METAHEURíSTICAS

Práctica 1

Universidad de Granada

CURSO 2016/2017

Técnicas de Búsqueda Basadas en Poblaciones para el Aprendizaje de Pesos en Características

Contenido
Búsqueda Local
Algoritmos Genéticos
Algoritmos Meméticos

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

17de abril 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 5x2. Para ello, haremos 5 particiones distintas de los datos a la mitad, y para cada partición, aprenderemos el vector de pesos con una mitad y lo evaluaremos con la otra. El valor de rendimiento promedio será la media de estas 10 evaluaciones.

```
for i from 1 to 5
    solucion1 = algoritmo(particion[i][1])
    fitness1[i] = f_objetivo(particion[i][2], solucion)

    solucion2 = algoritmo(particion[i][2])
    fitness2[i] = f_objetivo(particion[i][1], solucion)
end

return media(union(fitness1, fitness2))
```

La evaluación de una solución sobre una partición, o la función objetivo en sí, 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 f_objetivo(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 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)
    return sum(solucion[i]*(dato1[i]-dato2[i])^2) for i from 1 to solucion.size
end
```

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
```

Mecanismos de selección

El mecanismo de selección de padres para los algoritmos genéticos dependerá de si es generacional o estacionario, aunque solo variará el número de torneos binarios, y por tanto de padres, a escoger. En los AGG haremos tantos torneos binarios como individuos haya en la población. En los AGE, haremos cuatro torneos binarios. Surgirán cuatro candidatos a padres, pero solo los dos primeros tendrán asegurado serlo. Los dos restantes lo serán solo cuando el operador de cruce no proporcione los suficientes hijos solo con dos padres. Es importante tener en cuenta que no debe hacerse un torneo binario de un individuo consigo mismo, pues si el individuo escogido es el peor de la población no estaríamos fomentando la competición y selección de soluciones de mayor calidad.

```
# Torneo binario
def torneoBinario(indiv_1, indiv_2)
    if indiv_1.solucion.fitness > indiv_2.solucion.fitness return indiv_1
    else return indiv_2
end

# Selección AGG
def seleccionAGG
    for i in 1 to poblacion.size # En el caso de los AGE, tomamos solo 4
        indiv_1 = poblacion.extraerAleatorio()
        indiv_2 = poblacion.extraerAleatorio()
        padres.añadir(torneoBinario(indiv_1, indiv_2))
    end
end
```

Operadores de cruce y mutación

El operador de mutación utilizado es el mismo que el operador de generación de vecinos. En este caso, cada vez que se muta un gen, el gen se identifica con un atributo de una solución (individuo), al cual le corresponderá el índice sobre el que aplicaremos el operador de movimiento.

En cuanto a los operadores de cruce, utilizaremos dos. El cruce aritmético de dos soluciones originará una nueva solución cuyas componentes son la media aritmética de las componentes de los padres. Por otra parte, el cruce BLX-0.3 generará dos soluciones en las que cada gen está distribuido uniformemente sobre el intervalo determinado por los genes de los padres ampliado una cantidad proporcional a la longitud de dicho intervalo (en este caso, la cantidad es 0.3). Esta ampliación incentiva la diversidad, aumentando la capacidad de exploración.

```
def cruceAritmetico(sol1, sol2)
  hijo[i] = (sol1[i]+sol2[i])/2 for i from 1 to sol1.size
  return hijo
end

def cruceBLX03(sol1,sol2)
  for 1 from 1 to sol1.size
      cmax = max(sol1[i],sol2[i])
      cmin = min(sol1[i],sol2[i])
      l = cmax - cmin
      inf = cmin - 0.3*l
      sup = cmax + 0.3*l
      hijo1[i] = min(1,max(0,uniforme(inf,sup).nuevoNumero()))
      hijo2[i] = min(1,max(0,uniforme(inf,sup).nuevoNumero()))
  end

  return [hijo1, hijo2]
end
```

Métodos de búsqueda

Búsqueda local

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
```

Algoritmos genéticos

Tanto para los algoritmos generacionales como estacionarios, la estrategia de resolución es la misma, y se resume 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 también es común a ambos tipos de algoritmos, y se descompone en cuatro grandes bloques, como se muestra en las siguientes líneas:

```
def nuevaGeneracion()
    seleccion() # Selección de padres
    cruce()     # Generación de hijos
    mutacion()  # Mutación de hijos
    reemplazo() # Sustitución de la población
end
```

El mecanismo de selección ya se comentó en la sección anterior y solo se diferencia en el número de torneos binarios, y por tanto de padres, que se escogen. El mecanismo de cruce presenta más variaciones entre generacional y estacionario, y además dependerá del operador de cruce seleccionado.

Si el algoritmo es generacional, teniendo en cuenta que la lista de padres ya tiene una componente aleatoria dada por la selección, para ahorrar en cómputo de números aleatorios, se eligirán para realizar el cruce los M primeros individuos de la población, donde M es el valor esperado de individuos dado por la probabilidad de cruce. Los emparejamientos serán entre cromosomas consecutivos (primero con segundo, tercero con cuarto, ...). Si hicieran falta más hijos (es decir, si el operador de cruce fuera el aritmético), para favorecer la diversidad se harán nuevos tipos de cruces, emparejando el cromosoma i-ésimo con el cromosoma i-ésimo empezando desde el final. Finalmente, los hijos se rellenan con los padres que no han participado en el cruce, para completar la población, ya que estos intervienen en la mutación también.

```
def AGG.cruce()
    num_cruces = ceil(prob_cruce * padres.size / 2)
    indice_ultimo_hijo = 2*num_cruces -1

    for i from 0 to num_cruces-1
        # Indices de los padres a cruzar
        # Consecutivos
        i1 = 2*i
        i2 = 2*i+1
        # Para el cruce aritmético
        i3 = i
```

```
        i4 = indice_ultimo_hijo - i

        if operador_cruce.tipo = ARITMETICO
            hijos.añadir(operador_cruce(padres[i1],padres[i2]))
            hijos.añadir(operador_cruce(padres[i3],padres[i4]))
        else if operador_cruce.tipo = BLX
            hijos.añadir(operador_cruce(padres[i1],padres[i2]))
        end

    end

    # Completar población
    hijos.añadir(padres[i]) for i > indice_ultimo_hijo
end
```

En cuanto al cruce estacionario, puesto que partimos de la selección de dos únicos padres, o cuatro si el operador solo proporciona un hijo, simplemente tendremos que añadir a los hijos los cromosomas resultantes de aplicar las operaciones de cruce. En este caso, no se completa la población, y solo los dos hijos generados interverdrán en las mutaciones.

```
def AGE.cruce()
    if operador_cruce.tipo = ARITMETICO
        hijos.añadir(operador_cruce(padres.primero,padres.segundo))
        hijos.añadir(operador_cruce(padres.tercero,padres.cuarto))
    else if operador_cruce.tipo = BLX
        hijos.añadir(operador_cruce(padres.primero,padres.segundo))
    end
end
```

El procedimiento de mutación es análogo en ambos tipos de algoritmos, y de nuevo para reducir el cómputo de números aleatorios elegimos en cada generación tantos genes a mutar como indique el valor esperado para la probabilidad de mutación. En este caso, si la esperanza es menor que 1, tendremos que mutar cada cierto número de generaciones una vez, mientras que si es mayor que 1, mutaremos tantas veces como indique la esperanza en cada generación. Una vez decidido el número de mutaciones, para cada mutación escogemos al azar un cromosoma y un gen a mutar dentro de ese cromosoma, y aplicamos el operador de mutación indicado en la sección anterior.

```
def mutacion()
    esperanza = prob_mutacion * hijos.size * num_genes
    gen_muts = ceil(1.0/esperanza) # Periodo de mutaciones (en generaciones)

    # Determinación de mutaciones
    # (mutamos solo si estamos en el periodo de mutación)
    if gen_muts != 0 and generacion_actual mod gen_muts = 0
        num_mutaciones = ceil(esperanza)
    else num_mutaciones = 0

    #Realizar mutaciones
    for i from 1 to num_mutaciones
        hijo_mut = hijos.escogerAleatorio()
        indice_gen = aleatorioEntre(1,num_genes)
        hijo_mut.mov(indice_gen,0.3)
    end
```

```
end
```

Finalmente, los esquemas de reemplazamiento son diferentes para cada modelo. En el modelo generacional con elitismo, hay que mantener la mejor solución de la población anterior y sustituirla por la peor de la nueva generación, y volver a actualizar la mejor solución para que pueda ser utilizada en la próxima generación. En cuanto al modelo estacionario, se comparan las dos peores soluciones de la generación anterior con los hijos obtenidos, y pasan a la nueva generación los mejores.

El pseudocódigo del remplazamiento en los AGG es el siguiente:

```
# Suponemos que la mejor solución está guardada en mejor_solucion
# mejor_solucion se calcula en el propio método y en el algoritmo
# de inicialización de la población
def AGG.reemplazo()
    # Para reemplazar peor solución y recalcular mejor solución
    peor_val = 101.0
    mejor_val = -1.0
    indice_peor = -1
    indice_peor = -1

    for i from 1 to poblacion.size
        poblacion[i] = hijos[i] # Reemplazamiento

        # Actualización del peor valor
        if poblacion[i].fitness < peor_val
            peor_val = poblacion[i].fitness
            indice_peor = i
        end
        # Actualización del mejor valor
        if poblacion[i].fitness > mejor_val
            mejor_val = poblacion[i].fitness
            indice_mejor = i
        end
    end

    #Elitismo
    poblacion[indice_peor] = mejor_solucion

    #Actualización de la mejor solución
    if poblacion[indice_mejor].fitness > mejor_solucion.fitness
        mejor_solucion = poblacion[indice_mejor]
    end
end
```

Y a continuación se muestra el pseudocódigo del reemplazamiento para los AGE:

```
def AGE.reemplazo()
    # Hijos mejor y peor de los dos generados
    mejor_hijo = hijos.mejor()
    peor_hijo = hijos.peor()

    indice_peor = -1
    indice_2do_peor = -1
```

```
    peor_valor = 101.0
    peor_valor_2 = 102.0

    # Recorremos la población en busca de los peores
    for i from 1 to poblacion.size
        # Nuevo peor valor, el actual pasa a ser el segundo peor valor
        if poblacion[i].fitness < peor_valor
            peor_valor_2 = peor_valor
            indice_2do_peor = indice_peor
            peor_valor = poblacion[i].fitness
            indice_peor = i
        # Nuevo segundo peor valor
        else if poblacion[i].fitness < peor_valor_2
            peor_valor_2 = poblacion[i].fitness
            indice_2do_peor = i
        end
    end

    # Si los dos hijos son peores no reemplazamos
    if mejor_hijo < poblacion[indice_peor]
        hijos.borrar()

    # El mejor hijo es mejor que el peor, pero no que el 2º peor
    # o el mejor hijo es mejor que los dos peores, y el segundo
    # hijo no es mejor que ninguno (1 reemplazo)
    else if mejor_hijo < poblacion[indice_2do_peor]
         or peor_hijo < poblacion[indice_peor]
             poblacion[indice_peor] = mejor_hijo

    # Los dos hijos son mejores (2 reemplazos)
    else
        poblacion[indice_peor] = mejor_hijo
        poblacion[indice_2do_peor] = peor_hijo
    end
end
```

Un último detalle a comentar en los algoritmos genéticos es las llamadas a la función objetivo. En principio, tenemos que llamar a la función objetivo tanto en las fases de cruce como de mutación, ya que en ambas se generan individuos. En la práctica, hay hijos que no van a llegar a intervenir en el reemplazamiento de la población, ya que van a pasar antes por la mutación, por lo que es posible reducir el número de llamadas y aprovecharlo en futuras generaciones. En los algoritmos estacionarios, como solo intervienen dos hijos en las mutaciones basta utilizar dos llamadas a la función objetivo en cada generación, siempre después de las mutaciones. En los algoritmos generacionales el número de llamadas en cada generación dependerá, además del número de hijos generado, de si los individuos que mutan han sido hijos o padres que han pasado de la generación anterior. Utilizando un vector que nos indique qué individuos han sido los que ha mutado podemos evaluar fácilmente los fitness de los hijos y añadir si es necesario los de las mutaciones, evitando hacer llamadas de más a la función objetiv. Estas llamadas evitadas podrán ser aprovechadas en futuras generaciones.

Algoritmos meméticos

Los algoritmos meméticos utilizados combinan los algoritmos genéticos y la búsqueda local en función de tres factores: el periodo de generaciones para el que se aplica la búsqueda local, el porcentaje de individuos a los que se aplica la búsqueda local y el modo de selección de dichos individuos (al azar o a los mejores). Se aplicarán tantas generaciones del algoritmo genético consecutivas según indique el primer parámetro, y luego según el modo de selección, se reordena la población: por orden de fitness, si el modo de selección es seleccionar los mejores, o al azar, en caso contrario. Finalmente, se escogen los primeros individuos en ese vector, según el porcentaje especificado. El procedimiento se repite mientras no se verifiquen las condiciones de parada, que en este caso vienen dadas por el número de evaluaciones de la función objetivo. A continuación se muestra el pseudocódigo:

```
# gens_ls: Periodo de generaciones para el que se aplica la BL
# porc_ls: Porcentaje de individuos a escoger para la BL
# mejores: Modo de selección
# max_evals: Criterio de parada
def AlgMemetico(gens_ls, porc_ls, mejores, max_evals)
    num_evals = 0
    AG.inicializarPoblacion()
    #Número de individuos a los que se aplicará BL:
    num_indiv_ls = porc_ls * AG.tamañoPoblacion()


    #Condición de parada
    while num_evals < max_evals
        # Realizamos tantas generaciones como se
        # hayan especificado como argumento
        for i from 1 to gens_ls
            AG.nuevaGeneracion()
        end

        # Obtenemos población para aplicar BL
        soluciones = AG.obtenerPoblacion()

        if porc_ls != 1.0 # Para evitar cómputos si la
                          # probabilidad es 1
            if mejores
                sort(soluciones) # Ordenar por fitness
            else
                shuffle(soluciones)
            end
        end

        # Aplicamos búsqueda local
        for sol in soluciones.obtenerPrimeras(num_indiv_ls)
            LS.busquedaLocal(sol,criterio_parada)
        end

        AG.actualizarPoblacion(soluciones)
        num_evals=AG.evaluaciones + LS.evaluaciones

    end
end
```

Una consideración que se ha hecho en la implementación, y que se refleja en el pseudocódigo, es la de terminar siempre el algoritmo en una generación con búsqueda local, aunque el criterio de

parada se hay podido cumplir en alguna generación intermedia. Esto nos va a proporcionar una última mejora sobre la población final obtenida (al menos, sobre la muestra de entrenamiento) y el hacer estas generaciones de más no supone un coste adicional demasiado elevado puesto que el número de evaluaciones en generaciones sin búsqueda local es bajo, y hacer solo una búsqueda local más también es asequible.

Finalmente, el criterio de parada utilizado para la búsqueda local en todas las versiones del algoritmo memético es el de haber hecho $2n$ evaluaciones, con n el número de características.

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.
- `-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: 1.6 GHz
- 4 procesadores principales, 8 procesadores lógicos
- 4 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, 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.

El primer hecho que debemos destacar de los resultados obtenidos es que, en media, los fitness obtenidos apenas varían un 3 % entre todos los algoritmos considerados, independientemente de lo que hayan conseguido aprender en la muestra de entrenamiento Dentro de las particiones de los distintos algoritmos, los fitness presentan bastantes variaciones. De hecho, salvo en el greedy y en el algoritmo aleatorio, la desviación de los fitness sobre la partición test supera en una cantidad notable a la desviación de los fitness sobre la partición de entrenamiento. Esto nos muestra que cuando los algoritmos tienen una capacidad alta de aprendizaje, el comportamiento sobre la partición test es difícil de predecir. Como se discutirá en la siguiente sección, los algoritmos están sobreaprendiendo.

En cuanto a los conjuntos de datos, 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, vemos que, como era de esperar, el greedy es el más rápido. El tiempo que nos proporciona el `1NN` nos da una medida del tiempo que tarda en evaluarse cada llamada a la función objetivo. Hay que destacar también que cuando los tiempos son pequeños, la máquina que ha realizado las ejecuciones solo ha podido proporcionar precisión en milisegundos. En los

algoritmos genéticos y meméticos, vemos que aumenta el tiempo bastante y que la desviación es muy pequeña. Esto último puede deberse a que las ejecuciones se realizaron con prioridad alta, reduciendo así los cambios de contexto que pueden producir más variaciones de tiempos. También con respecto al tiempo de algoritmos genéticos y meméticos, vemos que hay muy poca variación entre los tiempos medios de los distintos algoritmos, y teniendo en cuenta los tiempos obtenidos de la función objetivo, vemos que, fijadas las 15000 evaluaciones que se realizan en todos algoritmos, el tiempo es casi igual a 15000 veces el tiempo medio de la la función objetivo, lo que nos indica que casi todo el tiempo que consumen estos algoritmos se emplea en evaluar la función objetivo. Finalmente vemos que, aunque las búsquedas locales también están limitadas a 15000 evaluaciones, tardan bastante menos tiempo. Esto se debe a que finalizan siempre por el criterio del vecindario recorrido sin mejora, y mucho antes de las 15000 evaluaciones.

Finalmente, en cuanto a la capacidad de aprendizaje de cada algoritmo en la muestra de entrenamiento, vemos que los que mejores resultados proporcionan son los algoritmos genéticos con el operador de cruce BLX, con muy poca variación entre el modelo generacional y estacionario, al menos con la semilla utilizada. Probando con otras semillas se puede ver que el modelo generacional es ligeramente mejor. El cruce BLX mejora notablemente al aritmético, y esto puede deberse, como veremos más adelante, a que proporciona bastante más diversidad. En cuanto a los meméticos, vemos que en general no superan a los genéticos BLX. Esto puede deberse en parte a que alcanzan un número bastante menor de generaciones, debido a que muchas evaluaciones se pierden en la búsqueda local, y en parte a que es posible que la búsqueda local no tenga tiempo suficiente de generar un buen vecindario sobre el que mejorar. Por último, hay que destacar que la búsqueda local partiendo de soluciones greedy proporciona unos resultados sorprendentemente buenos, llegando a superar en ocasiones a los genéticos. El inconviente que tiene es que, salvo la pequeña componente aleatoria que tiene el orden de búsqueda de vecinos en la búsqueda local, el algoritmo `RELIEF+LS` apenas admite mejoras. En cambio, los algoritmos genéticos, variando la semilla o ejecutándolo durante más generaciones, puede continuar mejorando su solución.

Análisis del aprendizaje

Como ya hemos comentado antes, en lo que respecta al fitness obtenido sobre los datos test apenas hay diferencias entre todos los algoritmos. Esto se debe que se está produciendo sobreaprendizaje, ya que los tamaños de las particiones utilizadas son bastante pequeños y los algoritmos llegan a aprender demasiado sobre la muestra de entrenamiento. Esto implica que los algoritmos consiguen aprender características muy concretas de los datos de la muestra de entrenamiento y estos rasgos no se presentan fuera de esta partición. La consecuencia final de esto es un aumento del error de generalización, independientemente de lo que el algoritmo haya conseguido minimizar el error dentro de la muestra.

Observando los resultados obtenidos en media, vemos que el clasificador no se presta a mejorar independientemente de lo que se haya aprendido. Tampoco llega a empeorar en gran medida. De nuevo, esto se debe al tamaño de los datos y a que la elección de particiones es aleatoria. Según cómo se hayan distribuido los pocos datos que tenemos en las particiones podrá haber pequeñas mejoras o empeoramientos. En general, no podemos establecer en estos conjuntos de datos ninguna relación entre la mejora sobre los datos de entrenamiento y los datos en el test.

Para ilustrar un caso en el que se produce sobreaprendizaje, consideramos el método de búsqueda local. La búsqueda local va generando soluciones vecinas y, en caso de mejorar a la solución actual sobre la muestra de entrenamiento, la sustituye. Esto va a permitir incrementar en gran medida la función objetivo sobre la muestra de entrenamiento, hasta alcanzar un óptimo local. Sin embargo, si a la vez vamos evaluando el fitness sobre los datos test, podemos apreciar que en muchas particiones disminuye conforme se actualiza la mejor solución de la búsqueda local. En la siguiente gráfica se muestra un ejemplo de una partición en la que esto ocurre, y en la que se aprecia claramente que hay sobreaprendizaje.

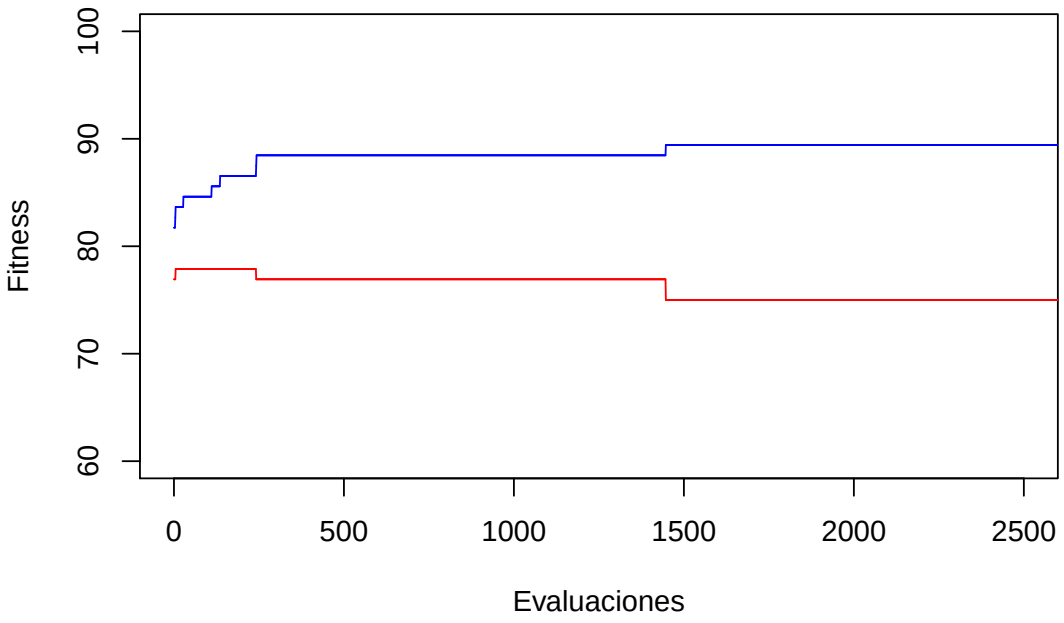


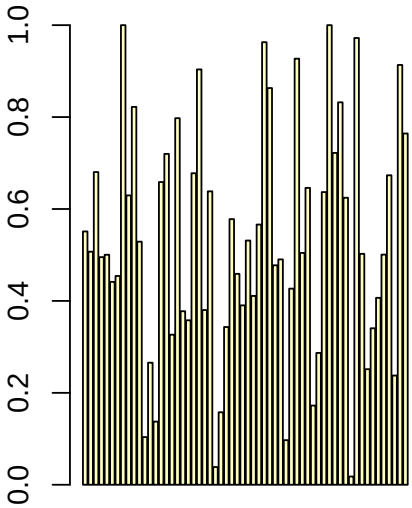
Figura 14: Evolución del fitness en sonar para una búsqueda local partiendo de una solución aleatora (semilla 5. cuarta partición en la validación cruzada). En azul, el fitness sobre la muestra de entrenamiento. En rojo, el fitness sobre la muestra test.

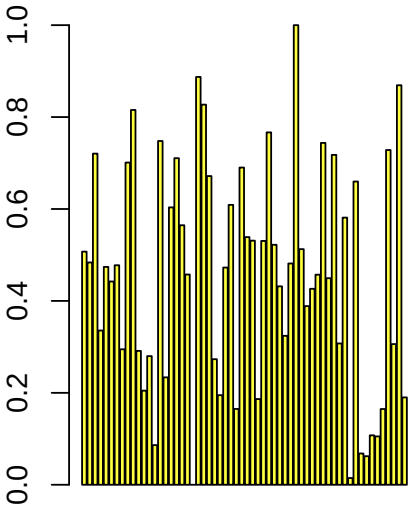
De ahora en adelante, visto el sobreaprendizaje (o simplemente la incapacidad de mejora o saturación) que se produce en los tres problemas estudiados, se centrarán los análisis en la capacidad que tienen los distintos algoritmos de aprender de la muestra de entrenamiento, y todas las consideraciones que se hagan sobre el fitness, salvo que se se indique lo contrario, serán sobre las muestras de entrenamiento.

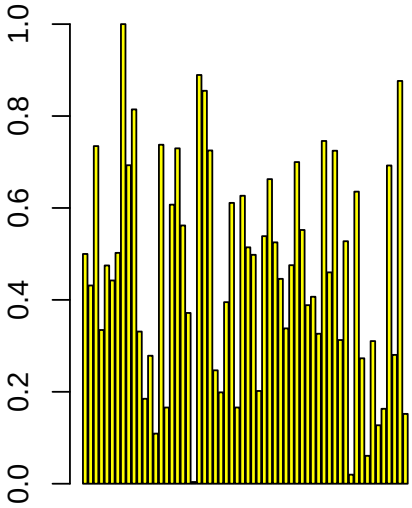
Análisis de los operadores de cruce

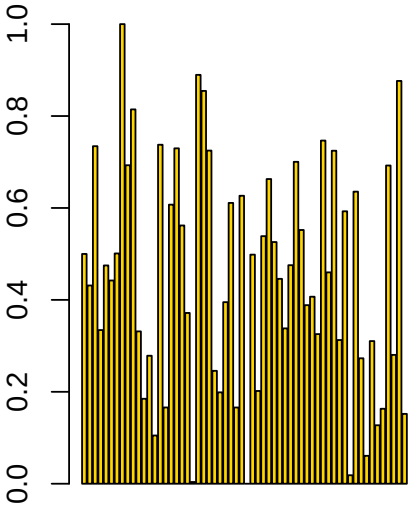
En esta sección vamos a analizar el comportamiento de las soluciones en los algoritmos genéticos según el operador de cruce utilizado. Utilizaremos en ambos casos el modelo generacional. El procedimiento realizado es el siguiente: se realiza una ejecución del `AGG` con cada operador de cruce, y para una de las particiones evaluadas, se comprueba gráficamente cómo va variando la estructura de la mejor solución conforme se avanza en las generaciones.

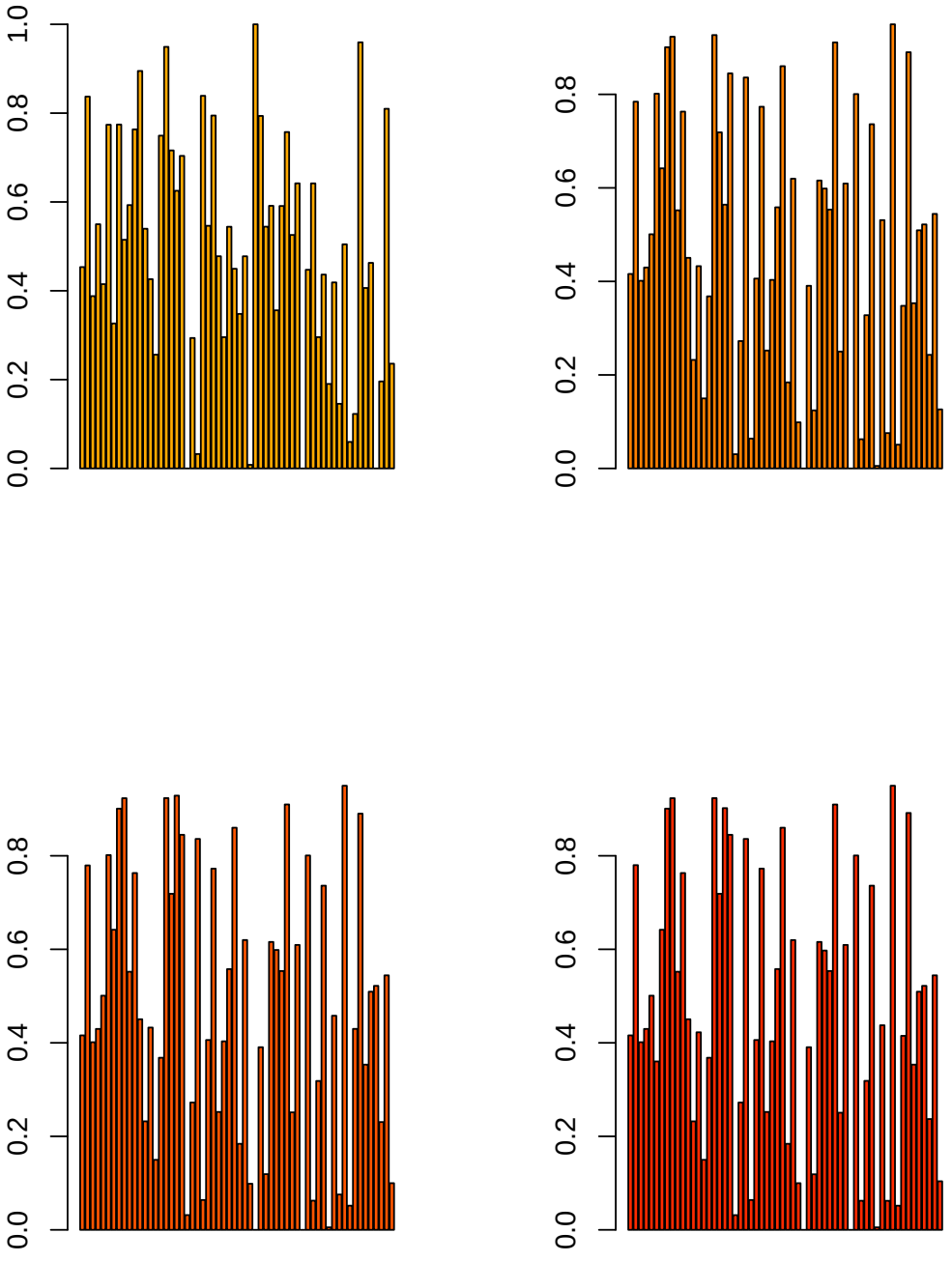
A continuación se muestran gráficamente los resultados obtenidos para el operador de cruce `BLX`:


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


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

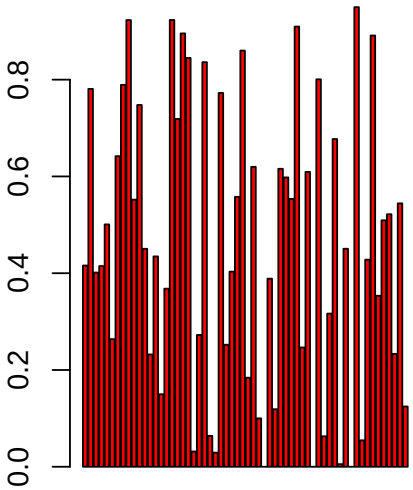


Figura 15: Evolución de las mejores soluciones en el AGG-BLX en sonar (semilla 3141592).

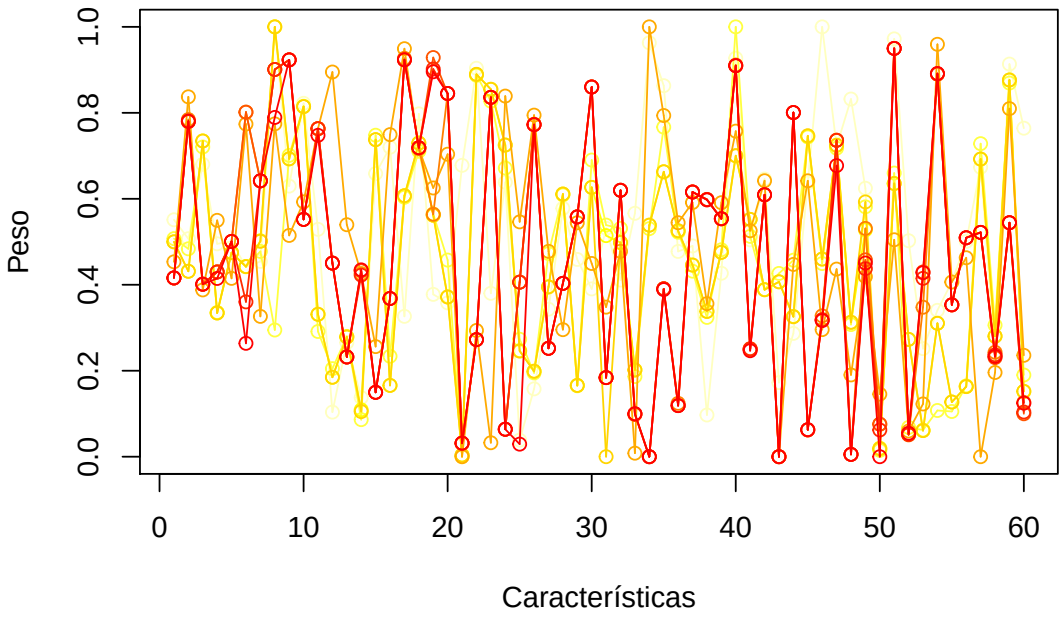
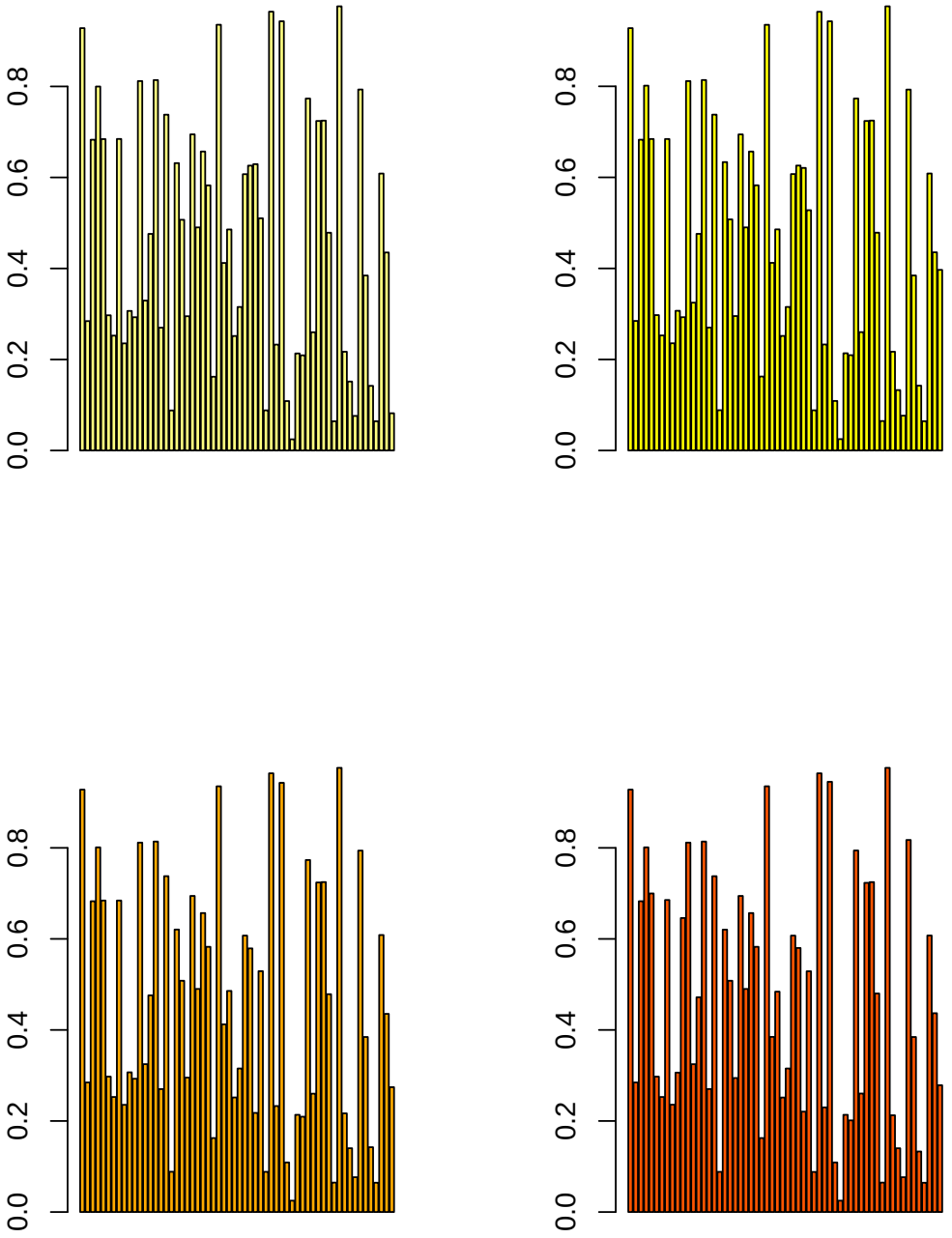


Figura 16: Comparación conjunta de la evolución de las mejores soluciones en el AGG-BLX en sonar (semilla 3141592). Los colores más cálidos representan las soluciones más evolucionadas.

Observando las figuras, podemos ver que las mejores soluciones han ido variando de forma considerable con el paso de las generaciones. También se puede comprobar que muchas de los pesos de las características se han ido extendiendo a los extremos del intervalo $[0, 1]$, más hacia el extremo inferior que al superior. Estos dos detalles nos muestran que el operador de cruce `BLX` tiene una gran capacidad de exploración, y además es un buen mecanismo para anular las características que no tienen importancia a la hora de clasificar.

A continuación realizamos el mismo procedimiento para el cruce aritmético, obteniendo los siguientes resultados:



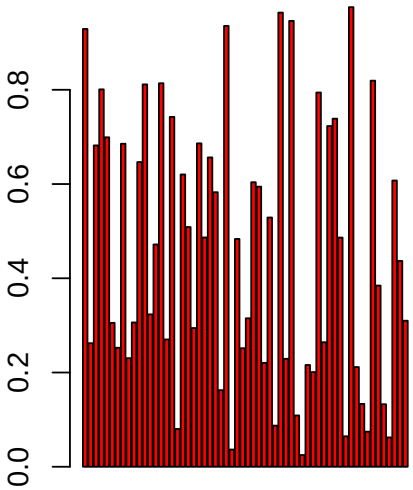


Figura 17: Evolución de las mejores soluciones en el AGG-CA en sonar (semilla 3141592).

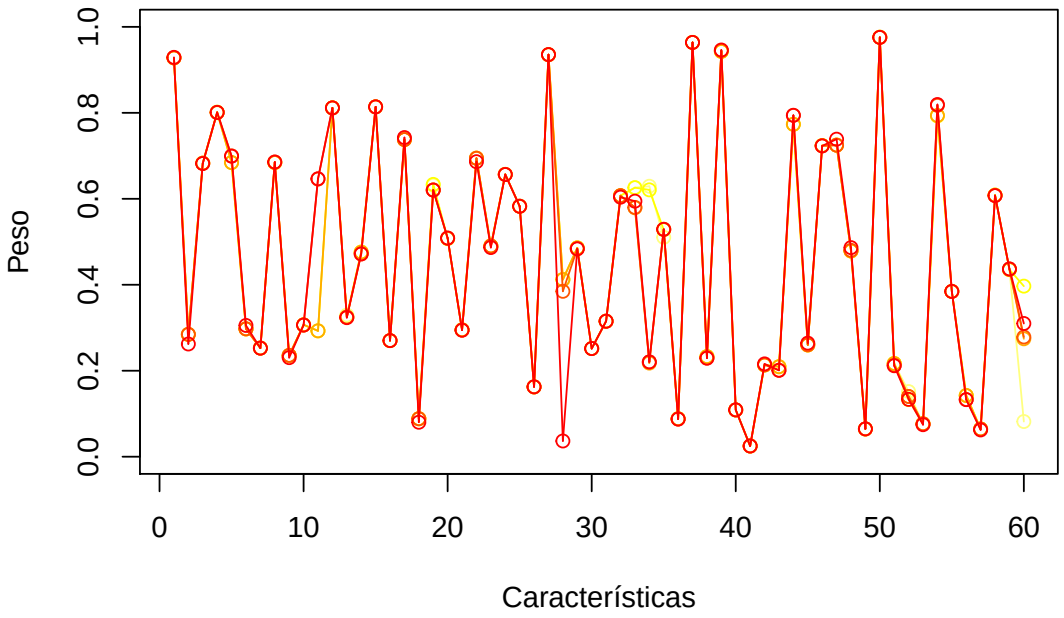


Figura 18: Comparación conjunta de la evolución de las mejores soluciones en el AGG-CA en sonar (semilla 3141592). Los colores más cálidos representan las soluciones más evolucionadas.

En este caso, en primer lugar vemos que se obtienen menos mejoras (un total de 5) que con el cruce BLX (9). Esto en general suele ocurrir, independientemente de las particiones escogidas. Por otra parte, observando la comparación conjunta de las soluciones vemos que apenas han variado unos pocos pesos desde la primera solución hasta la obtenida al final. Esto nos muestra que el cruce aritmético, en general, no nos produce soluciones que permitan mejorar el fitness. De hecho, muchos de los pocos cambios que se muestran en las figuras anteriores es posible que se hayan debido a mutaciones en vez de a la generación de nuevos hijos, restando aún más capacidad al operador de cruce.

Finalmente, comparamos de forma conjunta las soluciones obtenidas por ambos operadores:

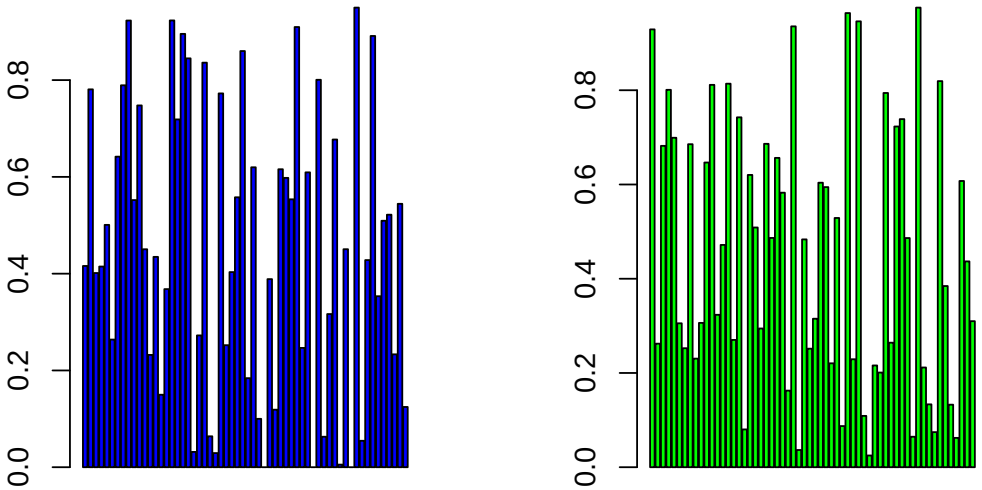


Figura 19: Comparación conjunta de las mejores soluciones obtenidas por el cruce BLX (izquierda) y aritmético (derecha).

Además de lo ya comentado sobre las distintas capacidades de exploración de ambos cruces, esta última figura nos permite apreciar ligeramente que el operador `BLX` origina pesos más dispersos en $[0, 1]$ (tiene más valores extremos), mientras que los pesos que origina el cruce aritmético tienden a estar concentrados la mayor cantidad entre el primer y el tercer cuartil.

Análisis de la evolución de algoritmos genéticos y meméticos

En esta sección, analizaremos gráficamente cómo evolucionan las poblaciones en los algoritmos genéticos. Estudiaremos dos casos distintos: el algoritmo genético estacionario y el algoritmo memético `(10,1.0)`. Dentro de este último también podremos analizar el algoritmo genético generacional que lleva incorporado. En ambos casos el operador de cruce utilizado será el `BLX`, que ya hemos visto que proporciona mejores resultados.

Para el AGE-BLX, consideraremos una población de 6 individuos. En cada generación dos hijos podrán sustituir a los peores individuos si los mejoran. Se realiza la ejecución utilizando el criterio de parada de las 15000 evaluaciones de la función objetivo. En la siguiente gráfica se muestra cómo van evolucionando los 6 individuos.

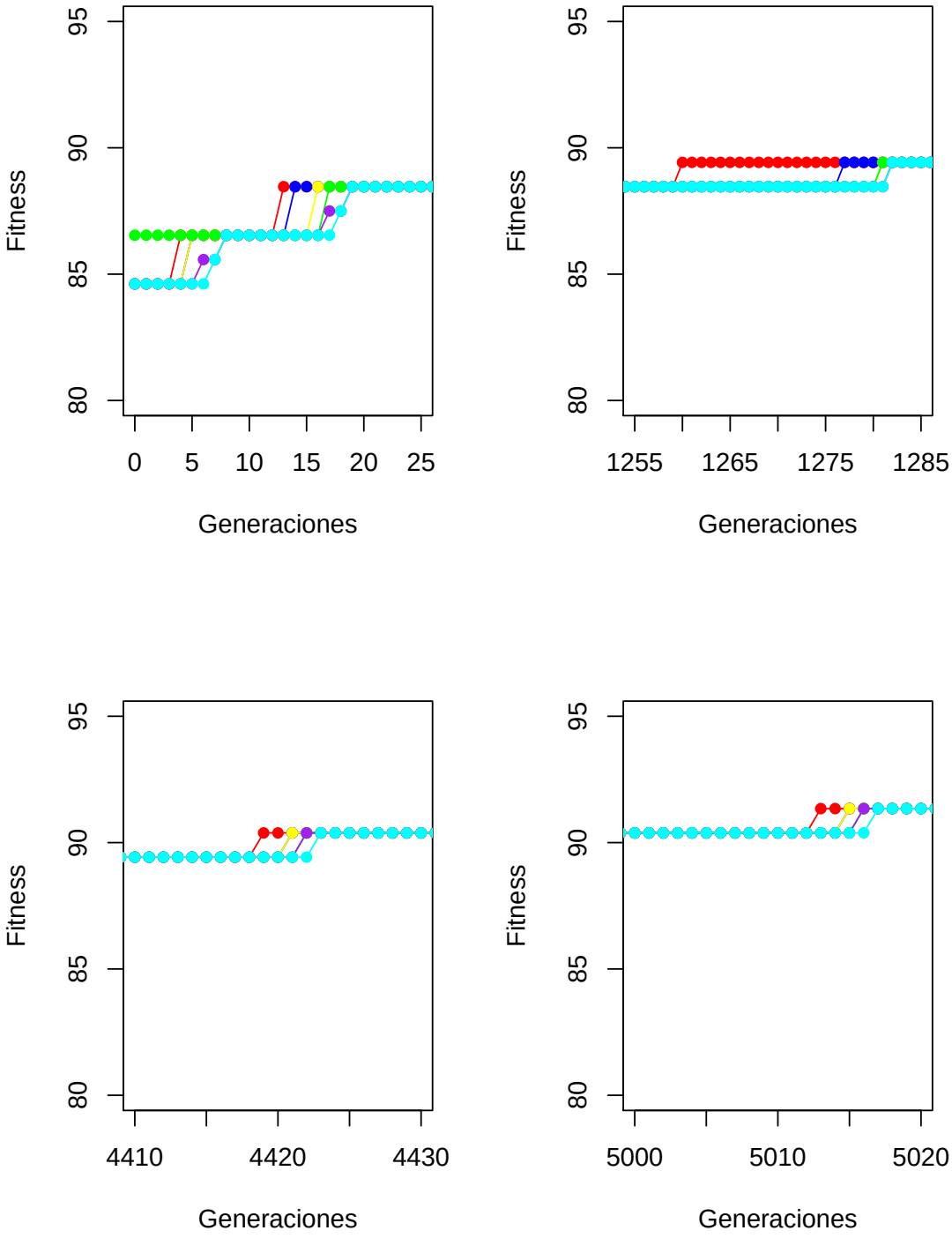


Figura 20: Evolución del fitness de una población de 6 individuos sobre ‘sonar‘ para el algoritmo genético estacionario ‘AGE-BLX‘. Cada color representa un cromosoma.

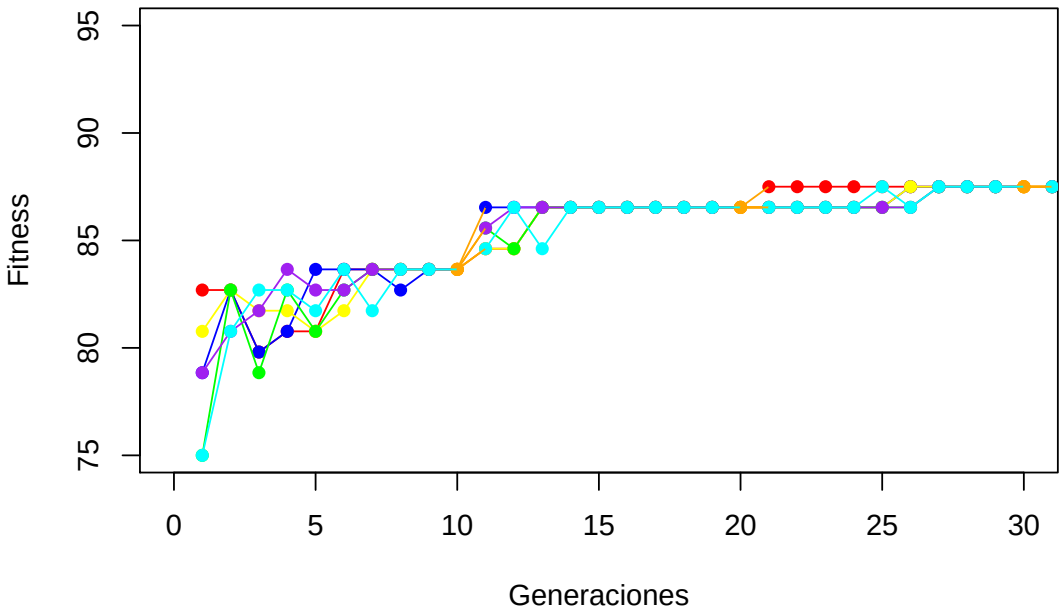
En las gráficas podemos apreciar varias cosas. En primer lugar, comprobamos que el algoritmo es elitista, ya que siempre que se alcanza un nuevo fitness máximo siempre va a haber un individuo que mantenga ese fitness. También vemos que el esquema de reemplazamiento hace que los individuos no desciendan en fitness, ya que los hijos solo se sustituyen si superan a las peores soluciones. En

cuanto al reemplazamiento, también observamos que en cada generación se modifican, a lo sumo, los fitness de dos individuos a la vez, pues es el número de hijos que se crean en cada generación.

El no empeoramiento de las soluciones, hace que tras una primera fase de evolución medianamente rápida de la población, todas las soluciones converjan a un mismo nivel de fitness, donde permanecen bastante tiempo, posiblemente debido al estancamiento en un óptimo local. Más adelante (1200 generaciones después), volvemos a ver una mejora en la población, posiblemente debida a una mutación, que permite a las soluciones abandonar el óptimo local y explorar un nuevo camino exitosamente. De nuevo la población permanece estancada durante muchas generaciones y acaba consiguiendo dos nuevas mejoras.

Un último detalle es que, como veremos en comparación con el memético (y que también ocurre con los AGG), fijado un número máximo de llamadas a la función objetivo (o fijado un tiempo máximo, ya que la mayoría del tiempo se emplea en la función objetivo), el número de generaciones que alcanza el algoritmo estacionario es mucho mayor que las que alcanzan meméticos y generacionales, ya que solo se evalúan dos nuevos hijos por generación. De esta forma, aunque el número de hijos generados pueda hacer creer que la población va a cambiar poco, el algoritmo lo compensa con un mayor número de generaciones computadas.

A continuación analizamos gráficamente el `AM-(10,1.0)`. Para ello consideramos también una población de 6 individuos, que evolucionarán con el esquema generacional, y a los que se aplicará una búsqueda local cada 10 generaciones. A continuación se muestran gráficamente los resultados para una partición:



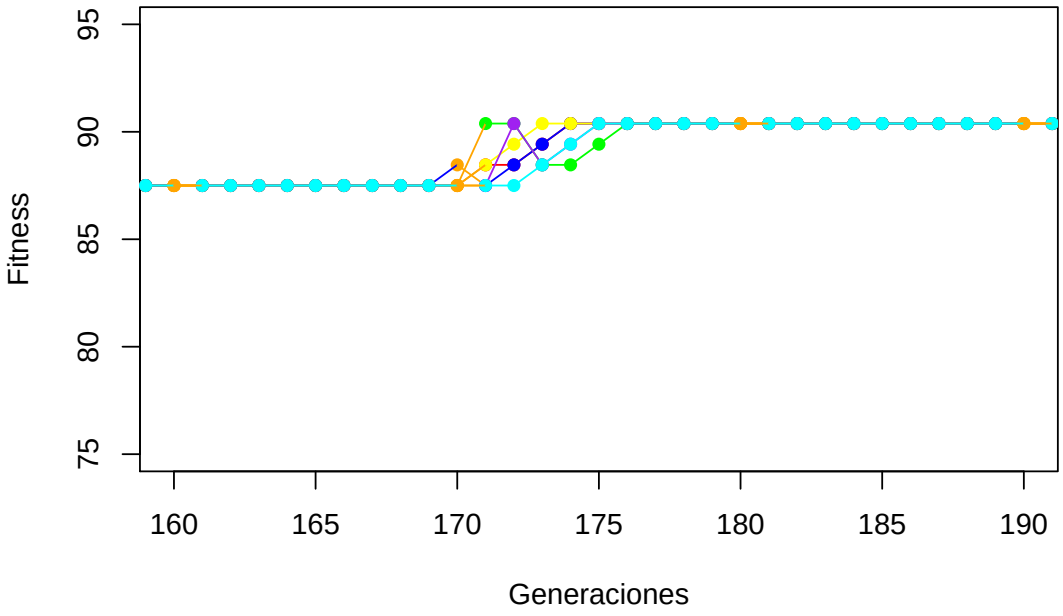


Figura 21: Evolución del fitness de una población de 6 individuos sobre 'sonar' para el algoritmo memético 'AM-(10,1.0)'. Cada color representa distinto del naranja un cromosoma. Las líneas naranjas representan el cambio en las generaciones en las que se aplica búsqueda local.

Podemos realizar también varias observaciones sobre la gráfica. En primer lugar, comprobamos también el elitismo implementado para la variante generacional, ya que , aunque en este caso se admiten soluciones peores al reemplazar, la peor obtenida siempre es reemplazada por la mejor de la generación anterior. En la gráfica, en las primeras generaciones, vemos que, aunque la mejor solución disminuye, sube otra a ocupar su lugar conservandose así la mejor solución. También vemos que se producen muchos más movimientos que en el algoritmo estacionario, sobre todo en las primeras generaciones, antes de que se estabilice la población, ya que la población entera es reemplazada en cada generación (y el 70 % al menos son nuevos individuos, sin contar mutaciones).

También vemos que, aunque se aprecia más movimiento, la población tiende a estabilizarse. En cuanto a la búsqueda local vemos que, en las primeras generaciones permite hacer avanzar a la población una cantidad considerable. Cuando la población está estabilizada vemos que apenas tiene efecto, puesto que posiblemente la población está atrapada en un óptimo local. Un hecho destacable es lo que se observa en torno a la generación 170. La población lleva bastante tiempo estabilizada, y de repente, un individuo consigue subir, posiblemente debido a una mutación. En la siguiente generación se aplica búsqueda local, y se produce un salto de calidad considerable. Hay que destacar que el individuo que había subido desciende tras la búsqueda local, es decir, es sustituido por un hijo de mala calidad, pero seguramente el otro hijo que ha producido (y que habrá sustituido al otro padre) es que ha dado el salto al nuevo fitness, gracias posiblemente a la búsqueda local. De nuevo la población vuelve a estabilizarse pendiente de nuevas mutaciones que le ayuden a explorar nuevas soluciones mejores.

Finalmente, observamos, como se comentó con los estacionarios, que el número de generaciones que se alcanzan fijado el número de evaluaciones es mucho menor, puesto que en este caso hay que evaluar al 70 % de los individuos, además de las evaluaciones resultantes de aplicar búsqueda local a todos los individuos.

Análisis de las mejores soluciones

En esta sección nos planteamos analizar cómo son las mejores soluciones obtenidas para cada problema. Para ello escogemos como algoritmo el AGG-BLX, que es el que mejores resultados proporciona en general, y nos quedamos con las tres mejores soluciones que mejor han aprendido los datos de su partición.

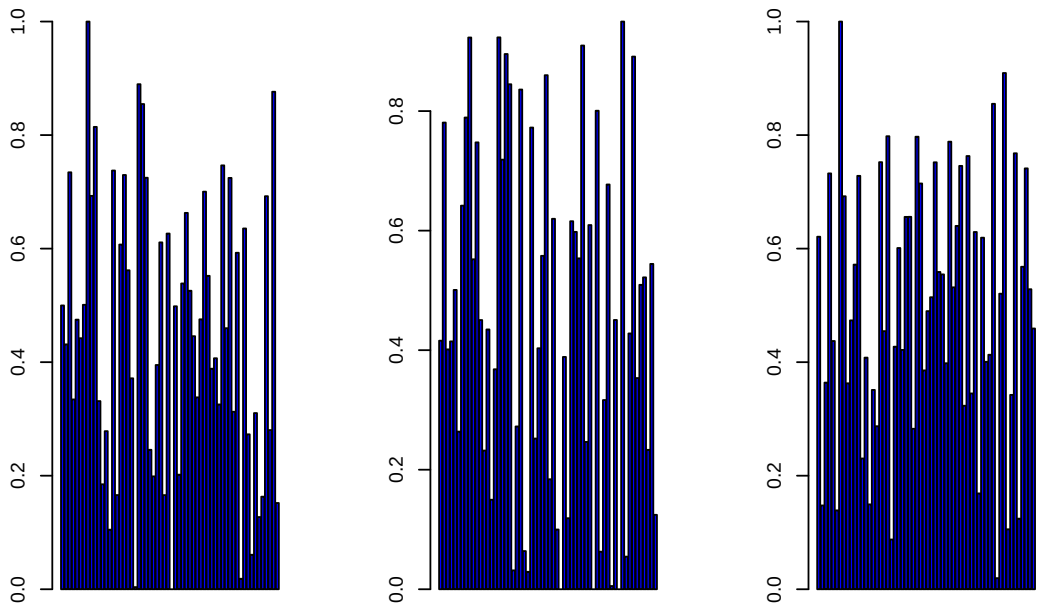


Figura 22: Representación de las tres mejores soluciones obtenidas en las particiones del algoritmo AGG-BLX para sonar(semilla 3141592).

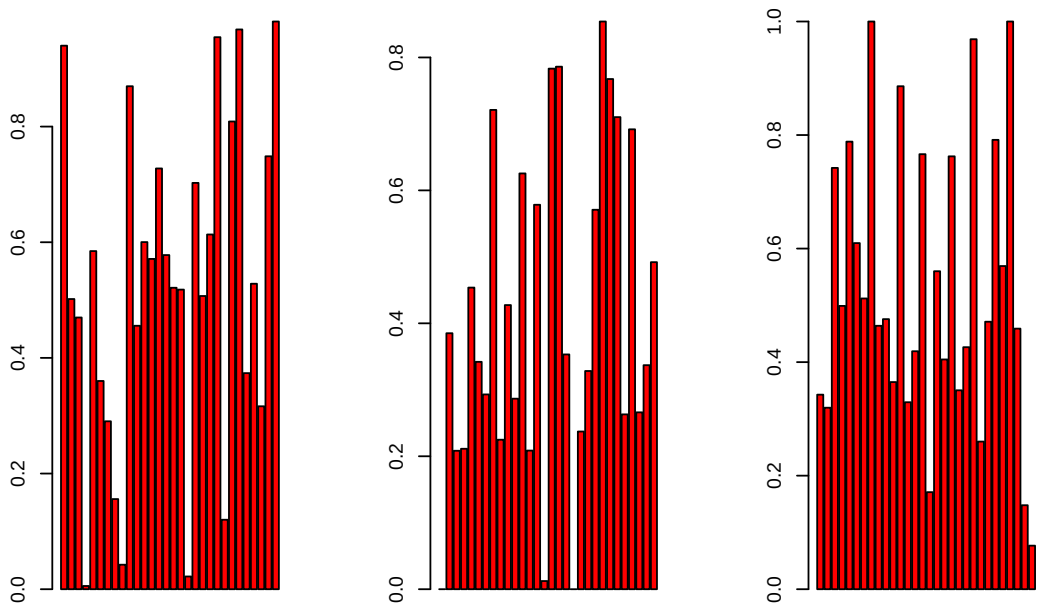


Figura 23: Representación de las tres mejores soluciones obtenidas en las particiones del algoritmo AGG-BLX para wdbc (semilla 3141592).

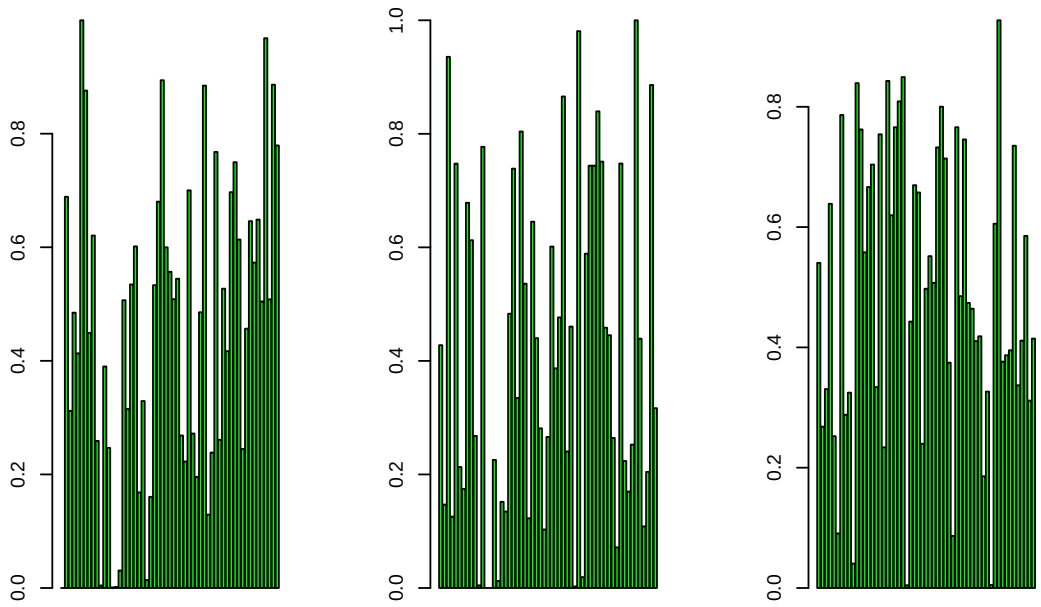


Figura 24: Representación de las tres mejores soluciones obtenidas en las particiones del algoritmo AGG-BLX para spambase (semilla 3141592).

Observando las imágenes, la conclusión es que, en general, las soluciones no comparten apenas parecidos. Esto vuelve a ser de nuevo una muestra del sobreaprendizaje que se produce. Las soluciones se adaptan demasiado bien a su partición y luego no funcionan bien sobre la partición de prueba. No se llegan a obtener soluciones similares, lo que sería un buen indicador de que el algoritmo se acerca a una solución general que aproxima bien a todo el conjunto.

Referencias

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

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