Práctica 1:

Búsquedas basadas en Poblaciones y Trayectorias simples.

Curso 2016/2017

Problema: APC

Algoritmos:

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

Francisco Javier Caracuel Beltrán

76440940-A

caracuel@correo.ugr.es

Grupo 3 (Lunes 17:30-19:30)

Índice

1. Descripción del problema

El problema elegido, en este caso, es el "***Problema del Aprendizaje de Pesos en Características***", consistente en optimizar el rendimiento de un clasificador basado en el algoritmo *1-NN* (vecino más cercano), a partir de una serie de pesos asociados a las características del problema.

Este clasificador hace uso de la distancia entre el ejemplo que se quiere clasificar y todos los ejemplos de la muestra. De este modo, se le asignará la etiqueta que tenga el ejemplo cuya distancia sea menor. Para el cálculo de esta distancia, se tienen en cuenta los pesos asociados a las características que se han comentado anteriormente.

La práctica consiste, principalmente, en generar multitud de conjuntos de pesos a través de distintas técnicas y obtener, finalmente, un conjunto de pesos que consigue la tasa de acierto más elevada.

Las distintas técnicas para generar esos conjuntos de pesos y que se explican en próximas secciones son:

- Algoritmo Aleatorio.
- Algoritmo Relief.
- Algoritmo Aleatorio + Búsqueda Local.
- Algoritmos Genéticos Generacionales con dos tipos de cruces (BLX y cruce aritmético).
- Algoritmos Genéticos Estacionarios con dos tipos de cruces (BLX y cruce aritmético).
- Algoritmos Meméticos (Algoritmos Genéticos + Búsqueda Local).

2. Descripción de los algoritmos comunes

A continuación, se detallan todas las consideraciones comunes a los distintos algoritmos.

a) Descripción del esquema de representación de soluciones:

La solución W vendrá dada por un contenedor de números reales normalizados en [0,1] y será de tamaño N.

Se entiende N como el número de atributos con el que se codifican los ejemplos del problema que se esté resolviendo.

$$W = (w_1, w_2, w_3, \ldots, w_N) \qquad \text{donde } w_i \in [0,1]$$

w_1	w_2	…	w_{n-1}	w_n

Un 1 en la posición w_i indica que la característica $_i$ se considera completamente en el cálculo de la distancia.

Un 0 en la posición w_i indica que la característica $_i$ no se considera en el cálculo de la distancia.

Si existe algún valor intermedio, gradúa el peso que tiene éste en esa característica, ponderando su importancia.

b) Descripción en pseudocódigo de la función objetivo:

```
well = 0
for i in numInstances
{
    if label[i] == NN1(data[i])
    {
        well++
    }
}
return well/numInstances
```

c) Descripción en pseudocódigo del operador de generación de vecino/mutación:

```
w = current_solution
posMutate = randomInt(1, numAttributes)
w[posMutate] = w[posMutate] + NormalDistribution(mean, standard_deviation)
truncate(w[posMutate], 0, 1)
return w
```

*Usado en la Búsqueda Local y en los Algoritmos Genéticos.

d) Descripción en pseudocódigo de la generación de soluciones aleatorias:

```
wRandom = container(numAttributes)
for i in numAttributes
{
    wRandom[i] = randomFloat(0, 1)
}
return wRandom
```

e) Descripción en pseudocódigo del mecanismo de selección de los AGs:

```
selection = container(numParents)
population = current_population
for i in numParents
{
    pos1 = randomInt(1, numPopulation)
    pos2 = randomInt(1, numPopulation)
    if success(population[pos1]) >= success(population[pos2])
        selection[i] = population[pos1]
    else
        selection[i] = population[pos2]
}
return selection
```

f) Descripción en pseudocódigo de los operadores de cruce de los AGs:

- Cruce BLX:

```
w1 = chromosome1
w2 = chromosome2
for i in numAttributes
{
    cMax = max(w1[i], w2[i])
    cMin = min(w1[i], w2[i])
    I = cMax - cMin
    w1[i] = randomFloat(cMin - I*crossAlpha, cMax + I*crossAlpha)
    w2[i] = randomFloat(cMin - I*crossAlpha, cMax + I*crossAlpha)
    truncate(w1[i], 0, 1)
    truncate(w2[i], 0, 1)
}
return w1, w2
```

- Cruce aritmético:

```
w = chromosome
w1 = chromosome1
w2 = chromosome2
for i in numAttributes
{
    w[i] = (w1[i]+w2[i])/2
}
return w
```

3. Descripción de los algoritmos específicos

A continuación, se detalla la descripción en pseudocódigo de todas aquellas operaciones relevantes de cada algoritmo.

a) Algoritmo Búsqueda Local:

```
// Contador del número de evaluaciones
iEval = 0
// Contador del número de vecinos sin mejorar
iNeig = 0
// Mejor acierto
pF = 0
w = current_solution
while iEval<numEvalMax && iNeig<numNeig*numAttributes
{
    wNew = w
    wNew = mutate(wNew)
    if evaluate(wNew) > pF
    {
        pF = evaluate(wNew)
        w = wNew
        iNeig = 0
    }
    iEval = iEval + 1
    iNeig = iNeig + 1
}
```

Descripción en lenguaje natural:

- *iEval* es el contador que se utiliza para controlar el número de iteraciones que puede hacer.
- *numEvalMax* es el número máximo de iteraciones que tiene permitido hacer. En el caso de esta práctica, este límite es 15.000.
- *iNeig* es el contador que se utiliza para controlar el número de iteraciones que transcurren sin que una mutación mejore a la mejor solución actual.
- *numNeig* es el número que, multiplicado al número de atributos del problema que se esté tratando, ofrecerá el número máximo de iteraciones que tiene permitido hacer sin que un conjunto de mutaciones mejore a la mejor solución actual. En el caso de esta práctica es 20. Si el problema a tratar tuviera 60 atributos, el límite máximo de iteraciones que puede realizar sin mejorar sería 1.200.
- *pF* contiene la tasa de acierto de la mejor solución obtenida.

El algoritmo de Búsqueda Local realiza una serie de mutaciones partiendo de la solución que recibe.

Estará realizando mutaciones mientras que no supere el límite máximo de mutaciones (o límite máximo de evaluaciones de la mutación creada) permitidas y, además, mientras consiga una mejora de la mejor solución obtenida hasta el momento cumpliendo las condiciones descritas anteriormente.

Por cada mutación hace una llamada a la función evaluación y comprueba que la tasa de acierto de la mutación, sea mayor que la tasa de acierto de la mejor solución actual. En el caso de que la mutación mejore la tasa de acierto, se actualizarán los datos y, a partir de ese momento, la mejor solución es la mutación.

Finalmente, el algoritmo termina devolviendo la mejor solución encontrada, ya sea la mutación o la solución inicial, si ninguna mutación la ha mejorado.

b) Algoritmo Relief:

```
data = instances
w = initialize(numAttributes, 0)
for i in numInstances
{
    posEnemy = searchEnemy(data[i])
    posFriend = searchFriend(data[i])
    for j in numAttributes
    {
        enemy = data[i][j] - data[posEnemy][j]
        friend = data[i][j] - data[posFriend][j]
        w[j] = w[j] + |enemy| - |friend|
    }
}
w = normalize(w)
return w
```

Descripción en lenguaje natural:

- *data* contiene todos los ejemplos de la muestra que se está tratando.
- *w* es un contenedor de números reales inicializados en 0 y de longitud igual al número de atributos del problema.

Este algoritmo comienza inicializando un vector de pesos a 0.

Recorre todos los ejemplos de la muestra y por cada uno de ellos calcula la posición que ocupa su enemigo más cercano (otro ejemplo cuya clase es diferente al del ejemplo actual) y la posición que ocupa su amigo más cercano (otro ejemplo cuya clase es igual al del ejemplo actual).

Cuando ya ha calculado la posición de su enemigo/amigo más cercano, recorre todos los atributos de *w* y le suma la diferencia entre el ejemplo actual y su enemigo y le resta la diferencia entre el ejemplo actual y su amigo.

Para terminar, debe normalizar todos los datos de *w* porque pueden encontrarse algunos que sean negativos (se truncan a 0) o algunos que sean mayores de 1.

Devuelve *w* que contiene una solución al problema.

c) Algoritmo Genético Generacional:

```
population = currentPopulation
successPopulation = currentSuccessPopulation
numCrosses = round(probability_of_crossing*size(population))
numMutations = round(probability_of_mutation*size(population)*numAttributes)
iEval = 0
while iEval < numEval
{
    newPopulation = selection()
    newSuccessPopulation = container(size(newPopulation))
    for i in numCrosses
    {
        newPopulation[i] = cross(newPopulation[i])
    }
    for i in numMutations
    {
        newPopulation[i] = mutate(newPopulation[i])
    }
    for i in size(newPopulation)
    {
        equals = false
        j = 0
        while not equals && j<size(population)
        {
            if newPopulation[i] == population[j]
                equals = true
            j = j + 1
        }
        if equals
            newSuccessPopulation[i] = successPopulation[j-1]
        else
            newSuccessPopulation[i] = evaluate(newPopulation[i])
    }
    if first(population) > last(newPopulation)
        update(first(population), last(newPopulation))
    population = newPopulation
    successPopulation = newSuccessPopulation
    iEval = iEval + size(population)
}
return bestSolution(), bestSucces()
```

Descripción en lenguaje natural:

- *iEval* es el número de evaluaciones que se realizan. Por cada iteración se realizan tantas evaluaciones como nuevos cromosomas hay en la población. En el caso de la práctica se tiene una población con 30 cromosomas.
- *numEval* es el número máximo de evaluaciones permitidas. En el caso de la práctica se pueden hacer 15.000 evaluaciones, lo que dan lugar a un total de 500 generaciones de cromosomas nuevas.

- *population* es un contenedor donde se encuentran todos los cromosomas actuales con los que dispone el algoritmo.
- *successPopulation* es un contenedor del tamaño de la población que contiene la tasa de acierto correspondiente a cada cromosoma.
- *numCrosses* es el número de cruces que se pueden realizar en cada iteración.
- *numMutations* es el número de mutaciones que se pueden realizar en cada iteración.

El algoritmo comienza con un conjunto de cromosomas aleatorios.

En cada generación hay que cruzar y mutar una serie de cromosomas en base a una probabilidad. Calcular qué cromosomas se cruzan o mutan directamente es muy ineficiente y por eso se calcula el número exacto de cromosomas que se van a cruzar o mutar. Como la selección es aleatoria, ya se tiene en cuenta esa aleatoriedad para realizar las operaciones.

En mi práctica en particular, el método que ejecuta el Algoritmo Genético Generacional, dispone de un valor que indica el cruce que debe realizar. De este modo, cuando se realiza la llamada al cruce, será el cruce *BLX* o el *cruce aritmético* dependiendo del valor que reciba. También se tiene en cuenta el tipo de cruce que debe realizar para la selección, ya que, si es el cruce *BLX*, la selección devuelve tantos padres como tamaño tenga la población. Si es el *cruce aritmético*, la selección devolverá el doble de padres del tamaño de la población, al generar en este cruce, un solo hijo por cada dos padres.

Otro aspecto a tener en cuenta es el número de evaluaciones. Se ha comprobado que se tarda menos tiempo de ejecución en comparar si un cromosoma es igual que otro ya existente en la población anterior, que en evaluar uno de ellos. Esto hace que cuando ya se han cruzado y mutado los correspondientes cromosomas, se compare si ya existía anteriormente. En el caso de que existiera, se le aplica la tasa de acierto que tuviera el cromosoma que es igual. Si no, se realizará la evaluación.

Para conservar el elitismo de la población, antes de terminar el tratamiento de la generación, se comprueba si el mejor de la población anterior es mejor que el peor de la población generada. En caso de que así sea, se elimina el peor de la nueva generación y se añade el mejor de la anterior.

El algoritmo termina devolviendo el mejor cromosoma generado y su tasa de acierto.

d) Algoritmo Genético Estacionario:

```
population = currentPopulation
successPopulation = currentSuccessPopulation
numMutations = round(probability_of_mutation*size(population)*numAttributes)
iEval = 0
while iEval < numEval
{
    newPopulation = selection(numParents)
    newSuccessPopulation = container(size(newPopulation))
    newPopulation = cross(newPopulation)
    for i in numMutations
    {
        newPopulation[i] = mutate(newPopulation[i])
    }
    for i in size(newPopulation)
    {
        equals = false
        j = 0
        while not equals && j<size(population)
        {
            if newPopulation[i] == population[j]
                equals = true
            j = j + 1
        }
        if equals
            newSuccessPopulation[i] = successPopulation[j-1]
        else
            newSuccessPopulation[i] = evaluate(newPopulation[i])
    }
    if best(newSuccessPopulation, last(successPopulation, 2))
        update(newPopulation, newSuccessPopulation,
                last(population, 2), last(successPopulation, 2))
    iEval = iEval + 2
}
return bestSolution(), bestSucces()
```

Descripción en lenguaje natural:

- *iEval* es el número de evaluaciones que se realizan. Por cada iteración se realizan tantas evaluaciones como nuevos cromosomas hay en la población. En el caso de la práctica se tiene una población con 30 cromosomas.
- *numEval* es el número máximo de evaluaciones permitidas. En el caso de la práctica se pueden hacer 15.000 evaluaciones, lo que dan lugar a un total de 500 generaciones de cromosomas nuevas.
- *population* es un contenedor donde se encuentran todos los cromosomas actuales con los que dispone el algoritmo.
- *successPopulation* es un contenedor del tamaño de la población que contiene la tasa de acierto correspondiente a cada cromosoma.

- *numMutations* es el número de mutaciones que se pueden realizar en cada iteración.

El algoritmo comienza con un conjunto de cromosomas aleatorios.

En cada generación se crean dos cromosomas y estos dos cromosomas se cruzan. Debido a esto no se debe calcular qué cantidad de cromosomas son los que se cruzarán. El número de mutaciones que se van a realizar sí se calculan, tal y como sucede en los *Algoritmos Genéticos Generacionales*.

Por cada generación, se hace una selección. Esta selección devuelve dos padres en el caso de que se vaya a realizar el cruce *BLX* y devolverá cuatro padres si se va a realizar el *cruce aritmético*.

Una vez que se tienen los cromosomas cruzados, siempre se tendrán dos cromosomas. Será este momento en el que se mutarán los que correspondan.

Al igual que ocurre con los *Algoritmos Genéticos Generacionales*, se comprueba si alguno de estos dos cromosomas es igual a otro ya existente. Si son iguales, se le aplica la tasa de acierto que tenga su igual y, si no, se hace uso de la función *evaluación*.

Cuando ya se tienen los dos cromosomas cruzados, mutados y evaluados, se comparan con los dos peores de la población anterior. La nueva generación la componen todos los cromosomas de la generación anterior, excepto los dos últimos. Estos dos últimos se compararán con los dos nuevos cromosomas generados y solo los dos mejores de los cuatro que se comparan, forman parte de la nueva generación.

El algoritmo termina devolviendo el mejor cromosoma generado y su tasa de acierto.

e) Algoritmo Memético:

```
    .
    .
    .
        if equals
            newSuccessPopulation[i] = successPopulation[j-1]
        else
            newSuccessPopulation[i] = evaluate(newPopulation[i])
    }
    evaluations = LocalSearch(newPopulation, newSuccessPopulation)
    iEval = iEval + evaluations
    if first(population) > last(newPopulation)
        update(first(population), last(newPopulation))
    .
    .
    .
```

Los *Algoritmos Memético*s son *Algoritmos Genéticos* a los que se le aplica una *Búsqueda Local*. En este caso, el esquema del *Algoritmo Memético* es similar al del *Algoritmo Genético Generacional*, salvo que después de conseguir las evaluaciones de los cromosomas y antes de realizar la operación de conservación del elitismo de la población, se aplica la *Búsqueda Local* a una serie de cromosomas.

A qué serie de cromosomas se le aplica la Búsqueda Local, depende del tipo de *Algoritmo Memético* que se quiera realizar.

A grandes rasgos y suponiendo que se dispone de un contenedor que indique la posición del cromosoma al que se le va a aplicar la *Búsqueda Local* (posLS), el pseudocódigo correspondiente al método *LocalSearch(newPopulation, newSuccessPopulation)* del *Algoritmo Memético* sería:

```
numEval = 0
for i in size(newPopulation)
{
    if posLS[i] == true
    {
        newPopulation[i] = getSolutionLocalSearch(newPopulation[i])
        newSuccessPopulation[i] = getSuccessLocalSearch(newPopulation[i])
        numEval = numEval + getEvalLocalSearch(newPopulation[i])
    }
}
return numEval
```

Cabe destacar, que el número de evaluaciones permitidas es 15.000, por lo que también se deben contar las evaluaciones que se realizan en la *Búsqueda Local*, aumentando el contador de éstas cuando se termina la llamada al método.

4. Descripción en pseudocódigo del algoritmo de comparación

Para hacer uso del algoritmo de comparación *1-NN* primero se debe definir el método que se encarga de calcular la *distancia euclídea* entre dos ejemplos de la muestra:

```
e1 = example1
e2 = example2
w = current_solution
res = 0
for i in numAttributes
{
    sub = e1[i] - e2[i]
    res = res + (w[i] * sub * sub)
}
res = sqrt(res)
return res
```

A continuación, el algoritmo de comparación *1-NN*:

```
data = container_data
label = container_class
e = current_example
labelRes = first(label)
dMin = de(first(data), e)
for i in size(data)
{
    dAux = de(data[i], e)
    if dAux<dMin
    {
        dMin = dAux
        labelRes = label[i]
    }
}
return labelRes, dMin
```

Descripción en lenguaje natural:

- *data* es el conjunto de ejemplos de los que se compone la muestra.
- *label* es el conjunto de clases (etiquetas) que tiene cada ejemplo de la muestra.
- *e* es el ejemplo actual sobre el que se quiere calcular la distancia euclídea.
- *labelRes* contiene la etiqueta del ejemplo de la muestra cuya distancia es menor con respecto al ejemplo que se evalúa.
- *dMin* es la distancia mínima que se ha encontrado hasta el momento.

El algoritmo *1-NN* consiste en obtener la clase de un ejemplo, ya que, en principio, sería desconocida.

Devuelve una etiqueta o clase comparando con todos los ejemplos que se encuentran en la muestra. Lógicamente, el ejemplo que se quiere comprobar no puede pertenecer a la muestra porque el resultado de calcular la distancia con él mismo, sería 0.

En el caso de querer comprobar la clase de un ejemplo perteneciente a la muestra, habría que aplicar el algoritmo *1-NN leave one out*, que compara con todos los ejemplos menos consigo mismo.

El algoritmo comienza obteniendo la distancia euclídea del ejemplo desconocido con el primer ejemplo de la muestra y guardando, a su vez, la clase a la que pertenece éste.

A continuación, se recorren todos los ejemplos restantes de la muestra, calculando la distancia euclídea de cada ejemplo y comparándola con la mínima que se haya encontrado. Si en algún momento, se encuentra una distancia menor, se actualiza este valor y el de la clase, guardando la que corresponda con dicho ejemplo.

Finalmente, se devuelve la clase mínima encontrada, que será la clase del ejemplo desconocido.

5. Explicación del procedimiento considerado para desarrollar la práctica

El núcleo principal de esta práctica ha sido desarrollado sin usar ningún framework, ni código proporcionado en prácticas o cualquier otro lugar.

El lenguaje de programación utilizado ha sido *C++* en el 100% de la práctica.

La aplicación se compone de cuatro clases principales:

- *Arff*: contiene la estructura de datos y todos los métodos necesarios para leer y realizar operaciones con la muestra.
- *Sorter*: es el clasificador de la aplicación. Hace uso de la solución que se va generando durante toda la práctica, para en base a los datos almacenados en *Arff*, clasificar un ejemplo. Contiene los *algoritmos de generación aleatoria de la solución*, *Relief*, *1-NN*, *distancia euclídea* y *función de evaluación*.
- *LocalSearch*: se encarga de aplicar la *Búsqueda Local* a una solución en concreto y devolverla. Contiene el *algoritmo de mutación* y el de *Búsqueda Local*.
- *GeneticA*: es la clase que aplica los *Algoritmos Genéticos*. Tiene dos *Algoritmos Genéticos* implementados y gestiona, dependiendo de los parámetros recibidos, el tipo de cruce que realizar (*BLX* o *cruce aritmético*) o la aplicación de la *Búsqueda Local* para convertir el *Algoritmo Genético Generacional* en *Algoritmo Memético*. Contiene el *algoritmo de Selección por Torneo Binario*, *cruce BLX*, *cruce aritmético*, *mutación*, *Algoritmo Genético Generacional* y *Algoritmo Genético Estacionario*.

Está gestionada por un archivo principal (*main.cpp*) que se encarga de lanzar todo el proceso y mostrar en pantalla la resolución de los algoritmos, publicando la tasa de acierto y tiempo en segundos que ha necesitado en la ejecución.

Para compilar todo el proyecto y crear el archivo ejecutable que lanzará la aplicación, existe un archivo Makefile.

Además, para la realización de la práctica, se hace uso de las siguientes librerías:

- *OpenMP*: la práctica está implementada haciendo uso de programación paralela a través de *OpenMP*.
- *random.h*: código disponible en la sección de código fuente de la web de la asignatura. Se hace uso de una semilla (*Seed*) que se inicializa con un valor dado por parámetro al ejecutar la aplicación para obtener los números aleatorios necesarios en el desarrollo de los algoritmos.
- <*sys/time.h*>: para medir el tiempo que requiere cada algoritmo.
- <*random*>: para generar el número aleatorio de la distribución normal de media 0 y desviación típica 0.3.
- *ARFF formatted file reader in C++*: para leer los archivos *.arff*. Arff Github

  Ha sido necesario readaptar levemente el código disponible de esta librería para solucionar problemas generados en la lectura de los archivos.

a) Manual de usuario:

Para compilar la aplicación, se hace uso de un archivo *Makefile*.

Se debe descomprimir el archivo *.zip* y ejecutar *make* en la raíz de los directorios generados.

Se habrá creado un archivo ejecutable en la carpeta */bin* llamado *p1*.

En la carpeta *data* se incluyen los archivos *.arff* con los que se quiera ejecutar la aplicación.

Para lanzar la aplicación se debe especificar qué archivo se desea utilizar y el valor de la semilla de inicialización de los valores aleatorios. Así, para la ejecución de esta práctica, se han utilizado los siguientes comandos a través del terminal:

- *./bin/p1 sonar.arff 1824*
- *./bin/p1 wdbc.arff 1824*
- *./bin/p1 spambase-460.arff 1824*

El archivo *Makefile* dispone de limpieza en caso de querer recompilar:

- *make clean*
- *make mr.proper*

6. Experimentos y análisis de resultados

a) Descripción de los casos del problema empleados y de los valores de los parámetros considerados en las ejecuciones de cada algoritmo:

En la realización de esta práctica se han utilizado tres tipos de problemas:

1. *Sonar*: conjunto de datos de detección de materiales mediante señales de sónar, discriminando entre objetos metálicos y rocas. 208 ejemplos con 60 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *sonar.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina el tipo de material del ejemplo se encuentra situada al final de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *false*, indicando que la clase del ejemplo se encuentra al final.

2. *WDBC*: esta base de datos contiene 30 características calculadas a partir de una imagen digitalizada de una punción aspiración con aguja fina (PAAF) de una masa en la mama. Se describen las características de los núcleos de las células presentes en la imagen. La tarea consiste en determinar si un tumor encontrado es benigno o maligno (M = maligno, B = benigno). 569 ejemplos con 30 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *wdbc.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina el tipo de tumor del ejemplo se encuentra situada al principio de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *true*, indicando que la clase del ejemplo se encuentra al principio.

3. *SpamBase*: conjunto de datos de detección de SPAM frente a correo electrónico seguro. 460 ejemplos con 57 características que deben ser clasificados en 2 clases.

   El fichero utilizado para obtener estos datos está nombrado como *spambase-460.arff*. Se utiliza la librería *ARFF formatted file reader in C++*. La clase que determina si el ejemplo es spam o no se encuentra situada al final de todos los atributos. Al cargar el archivo en la clase *Arff* se debe indicar como primer parámetro el nombre del fichero y como segundo parámetro *false*, indicando que la clase del ejemplo se encuentra al final.

La semilla de inicialización para la ejecución de esta prueba es *1824* para todos los problemas y se especifica al ejecutar la aplicación.

Los ficheros de datos que leerá la aplicación se encuentran en el directorio *data*, aunque también se podrán encontrar disponibles en el directorio *bin*, tal y como se indica en el guion.

b) Resultados obtenidos según el formato especificado:

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	85,44	0,01140300	93,66	0,01548600	84,72	0,01543700
Partición 1-2	80,95	0,00047900	91,93	0,00239600	80,95	0,00198800
Partición 2-1	84,47	0,00051200	92,63	0,00253600	81,39	0,00221400
Partición 2-2	83,81	0,00047800	95,42	0,00218400	84,28	0,00237800
Partición 3-1	78,64	0,00047100	92,25	0,00582400	88,21	0,00199100
Partición 3-2	80,95	0,00047300	94,39	0,00235100	83,55	0,00201200
Partición 4-1	82,86	0,00286200	96,49	0,00226100	82,68	0,00216600
Partición 4-2	79,61	0,00045200	95,42	0,00228500	88,21	0,00203200
Partición 5-1	84,47	0,00047700	93,31	0,00240100	84,85	0,00217700
Partición 5-2	72,38	0,00047800	92,98	0,00235900	85,15	0,00205100
Media	81,36	0,00180850	93,85	0,00400830	84,40	0,00344460

Tabla 6.b.1: Resultados obtenidos por el algoritmo 1-NN en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	81,55	0,00058900	92,96	0,00246800	81,66	0,00216400
Partición 1-2	81,90	0,00061500	93,33	0,00244800	70,56	0,00215000
Partición 2-1	81,55	0,00054900	92,98	0,00239600	81,39	0,00213300
Partición 2-2	85,71	0,00059800	95,42	0,00237000	78,17	0,00217100
Partición 3-1	72,82	0,00055700	94,01	0,00235000	73,36	0,00220900
Partición 3-2	81,90	0,00056600	93,68	0,00246700	79,65	0,00222200
Partición 4-1	80,00	0,00060300	96,84	0,00241700	83,55	0,00219400
Partición 4-2	80,58	0,00055500	96,13	0,00234500	66,81	0,00217400
Partición 5-1	87,38	0,00058400	93,31	0,00231600	65,80	0,00224600
Partición 5-2	76,19	0,00057300	92,98	0,00254000	84,28	0,00227000
Media	80,96	0,00057890	94,16	0,00241170	76,52	0,00219330

Tabla 6.b.2: Resultados obtenidos por el algoritmo Relief en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	85,44	0,65165100	93,66	1,48910000	84,72	2,59528000
Partición 1-2	82,86	0,96643200	91,93	1,46936000	82,68	3,54365000
Partición 2-1	86,41	0,63219400	92,63	1,38300000	83,98	3,68522000
Partición 2-2	84,76	0,62137100	96,48	1,54793000	86,03	3,25619000
Partición 3-1	78,64	0,58809300	92,61	1,47303000	89,96	4,95662000
Partición 3-2	82,86	0,61640800	94,74	1,46810000	85,28	4,05407000
Partición 4-1	84,76	0,75861100	97,54	1,48074000	84,42	4,93025000
Partición 4-2	81,55	0,70530200	96,48	1,70167000	89,52	2,69061000
Partición 5-1	84,47	0,57916400	93,66	2,04521000	86,58	3,37432000
Partición 5-2	72,38	0,60365100	95,09	2,86875000	87,34	2,87519000
Media	82,41	0,67228800	94,48	1,69268900	86,05	3,59614000

Tabla 6.b.3: Resultados obtenidos por el algoritmo BL en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	94,17	7,91373000	97,18	29,99550000	90,39	27,27780000
Partición 1-2	90,48	7,70449000	96,84	28,39900000	91,34	26,90310000
Partición 2-1	94,17	7,96976000	96,49	27,25190000	92,21	26,31580000
Partición 2-2	94,29	7,52658000	97,18	29,19900000	91,70	27,20200000
Partición 3-1	87,38	7,57513000	95,77	27,38290000	94,76	27,15760000
Partición 3-2	93,33	7,91154000	98,95	27,58480000	90,91	26,93720000
Partición 4-1	90,48	7,53045000	98,60	27,45800000	93,07	26,50400000
Partición 4-2	89,32	7,38967000	97,54	28,50760000	93,89	26,80520000
Partición 5-1	95,15	7,35036000	96,48	27,31360000	94,37	26,73790000
Partición 5-2	85,71	7,60992000	97,19	27,36650000	92,14	27,97470000
Media	91,45	7,64816300	97,22	28,04588000	92,48	26,98153000

Tabla 6.b.4: Resultados obtenidos por el algoritmo GG-BLX en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	91,26	18,20190000	95,42	67,12850000	92,14	33,05010000
Partición 1-2	88,57	16,50680000	96,49	74,83650000	87,88	39,81980000
Partición 2-1	91,26	18,25420000	96,49	31,73060000	91,34	38,17500000
Partición 2-2	92,38	21,48260000	97,18	65,13700000	89,08	46,90120000
Partición 3-1	84,47	18,64560000	95,07	59,38920000	91,27	36,04250000
Partición 3-2	90,48	18,23450000	98,60	64,04080000	90,04	37,09100000
Partición 4-1	87,62	17,26970000	97,89	71,77570000	91,34	44,94730000
Partición 4-2	87,38	15,83870000	97,18	61,62690000	91,70	40,79650000
Partición 5-1	93,20	15,45250000	96,48	71,12710000	90,48	39,24050000
Partición 5-2	83,81	17,82890000	95,79	70,64120000	91,70	42,94570000
Media	89,04	17,77154000	96,66	63,74335000	90,70	39,90096000

Tabla 6.b.5: Resultados obtenidos por el algoritmo GG-CA en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	96,12	21,13870000	97,54	52,52450000	91,27	56,33090000
Partición 1-2	92,38	21,53840000	97,19	51,95300000	92,64	56,39350000
Partición 2-1	96,12	21,15500000	95,79	46,49420000	93,94	56,26880000
Partición 2-2	93,33	21,08550000	97,89	51,51020000	93,89	56,85610000
Partición 3-1	89,32	20,73190000	95,77	50,83990000	94,76	56,24570000
Partición 3-2	92,38	21,55920000	98,25	51,27870000	93,07	58,02650000
Partición 4-1	91,43	21,70090000	98,60	52,22910000	94,81	55,95600000
Partición 4-2	92,23	20,88440000	97,89	51,17880000	94,76	58,10110000
Partición 5-1	96,12	20,96540000	97,18	51,07450000	93,94	56,44890000
Partición 5-2	87,62	21,42910000	97,89	51,70070000	93,45	56,54120000
Media	92,71	21,21885000	97,40	51,07836000	93,65	56,71687000

6.b.6: Resultados obtenidos por el algoritmo GE-BLX en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	92,23	32,70340000	95,77	62,21810000	90,83	74,27670000
Partición 1-2	88,57	32,84510000	96,84	59,37830000	91,34	75,19990000
Partición 2-1	95,15	35,21370000	95,44	60,60420000	93,94	74,38010000
Partición 2-2	93,33	32,70070000	97,18	62,13510000	91,27	74,20030000
Partición 3-1	89,32	32,41570000	96,48	53,85170000	93,45	73,80790000
Partición 3-2	90,48	32,59110000	97,19	58,63720000	92,64	76,21580000
Partición 4-1	90,48	32,55340000	98,25	59,53820000	94,37	73,90450000
Partición 4-2	92,23	32,68640000	97,18	62,70080000	94,32	74,80090000
Partición 5-1	94,17	32,34920000	96,83	61,31400000	93,07	74,41550000
Partición 5-2	86,67	37,10570000	96,49	62,73270000	91,70	74,91890000
Media	91,26	33,31644000	96,77	60,31103000	92,69	74,61205000

Tabla 6.b.7: Resultados obtenidos por el algoritmo GE-CA en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	83,50	7,51390000	94,37	33,51800000	87,77	32,40780000
Partición 1-2	88,57	7,53474000	95,79	35,57280000	89,18	31,86240000
Partición 2-1	88,35	7,69074000	95,09	34,65470000	87,45	31,00620000
Partición 2-2	88,57	7,37878000	97,54	35,91560000	87,88	33,91970000
Partición 3-1	85,44	7,79888000	94,37	32,91030000	90,83	33,10850000
Partición 3-2	87,62	7,37500000	96,49	35,91930000	88,31	34,68110000
Partición 4-1	86,67	8,23047000	97,19	34,66620000	87,88	33,69370000
Partición 4-2	83,50	7,78165000	95,42	35,47190000	90,83	33,73500000
Partición 5-1	93,20	7,77977000	96,48	34,17380000	89,18	31,44480000
Partición 5-2	85,44	9,04013000	94,39	36,77700000	86,90	33,77680000
Media	87,09	7,81240600	95,71	34,95796000	88,62	32,96360000

Tabla 6.b.8: Resultados obtenidos por el algoritmo AM-(10,1,0) en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	92,23	7,82320000	94,72	18,47590000	91,27	27,72540000
Partición 1-2	89,52	8,10058000	95,77	21,06170000	87,88	30,89970000
Partición 2-1	93,20	7,94910000	95,09	19,61210000	90,91	30,04270000
Partición 2-2	92,38	7,84888000	96,48	20,50500000	89,52	32,75060000
Partición 3-1	85,44	7,75195000	95,42	17,65930000	90,39	25,53110000
Partición 3-2	88,57	8,20480000	96,84	18,92950000	89,52	26,31190000
Partición 4-1	88,57	8,30194000	97,54	17,92940000	92,21	30,22140000
Partición 4-2	89,32	8,04152000	98,25	17,71450000	92,58	30,77120000
Partición 5-1	89,32	7,79787000	95,42	17,23460000	92,21	26,59920000
Partición 5-2	86,41	7,89790000	96,14	19,15850000	90,83	30,94550000
Media	89,50	7,97177400	96,17	18,82805000	90,73	29,17987000

Tabla 6.b.9: Resultados obtenidos por el algoritmo AM-(10,0,1) en el problema del APC

	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
Partición 1-1	90,29	7,93323000	94,72	17,28310000	90,83	30,53470000
Partición 1-2	79,61	8,00408000	94,01	18,14730000	91,34	30,92620000
Partición 2-1	91,26	8,10872000	94,39	18,22670000	90,48	29,81900000
Partición 2-2	92,38	7,89792000	91,58	19,62710000	84,42	30,45240000
Partición 3-1	89,32	7,76380000	94,72	17,62080000	90,83	30,50960000
Partición 3-2	90,48	7,91734000	95,42	19,30340000	86,03	30,71570000
Partición 4-1	91,43	7,95369000	97,54	19,24900000	89,61	26,08450000
Partición 4-2	88,35	7,98397000	96,84	17,77870000	83,12	31,04130000
Partición 5-1	94,17	7,79880000	96,13	16,69250000	90,48	28,12560000
Partición 5-2	83,81	6,29914000	92,96	20,04070000	85,28	30,89380000
Media	89,11	7,76606900	94,83	18,39693000	88,24	29,91028000

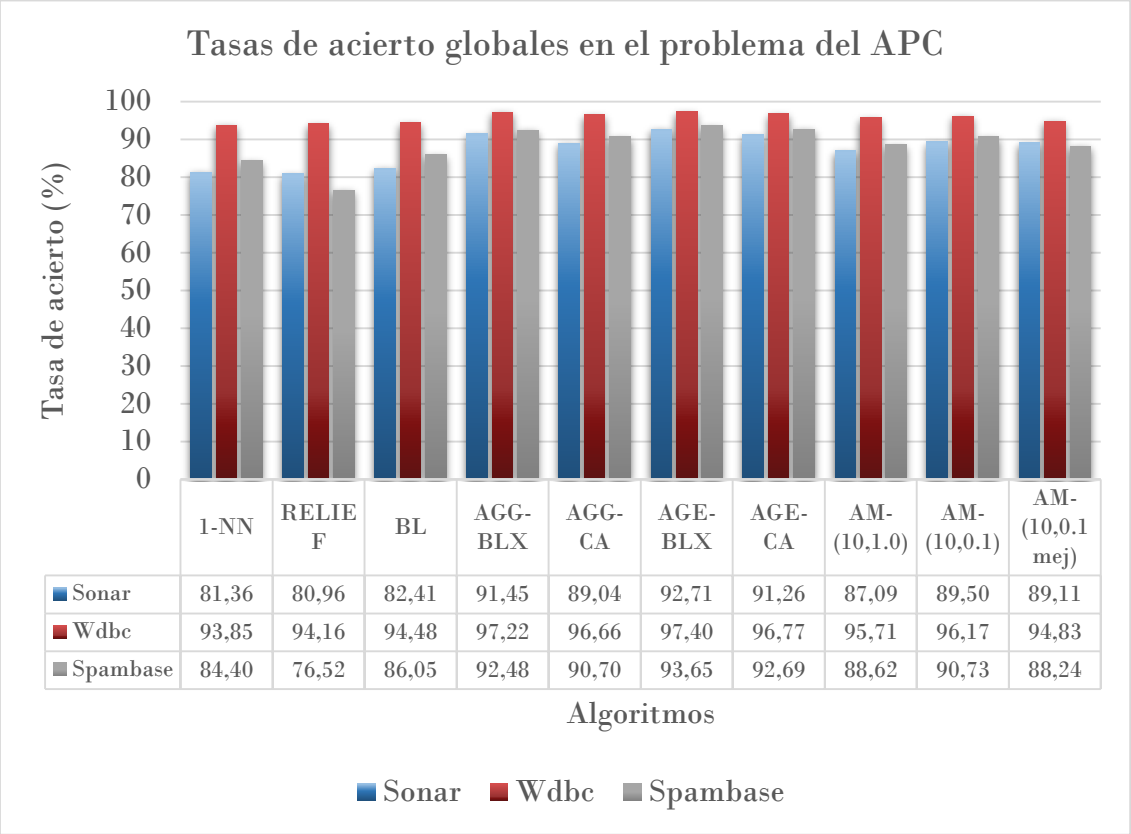
Tabla 6.b.10: Resultados obtenidos por el algoritmo AM-(10,0,1mej) en el problema del APC

Algoritmo	Sonar		Wdbc		Spambase	
	%_clas	T	%_clas	T	%_clas	T
1-NN	81,36	0,00180850	93,85	0,00400830	84,40	0,00344460
RELIEF	80,96	0,00057890	94,16	0,00241170	76,52	0,00219330
BL	82,41	0,67228800	94,48	1,69268900	86,05	3,59614000
AGG-BLX	91,45	7,64816300	97,22	28,04588000	92,48	26,98153000
AGG-CA	89,04	17,77154000	96,66	63,74335000	90,70	39,90096000
AGE-BLX	92,71	21,21885000	97,40	51,07836000	93,65	56,71687000
AGE-CA	91,26	33,31644000	96,77	60,31103000	92,69	74,61205000
AM-(10,1.0)	87,09	7,81240600	95,71	34,95796000	88,62	32,96360000
AM-(10,0.1)	89,50	7,97177400	96,17	18,82805000	90,73	29,17987000
AM-(10,0.1mej)	89,11	7,76606900	94,83	18,39693000	88,24	29,91028000

Tabla 6.b.11: Resultados globales en el problema del APC

c) Análisis de resultados:

- Tasas de acierto globales en el problema del APC:



	1-NN	RELIEF	BL	AGG-BLX	AGG-CA	AGE-BLX	AGE-CA	AM-(10,1.0)	AM-(10,0.1)	AM-(10,0.1 mej)
Sonar	81,36	80,96	82,41	91,45	89,04	92,71	91,26	87,09	89,50	89,11
Wdbc	93,85	94,16	94,48	97,22	96,66	97,40	96,77	95,71	96,17	94,83
Spambase	84,40	76,52	86,05	92,48	90,70	93,65	92,69	88,62	90,73	88,24

Como dato más interesante, se puede decir que el algoritmo que mejor resultados ofrece es el Algoritmo Genético Estacionario con cruce BLX.

Este hecho se debe a que, en cada nueva generación, siempre se va a tener la mayor parte de cromosomas de la generación actual, más dos nuevos cromosomas obtenidos del cruce de dos padres (en caso de que mejoren a los peores de la generación actual). La nueva generación nunca será peor que la generación actual, por lo que la tasa de acierto media siempre será igual o superior.

Con el Algoritmo Genético Generacional, la nueva generación que se crea, puede ser completamente peor que la actual. Conservar el elitismo de la población, añadiendo a la nueva generación el mejor cromosoma de la actual si supera al peor, permite que poco a poco los peores cromosomas vayan desapareciendo y gracias al cruce, sean mejores. Pese a esto, es un proceso al que le cuesta más conseguir mejores cromosomas, como así lo confirman los resultados obtenidos.

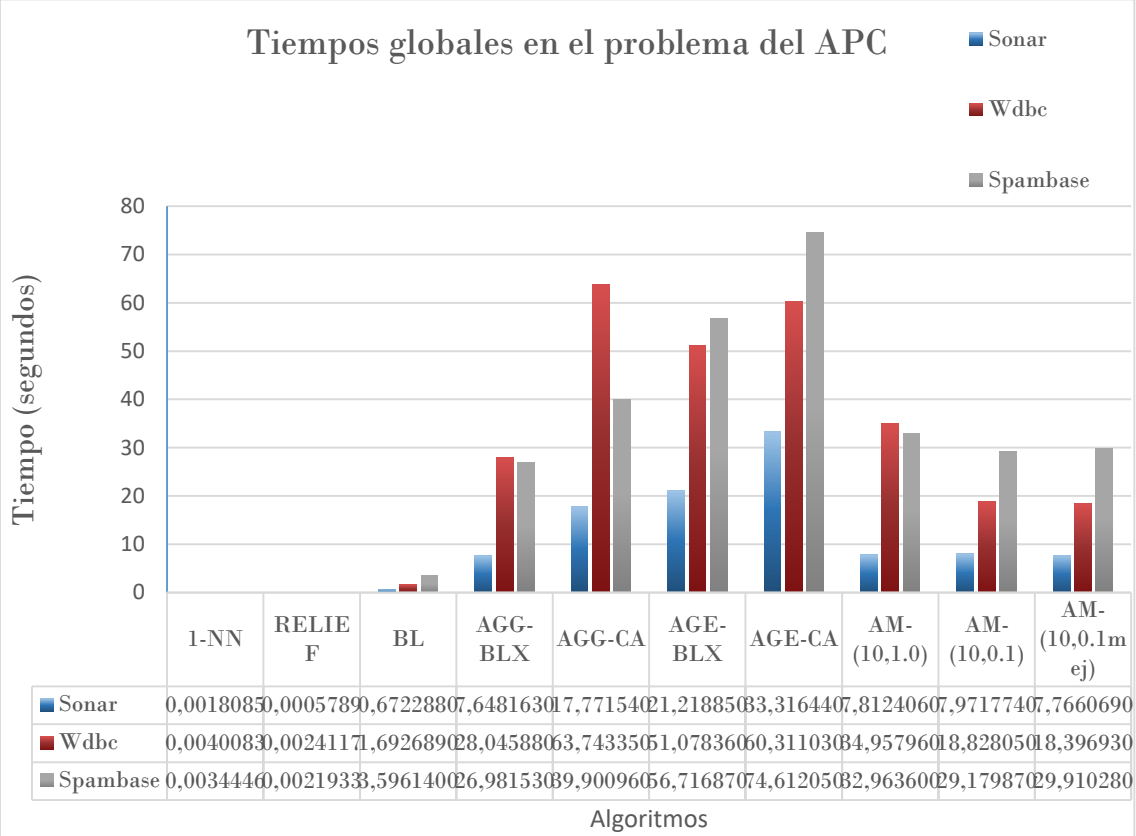
Con respecto a los tipos de cruce empleados, el cruce que mejor tasa de acierto consigue es el cruce BLX. Con este cruce se consigue una rápida convergencia hacia los cromosomas que mejor resultado ofrecen. En cambio, con el cruce aritmético, se penaliza más que con el cruce BLX si dos padres son muy dispares, con lo que le cuesta más conseguir cromosomas de calidad.

Otro dato interesante es la tasa de acierto conseguida con la solución aleatoria y el Algoritmo Relief. Los dos tipos de soluciones son igual de malos. Pese a que se consigue una alta tasa de acierto en los datos WDBC, no es debido a la calidad de los algoritmos, sino a que es una muestra muy sencilla. Teniendo en cuenta estos datos, no merece la pena implementar el Algoritmo Relief ni como solución inicial.

La Búsqueda Local está basada en el algoritmo aleatorio y mejora aproximadamente un 1% la tasa de acierto. En principio, es buena idea implementarla.

Con esta mejora de la Búsqueda Local, se podría pensar que añadirla a los Algoritmos Genéticos sería una buena opción. La realidad no es así. El equilibrio entre exploración y explotación no se ha encontrado. Principalmente, se debe a la baja tasa de acierto que mejora la Búsqueda Local y al número de evaluaciones que se pierden y, por tanto, capacidad para crear nuevas generaciones entre los Algoritmos Meméticos. Para estas muestras en concreto, interesa más explorar que explotar.

- Tiempos globales en el problema del APC:



	1-NN	RELIEF	BL	AGG-BLX	AGG-CA	AGE-BLX	AGE-CA	AM-(10,1.0)	AM-(10,0.1)	AM-(10,0.1mej)
Sonar	0,0018085	0,0005789	0,6722880	7,6481630	17,771540	21,218850	33,316440	7,8124060	7,9717740	7,7660690
Wdbc	0,0040083	0,0024117	1,6926890	28,045880	63,743350	51,078360	60,311030	34,957960	18,828050	18,396930
Spambase	0,0034446	0,0021933	3,5961400	26,981530	39,900960	56,716870	74,612050	32,963600	29,179870	29,910280

En estos datos se puede apreciar como el problema Sonar es el más sencillo de todos y, debido a esto, el más rápido en ejecutarse. WDBC se encuentra prácticamente en el mismo nivel de complejidad que SpamBase, aunque quizás, este último sea el más complejo y laborioso en cuanto a computación.

El algoritmo 1-NN y Relief apenas tienen carga de cómputo ya que utilizan operaciones muy sencillas en sus cálculos.

A partir de la Búsqueda Local, se puede apreciar un incremento del tiempo de cálculo. Al tener que realizar 15.000 evaluaciones de los cromosomas, junto a las mutaciones, hace que el tiempo avance hasta 0.6 segundos en el Sonar, 1.7 segundos WDBC y 3.5 segundos en SpamBase.

El tiempo total que ha tardado en ejecutarse completamente la aplicación, ha sido 1 hora y 52 minutos. Es un tiempo que opino que es bastante bueno, pese a que se ha visto perjudicado por los Algoritmos Genéticos Estacionarios y principalmente por el cruce aritmético.

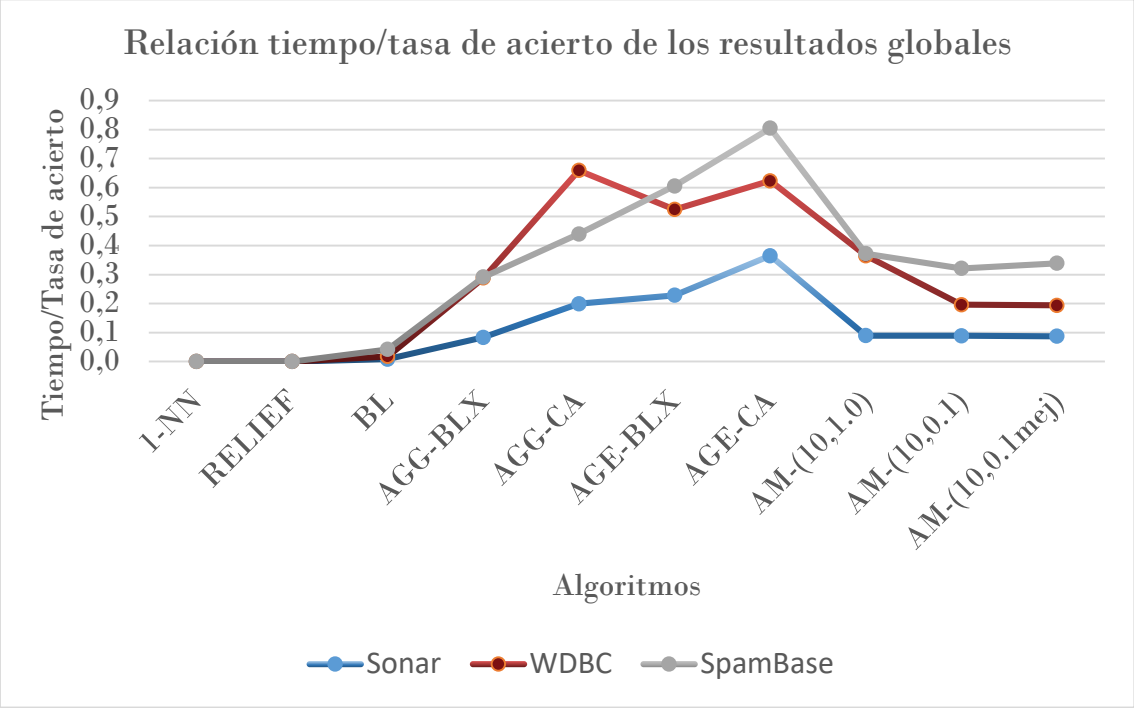
Se ha conseguido realizar la Búsqueda Local y el Algoritmo Genético Generacional con cruce BLX en tiempos muy bajos, lo que sumado a la tasa de acierto que ofrece el Algoritmo Genético Generacional con cruce BLX, hace de éste la mejor opción a tener en cuenta para implementar en la aplicación final.

Debido a una segunda versión implementada en el cruce aritmético que no está correctamente optimizada y, sumado a que, para cada nueva generación hay que calcular el doble de padres para realizar los cruces, los Algoritmos Genéticos que hacen uso de este cruce, se ejecutan en tiempos realmente malos.

Los Algoritmos Genéticos Estacionarios, pese a obtener las mejores tasas de acierto, penalizan mucho en que se crean 7.500 generaciones, 7.000 más que en los Algoritmos Genéticos Generacionales. Debido a este hecho, los Generacionales tienen la mejor relación tiempo/tasa de acierto.

Los Algoritmos Meméticos se ejecutan en un buen plazo de tiempo, tanto como los Algoritmos Genéticos Generaciones, gracias a que se basan en éstos y en la Búsqueda Local (dos algoritmos bien optimizados) pero, como su tasa de acierto es inferior, no se elegirían como una solución al problema.

- Relación Tiempo/Tasa de acierto:



Para comprobar qué algoritmos ofrecen mejor relación tiempo/tasa de acierto, se puede consultar en la gráfica anterior.

A medida que el algoritmo se acerque a 1, la relación será peor. Esto nos indica que el algoritmo 1-NN y Relief obtienen la mejor relación.

Es tarea del cliente decidir qué algoritmo implementar, valorando la tasa mínima de acierto que está dispuesto a aceptar o el tiempo máximo de que dispone.

Teniendo en cuenta la tasa de acierto, se puede apreciar como el Algoritmo Genético Generacional con cruce BLX es la mejor decisión a adoptar, siendo el Algoritmo Genético Estacionario con cruce aritmético el peor.

7. Referencias bibliográficas

La información principal para realizar esta práctica ha sido consultada en la web de la asignatura:

- Tema 2. Modelos de Búsqueda: Entornos y Trayectorias vs. Poblaciones.
- Seminario 2. Problemas de optimización con técnicas basadas en trayectorias simples: Búsqueda Local.
- Seminario 3. Problemas de optimización con técnicas basadas en poblaciones.
- Guion B: APC
- Generador de números pseudoaletarios
- Tablas e instancias para el problema del Aprendizaje de Pesos en Características (APC)

La biblioteca para leer los ficheros Arff se encuentra en:

- https://github.com/teju85/ARFF

Otras fuentes consultadas:

- http://www.cplusplus.com/reference/vector/vector/vector/
- http://www.cplusplus.com/reference/algorithm/sort/
- https://totaki.com/poesiabinaria/2014/01/concurrencia-posix-threads-y-variables-compartidas-en-c/
- http://stackoverflow.com/questions/10874214/measure-execution-time-in-c-openmp-code
- http://stackoverflow.com/questions/496448/how-to-correctly-use-the-extern-keyword-in-c
- http://stackoverflow.com/questions/4940259/lambdas-require-capturing-this-to-call-static-member-function
- http://stackoverflow.com/questions/9712413/c-stack-smashing-detected
- https://es.wikipedia.org/wiki/Funci%C3%B3n_de_densidad_de_probabilidad
- http://www.cplusplus.com/reference/random/normal_distribution/
- http://ieeexplore.ieee.org/abstract/document/996017/