75.4762	97.6190	53.3333	14.3756	71.8474	81.0526	92.1053	70.0000	55.6086	81.5934	85.9172	85.8696	85.9649	67.3933	87.0042
PARTITION 4	72.7439	80.4878	65.0000	14.5858	78.0090	79.4737	95.6140	63.3333	54.1903	79.4689	77.6888	86.9565	68.4211	67.9398	79.8627
PARTITION 5	75.5081	92.6829	58.3333	14.9868	76.1727	80.6785	94.6903	66.6667	57.0351	81.2500	77.0595	80.4348	73.6842	68.1847	81.5432
MEAN	**73.0314**	**86.0627**	**60.0000**	**14.7839**	**75.9729**	**82.4515**	**94.9030**	**70.0000**	**55.5272**	**82.6053**	**80.2441**	**84.3478**	**76.1404**	**67.5164**	**82.6626**
STDEV	*2.9015*	*7.6248*	*5.0552*	*0.2564*	*2.3701*	*2.9421*	*1.5101*	*5.5777*	*1.0728*	*2.8998*	*4.1454*	*4.3804*	*5.9339*	*0.5005*	*2.3685*

Figura 4: Resultados de la ejecución del algoritmo AGG-BLX.

AM-10-0.1mej	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	69.2857	78.5714	60.0000	14.9347	75.7831	88.3333	83.3333	93.3333	57.4397	93.3700	81.5694	77.1739	85.9649	66.2931	86.7325
PARTITION 2	78.4524	78.5714	78.3333	14.9200	87.9619	90.9649	88.5965	93.3333	56.8189	91.6117	84.3726	88.0435	80.7018	68.6867	87.2259
PARTITION 3	78.8095	80.9524	76.6667	14.8610	84.4177	91.4912	92.9825	90.0000	56.6432	92.3626	86.1270	88.0435	84.2105	67.7339	85.9911
PARTITION 4	71.9715	75.6098	68.3333	15.0340	80.5739	92.7193	92.1053	93.3333	57.0342	93.6996	87.5477	89.1304	85.9649	64.6792	89.7216
PARTITION 5	78.5163	85.3659	71.6667	14.3645	82.5399	91.4602	92.9204	90.0000	57.0791	91.6009	87.8814	88.0435	87.7193	67.4149	88.9683
MEAN	**75.4071**	**79.8142**	**71.0000**	**14.8229**	**82.2553**	**90.9938**	**89.9876**	**92.0000**	**57.0030**	**92.5290**	**85.4996**	**86.0870**	**84.9123**	**66.9615**	**87.7279**
STDEV	*3.9948*	*3.2523*	*6.5490*	*0.2358*	*4.0505*	*1.4507*	*3.6950*	*1.6331*	*0.2688*	*0.8728*	*2.3229*	*4.4762*	*2.3799*	*1.3736*	*1.3979*

Figura 5: Resultados de la ejecución del algoritmo AM-10-0.1mej.

SA	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	78.9286	76.1905	81.6667	10.7083	87.5201	91.4035	89.4737	93.3333	30.4286	93.6996	87.1281	84.7826	89.4737	55.1069	91.0684
PARTITION 2	81.7857	78.5714	85.0000	11.2435	90.0904	93.1579	92.9825	93.3333	30.8173	93.5897	89.9695	86.9565	92.9825	52.6517	92.1434
PARTITION 3	82.5000	83.3333	81.6667	11.2872	89.0261	93.1579	92.9825	93.3333	48.0433	93.3700	85.7075	83.6957	87.7193	55.2438	89.7836
PARTITION 4	83.5772	80.4878	86.6667	11.2368	91.2375	92.2807	91.2281	93.3333	30.3007	93.3700	90.3890	91.3043	89.4737	53.8726	91.0684
PARTITION 5	82.2967	82.9268	81.6667	9.8196	86.9411	93.1268	92.9204	93.3333	48.9281	93.4868	89.0923	86.9565	91.2281	54.8066	89.9075
MEAN	**81.8176**	**80.3020**	**83.3333**	**10.8591**	**88.9630**	**92.6254**	**91.9174**	**93.3333**	**37.7036**	**93.5032**	**88.4573**	**86.7391**	**90.1754**	**54.3363**	**90.7942**
STDEV	*1.5583*	*2.6837*	*2.1083*	*0.5616*	*1.5888*	*0.6967*	*1.3940*	*0.0000*	*8.8096*	*0.1288*	*1.7748*	*2.6088*	*1.7893*	*0.9688*	*0.8700*

Figura 6: Resultados de la ejecución del algoritmo Enfriamiento Simulado.

ILS	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	78.0952	76.1905	80.0000	15.0618	84.8795	93.1579	92.9825	93.3333	58.1310	94.1392	85.7075	83.6957	87.7193	70.6167	90.0553
PARTITION 2	80.8333	83.3333	78.3333	15.1045	87.0582	93.1579	92.9825	93.3333	58.7871	93.5897	87.6716	85.8696	89.4737	70.1089	90.5249
PARTITION 3	72.8571	69.0476	76.6667	15.0900	83.8153	91.9298	93.8596	90.0000	57.9531	93.5714	85.7075	83.6957	87.7193	68.5148	89.9194
PARTITION 4	84.3496	85.3659	83.3333	15.2457	87.4751	91.8421	90.3509	93.3333	55.3295	93.8095	87.1281	84.7826	89.4737	68.3818	89.3021
PARTITION 5	83.9024	87.8049	80.0000	14.8382	87.0060	94.0118	94.6903	93.3333	57.3521	93.5965	91.2662	91.3043	91.2281	68.9792	90.4510
MEAN	**80.0076**	**80.3484**	**79.6667**	**15.0680**	**86.0468**	**92.8199**	**92.9731**	**92.6667**	**57.5106**	**93.7413**	**87.4962**	**85.8696**	**89.1228**	**69.3203**	**90.0505**
STDEV	*4.2306*	*6.8510*	*2.2109*	*0.1314*	*1.4368*	*0.8244*	*1.4571*	*1.3328*	*1.1828*	*0.2165*	*2.0386*	*2.8344*	*1.3127*	*0.8883*	*0.4386*

Figura 7: Resultados de la ejecución del algoritmo Búsqueda Local Reiterada.

DE-RAND	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	90.2381	90.4762	90.0000	14.8074	92.2892	90.5263	87.7193	93.3333	54.0429	94.1392	85.4977	81.5217	89.4737	67.2046	90.3890
PARTITION 2	81.9048	73.8095	90.0000	14.7132	92.5904	93.1579	92.9825	93.3333	54.1080	93.5897	89.0923	86.9565	91.2281	68.0348	92.2173
PARTITION 3	77.5000	66.6667	88.3333	14.6252	90.5522	91.8421	90.3509	93.3333	55.3684	94.0293	86.0412	82.6087	89.4737	68.6367	91.3401
PARTITION 4	82.3577	78.0488	86.6667	14.8624	90.6387	92.7193	92.1053	93.3333	54.3290	93.6996	88.8825	84.7826	92.9825	66.7428	92.0075
PARTITION 5	86.8496	85.3659	88.3333	14.9192	91.4721	94.0118	94.6903	93.3333	56.7629	93.5965	88.0053	84.7826	91.2281	66.4902	91.4021
MEAN	**83.7700**	**78.8734**	**88.6667**	**14.7855**	**91.5085**	**92.4515**	**91.5696**	**93.3333**	**54.9222**	**93.8109**	**87.5038**	**84.1304**	**90.8772**	**67.4218**	**91.4712**
STDEV	*4.3843*	*8.3918*	*1.2469*	*0.1050*	*0.8315*	*1.1900*	*2.3807*	*0.0000*	*1.0368*	*0.2296*	*1.4722*	*1.8953*	*1.3129*	*0.8037*	*0.6381*

Figura 8: Resultados de la ejecución del algoritmo DE-RAND.

DE-CURR-TO-BEST	SONAR					WDBC					SPAMBASE				
	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT	FITNESS	CLASS	RED	TIME	TRAIN FIT
PARTITION 1	71.6667	83.3333	60.0000	14.2765	75.4819	84.8246	92.9825	76.6667	57.5802	85.5861	75.6007	88.0435	63.1579	65.9952	75.0572
PARTITION 2	71.7857	78.5714	65.0000	14.6351	77.0783	91.9298	93.8596	90.0000	57.6554	90.6044	70.0420	80.4348	59.6491	67.2328	75.2050
PARTITION 3	73.9286	92.8571	55.0000	14.6132	69.9699	76.4912	92.9825	60.0000	56.6139	78.6813	74.3898	89.1304	59.6491	68.2301	72.8952
PARTITION 4	69.0244	78.0488	60.0000	14.8885	75.8084	85.7895	98.2456	73.3333	56.3077	84.9084	70.2517	82.6087	57.8947	67.5619	73.2408
PARTITION 5	72.7439	80.4878	65.0000	14.3175	76.8114	86.0177	92.0354	80.0000	56.5110	86.3816	70.9191	80.4348	61.4035	66.6642	75.4028
MEAN	**71.8298**	**82.6597**	**61.0000**	**14.5462**	**75.0300**	**85.0106**	**94.0211**	**76.0000**	**56.9336**	**85.2324**	**72.2407**	**84.1304**	**60.3509**	**67.1368**	**74.3602**
STDEV	*1.6211*	*5.4246*	*3.7417*	*0.2257*	*2.5994*	*4.9419*	*2.1897*	*9.7525*	*0.5675*	*3.8295*	*2.2998*	*3.7402*	*1.7890*	*0.7629*	*1.0664*

Figura 9: Resultados de la ejecución del algoritmo DE-CURRENTTOBEST.

SEED-3141592	SONAR					WDBC					SPAMBASE				
ALGORITHMS	FITNESS	CLASS_RATE	RED_RATE	TIME	TRAIN	FITNESS	CLASS_RATE	RED_RATE	TIME	TRAIN	FITNESS	CLASS_RATE	RED_RATE	TIME	TRAIN
1NN	43.7573	87.5145	0.0000	0.0011	42.6672	47.3630	94.7260	0.0000	0.0044	47.7591	41.8478	83.6957	0.0000	0.0053	42.1196
RELIEF	49.4419	86.5505	12.3333	0.0018	49.0727	48.8741	95.0815	2.6667	0.0064	49.0707	55.1163	88.4783	21.7544	0.0080	54.9261
RANDOM+LS	76.6620	87.9907	65.3333	3.4512	78.6998	87.0281	91.3895	82.6667	7.2895	88.7193	82.8604	87.8261	77.8947	23.1831	84.0832
AGG-BLX	73.0314	86.0627	60.0000	14.7839	75.9729	82.4515	94.9030	70.0000	55.5272	82.6053	80.2441	84.3478	76.1404	67.5164	82.6626
AM-10-0.1mej	75.4071	79.8142	71.0000	14.8229	82.2553	90.9938	89.9876	92.0000	57.0030	92.5290	85.4996	86.0870	84.9123	66.9615	87.7279
SA	81.8176	80.3020	83.3333	10.8591	88.9630	92.6254	91.9174	93.3333	37.7036	93.5032	88.4573	86.7391	90.1754	54.3363	90.7942
ILS	80.0076	80.3484	79.6667	15.0680	86.0468	92.8199	92.9731	92.6667	57.5106	93.7413	87.4962	85.8696	89.1228	69.3203	90.0505
DE-RAND	83.77	78.8734	88.6667	14.7855	91.5085	92.4515	91.5696	93.3333	54.9222	93.8109	87.5038	84.1304	90.8772	67.4218	91.4712
DE-CURR-TO-BEST	71.8298	82.6597	61.0000	14.5462	75.03	85.0106	94.0211	76.0000	56.9336	85.2324	72.2407	84.1304	60.3509	67.1368	74.3602

Figura 10: 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).

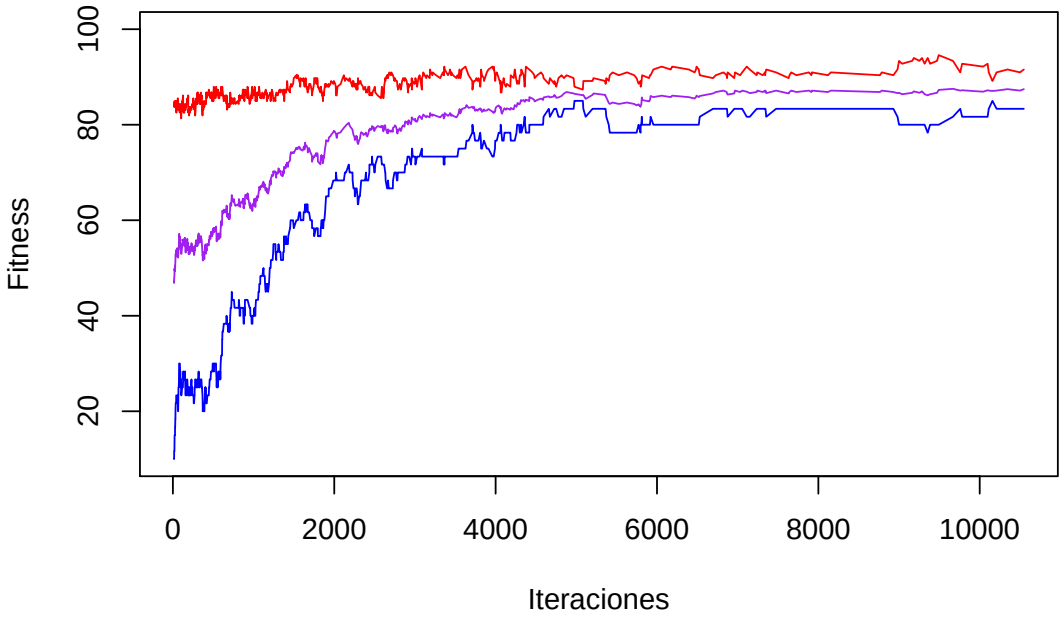


Figura 11: Evolución de la función objetivo en función del número de iteraciones en enfriamiento simulado.

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.

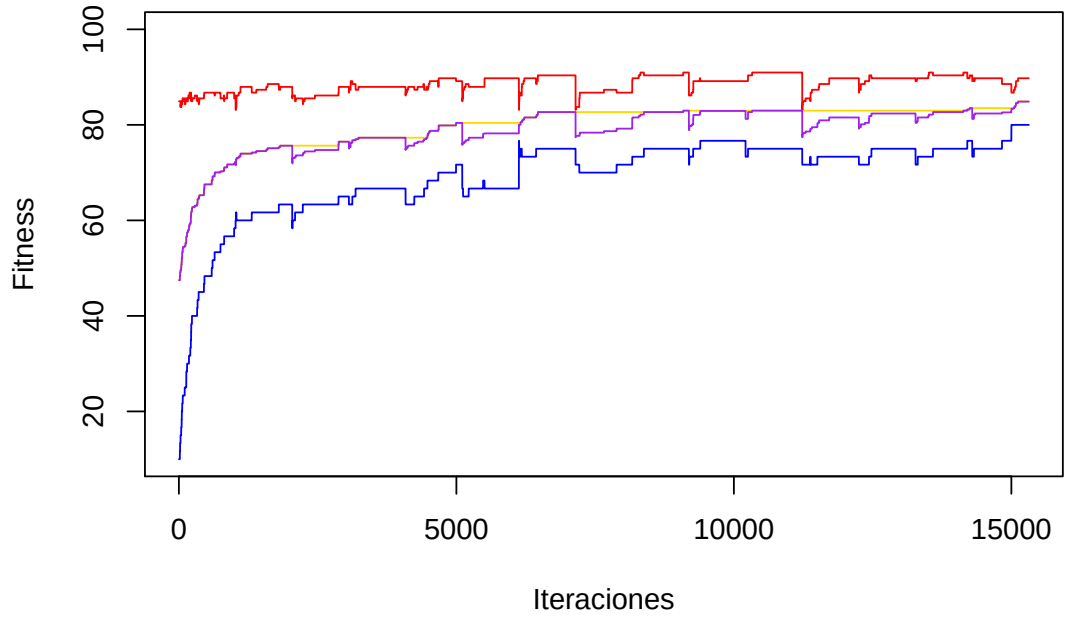


Figura 12: Evolución de la función objetivo en función del número de iteraciones en ILS

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.

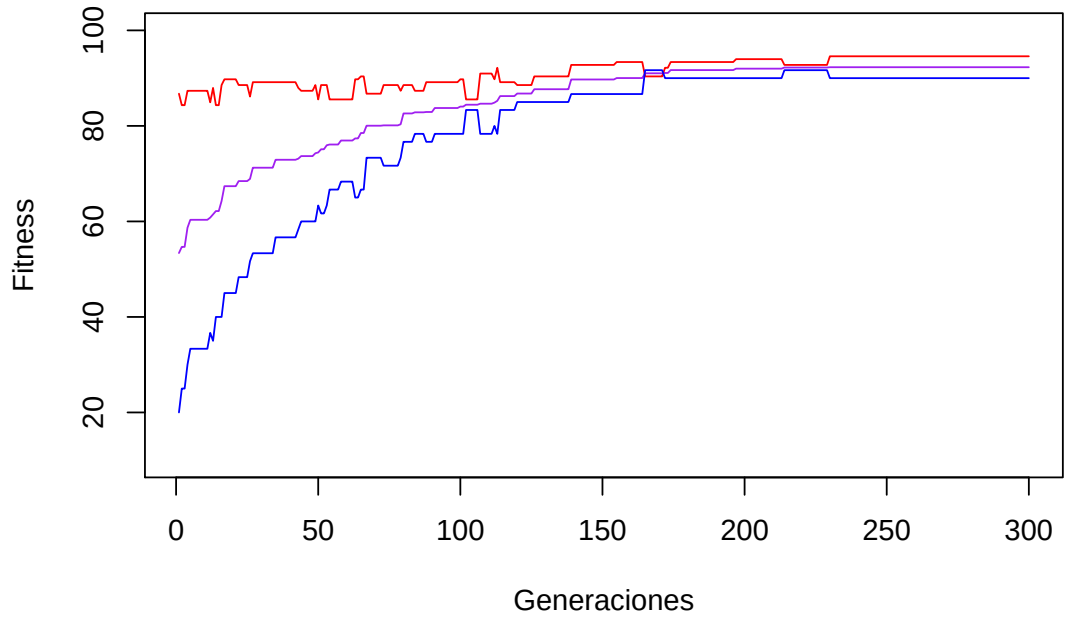


Figura 13: Evolución de la función objetivo en función del número de generaciones en DE-RAND

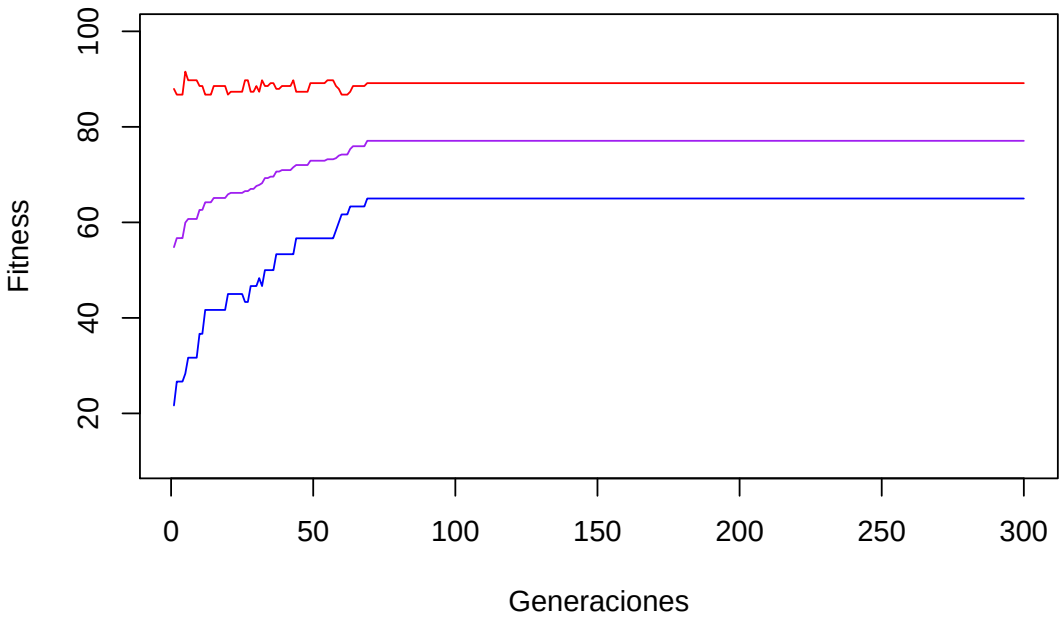


Figura 14: Evolución de la función objetivo en función del número de generaciones en DE-CurrentToBest

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).

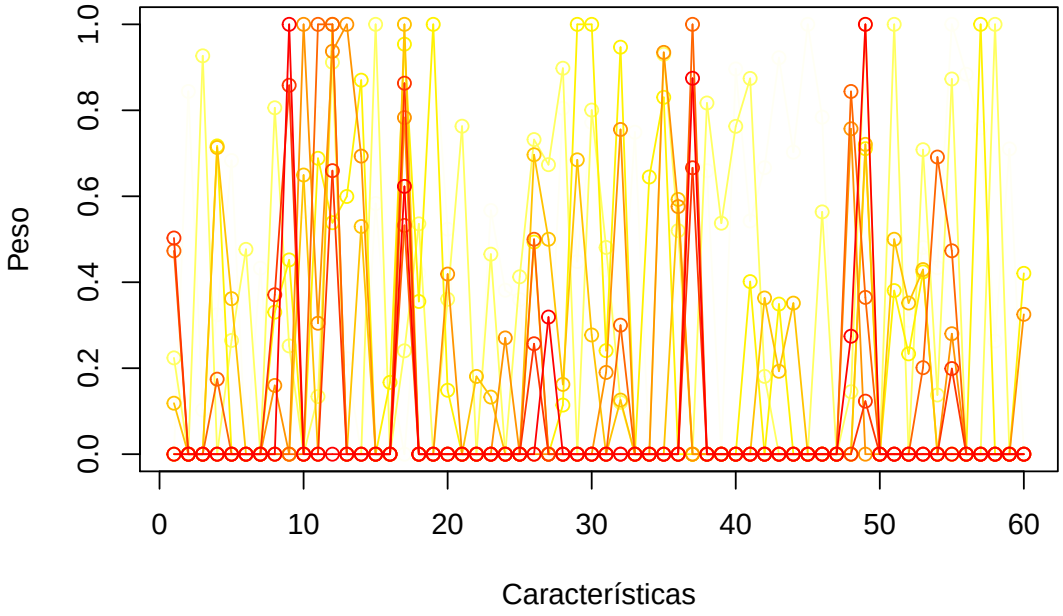
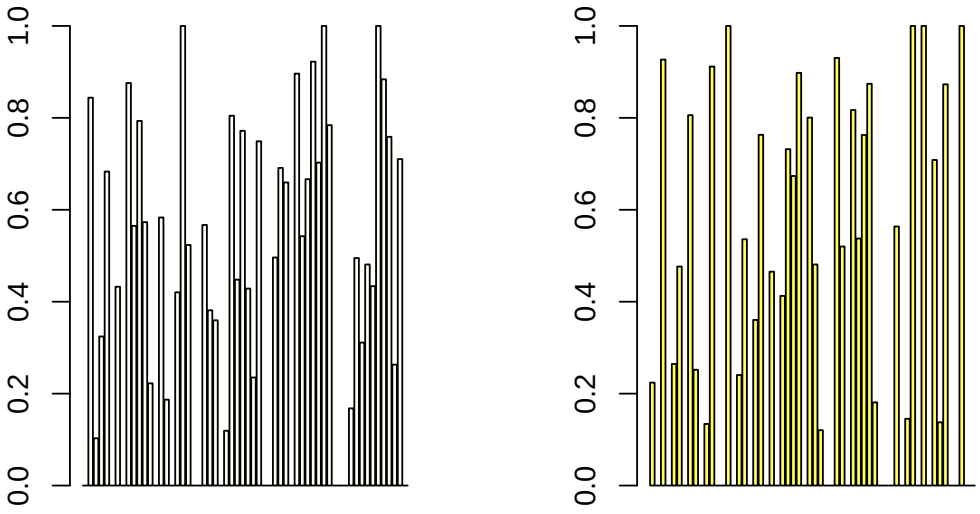
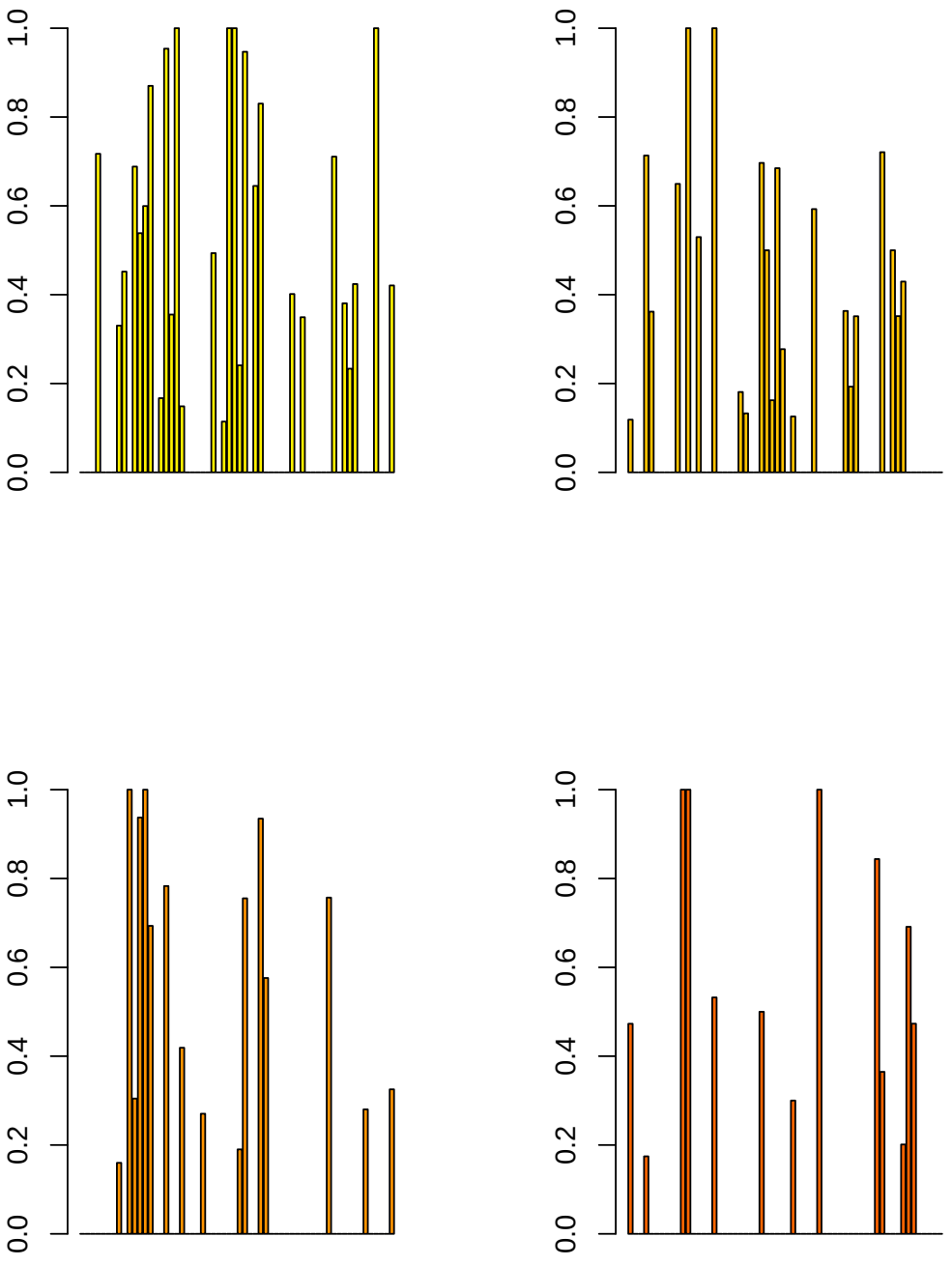


Figura 15: Evolución superpuesta de las mejores soluciones con el operador de cruce RAND en evolución diferencial (los colores más cálidos representan soluciones más evolucionadas).




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

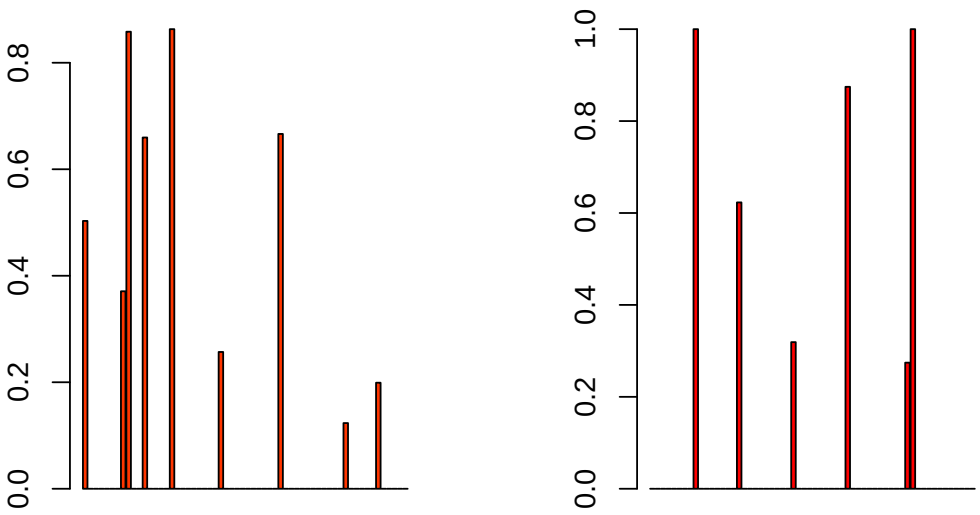


Figura 16: Evolución de las mejores soluciones con el operador de cruce RAND en evolución diferencial.

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`.

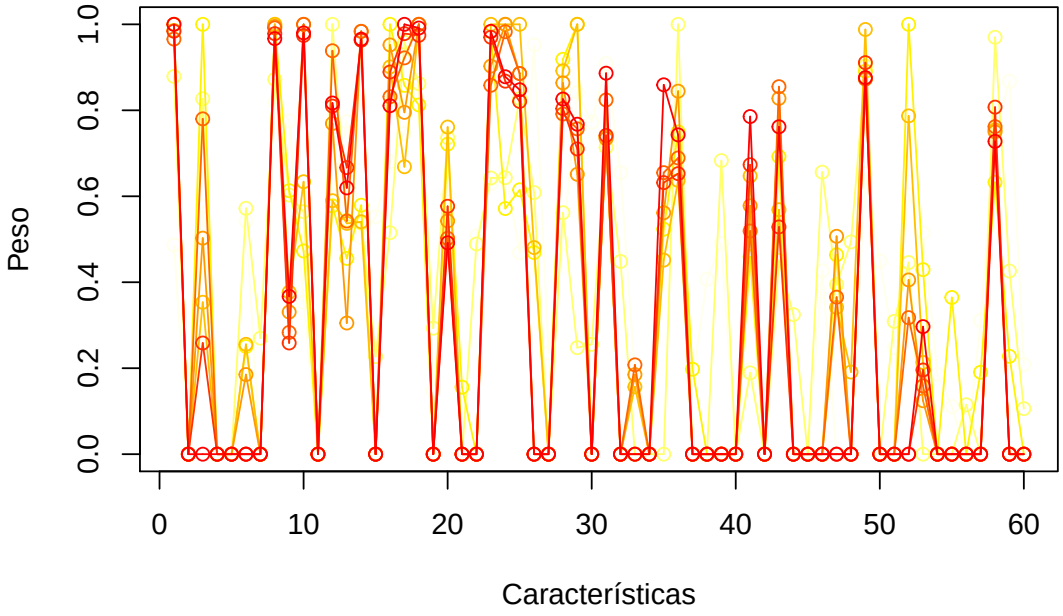
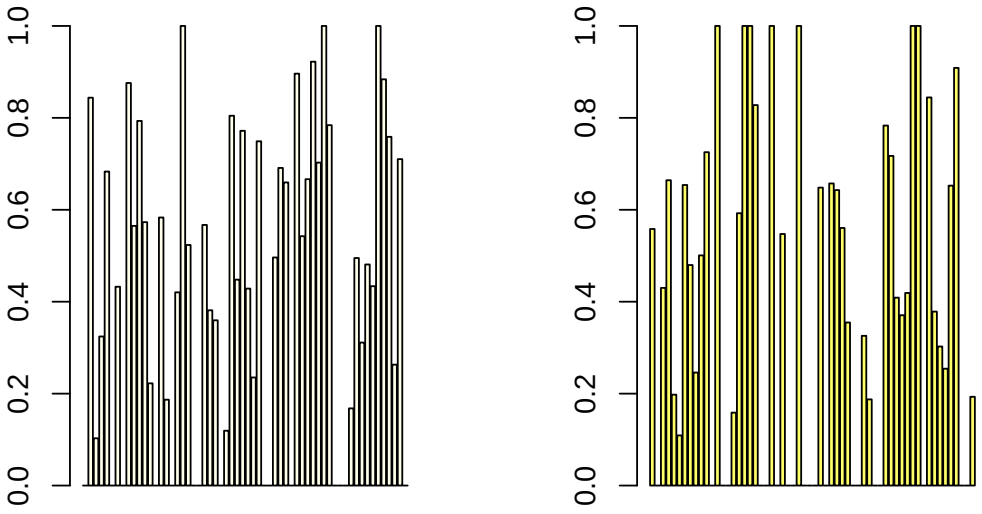
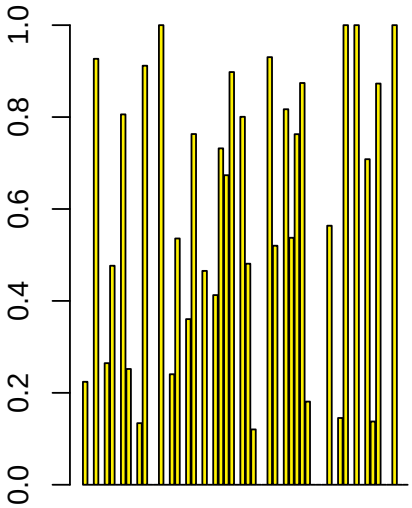
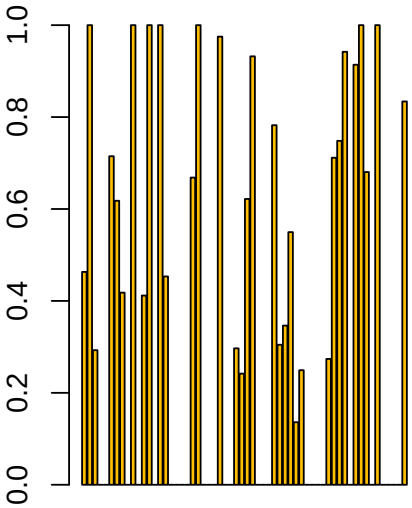
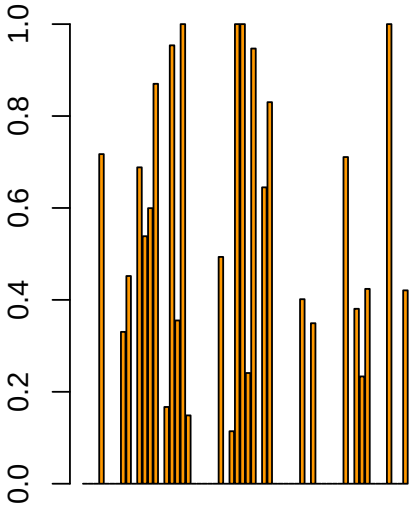


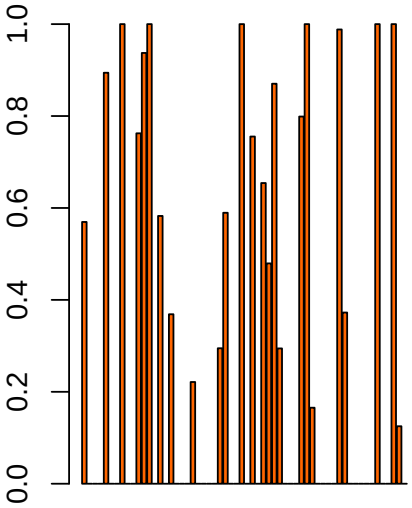
Figura 17: Evolución de las mejores soluciones superpuestas con el operador de cruce CurrentToBest en evolución diferencial (los colores más cálidos representan soluciones más evolucionadas)




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

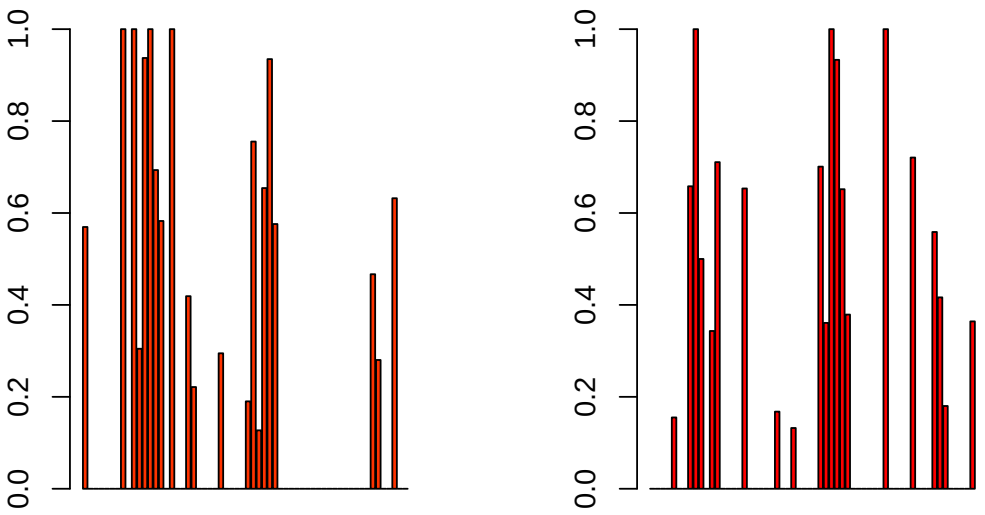
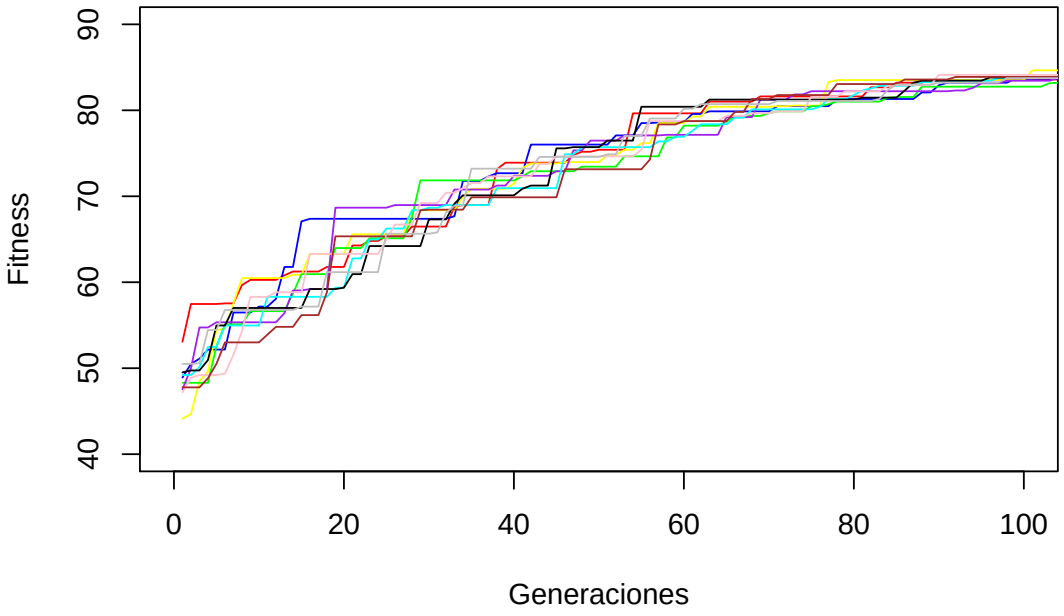


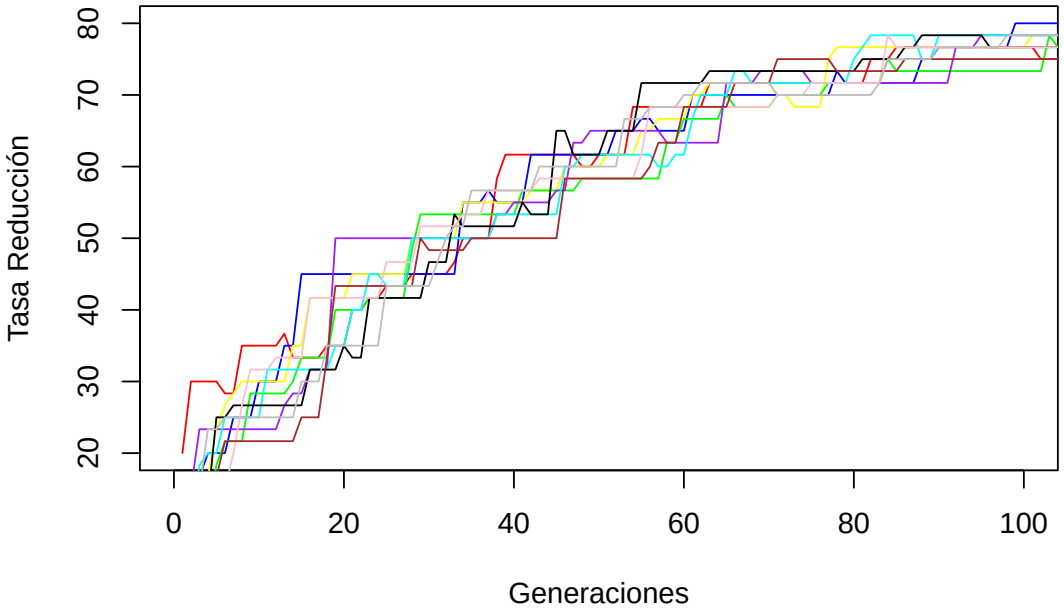
Figura 18: Evolución de las mejores soluciones superpuestas con el operador de cruce CurrentToBest en evolución diferencial

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

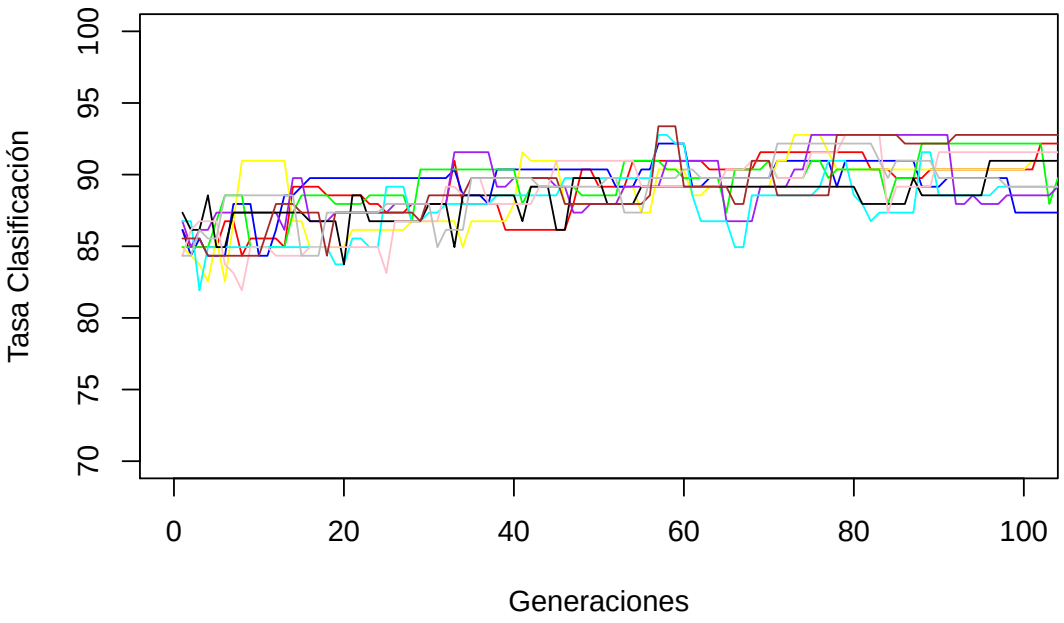


Figura 19: Evolución de fitness, tasa de clasificación y tasa de reducción en DE-RAND para una población de 10 individuos. Cada color representa a un individuo.

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.

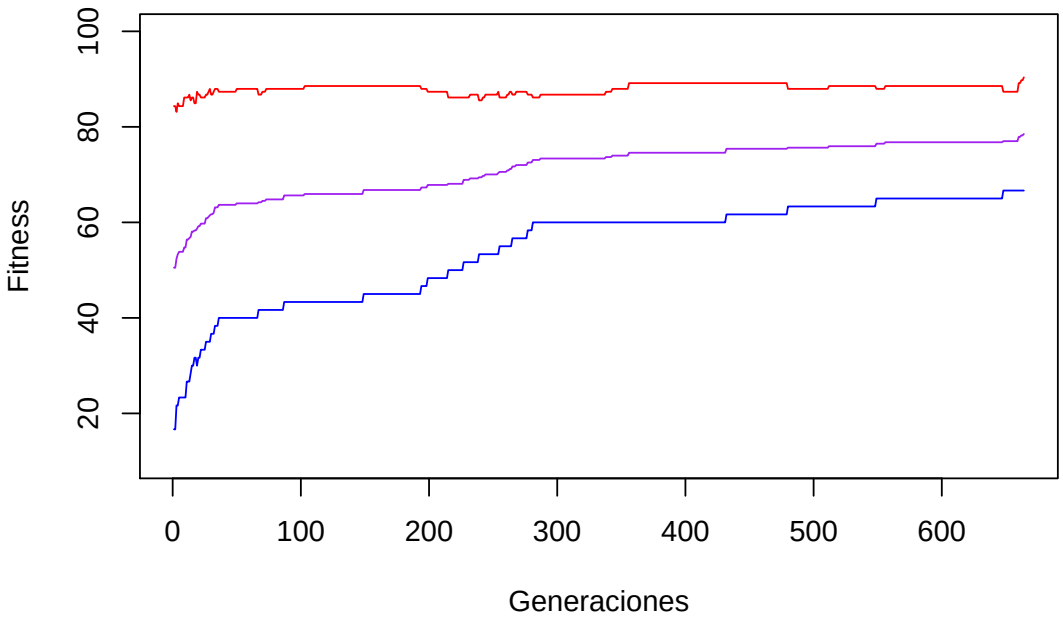


Figura 20: Evolución de la función objetivo en función del número de generaciones en AGG-BLX

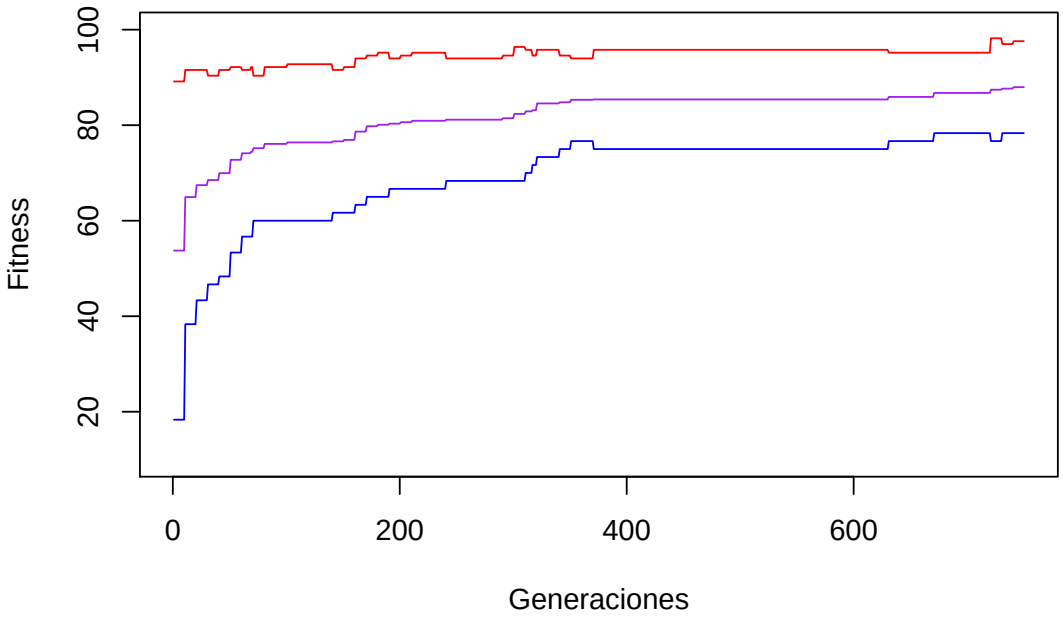


Figura 21: Evolución de la función objetivo en función del número de generaciones en AM-10-0.1mej

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.