Metaheurísticas (2016-2017)
Grado en Ingeniería Informática
Universidad de Granada

Practica 1: APC

Samuel Cardenete Rodríguez.
Correo: samuelcr1995@correo.ugr.es
DNI: 75934968P

17 de abril de 2017

Índice

Índice de figuras

1. Formulación del problema. APC

1.1. Introducción

El problema a abordar en esta práctica consiste en el aprendizaje de pesos por características (APC).
Se trata de un problema de búsqueda con codificación real en el espacio n-dimensional, siendo n el número de características.

Estamos delante de un problema de Machine Learning, mediante el cuál dado un conjunto de datos no etiquetados obtengamos una etiqueta para cada uno de los datos (por ejemplo, determinar a partir de determinados parámetros si se tiene o no cáncer) bajo una tasa de acierto, siempre a maximizar.

1.2. Objetivos

Como objetivo principal del problema, ajustaremos un conjunto de ponderaciones o pesos asociados al conjunto total de características, utilizando para ello un conjunto de datos de entrenamiento (ya etiquetados), de tal forma que los clasificadores que se construyan a partir de él sean mejores.

APC asigna valores reales a las características, de tal forma que se describe o pondera la relevancia de cada una de ellas al problema del aprendizaje.
Para la resolución del problema, nos basaremos en optimizar el rendimiento de un clasificador basado en vecinos cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos. En nuestro caso, el clasificador considerado será el 1-NN (vecino más cercano). Para ello adaptaremos las dos siguientes técnicas metaheurísticas:

- Algoritmos Genéticos: Dos variantes generacionales elitistas (AGGs) y otras dos estacionarias (AGEs). Aparte del esquema de evolución, la única diferencia entre los dos modelos de AGGs y AGE será el operador de cruce empleado.
- Algoritmos Meméticos: Tres variantes de algoritmos meméticos (AMs) basadas en un AGG. La única diferencia entre las tres variantes de AMs serán los parámetros considerados para definir la aplicación de la búsqueda local.

Tras esto realizaremos un estudio comparativo de los resultados obtenidos por las distintas técnicas metaheurísticas.

2. Algoritmos empleados al problema.

2.1. Algoritmos comunes

Representación y explicación de los principales algoritmos comunes (operadores cruce, torneo binario...) así como su estructura en pseudocódigo.

2.2. Generados soluciones aleatorias

En el ámbito de nuestro problema APC, la representación de una solución corresponde a un vector de pesos W. Para determinados algoritmos como los genéticos o la búsqueda local partimos de una solución aleatoria, es decir, un vector de pesos con componentes aleatorias generadas entre un intervalo $[0, 1]$ (de esta forma ahorramos la normalización). El pseudocódigo sería el siguiente:

```
Desde i hasta numero atributos
        SolucionInicialGenerarAleatorio ⇐ Aleatorio (0,1)
Fin Desde

Devuelve SolucionInicial
```

De esta forma obtenemos un cromosoma aleatorio ya normalizado.

2.3. Normalización de datos.

Dados unos conjuntos de datos a utilizar como base de pruebas para nuestro problema (wdbc, sonar...), para no priorizar unos datos sobre otros es necesario la normalización de los datos en un intervalo $[0, 1]$. Para ello, empleo un script en R, de forma que leemos los ficheros .arff, y utilizamos la función normalizarDatos para guardarlos en un .csv ya normalizados, de forma que dado un valor x_j perteneciente a un atributo j del ejemplo x, y sabiendo que el dominio del atributo j es $[Min_j, Max_j]$, el valor normalizado de x_j es:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j}$$

2.4. Calcular distancia

El clasificador empleado cuyo rendimiento pretendemos optimizar emplea la distancia euclídea entre dos características para el cálculo del vecino más cercano.
El pseudocódigo de la función que calcula la distancia entre dos características sería el siguiente:

```
//Calcular la distancia entre dos cromosomas
// teniendo en cuenta el vector de pesos:
Calcular distancia cromosoma A y B:
        Para cada i ∈ Atributos(A) y j ∈ Atributos(B)
                distancia = distancia + √(Pesos_i * (i − j)^2)
        Fin para cada
```

```
        Devolver distancia
```

De esta forma podemos utilizar dicha distancia para compararlas entre atributos y ver cual se encuentra más cerca del atributo actual.

2.5. Función de evaluación

Implementación de la función de evaluación 1-NN explicada en la descripción anteriormente.
El pseudocódigo es el siguiente:

```
1-NN(train, test, pesos)

//Recorremos el conjunto test:
Para todo a_i ∈ Test
        //inicializamos las distancias minimas al enemigo
        // y al amigo a infinito
        d_min = ∞


        //Ahora recorremos el conjunto train
        // calculando las distancias, buscando el min

        Para todo b_i ∈ Train
                Si a_i ≠ b_i //leave one out
                        Si (calculaDistancia(a_i, b_i) < d_min^enemigo)
                                d_min = calculaDistancia(a_i, b_i, pesos)

                        Fin Si
        Fin para todo

        //Ahora calculamos la tasa:



Fin para todo

tasa = 100*etiquetas bien clasificadas / numero total de etiquetas

Devolver tasa
```

2.6. Generación de vecinos

En el esquema de generación de vecinos, necesario tanto para las mutaciones en los algoritmos genéticos, como para la generacion de vecindario en la búsqueda local, emplea un movimiento de cambio por mutación Normal, Mov(W,σ), deforma que a una característica del atributo se le suma un valor aleatorio obtenido apartir de una distribución normal de media 0 y desviación típica 0.3.

El pseudocódigo del algoritmo quedaría tal que así:

```
Generar vecino (Pesos, a_i = atributo a modificar)
        //Obtenemos un valor de la distribucion normal mencionada:
        aleatorio = Distribucion normal(media = 0, σ = 0,3)
        //modificamos el atributo a_i de los pesos:
        Pesos(a_i) = Pesos(a_i) + aleatorio

        //truncamos si es necesario para tener los valores normalizados
        //entre [0,1]:

        Si Pesos(a_i) < 0
                Pesos(a_i) = 0
        Si Pesos(a_i) > 1
                Pesos(a_i) = 1

        Devolvemos Pesos
```

De esta forma tendremos un vecino generado.

2.7. Operador cruce BLX

Para el algoritmo genético BLX es necesario la implementación de dicho cruce BLX-α, donde en nuestro caso $\alpha = 0{,}3$. Dados dos cromosomas p_1, p_2 pertenecientes a los padres, generamos dos descendientes h_1, h_2 de la siguiente forma descrita en pseudocódigo:

```
CruceBLX entre p_1 y p_2
        //Obtenemos los atributos maximos y minimos para cada padre:
        a^1_max, a^1_min ∈ p_1
        a^2_max, a^2_min ∈ p_2

        //Ahora obtenemos el maximo y el minimo de los dos padres:
        Max(a^1_max, a^2_max)
        Min(a^1_min, a^2_min)

        //Ahora generamos el intervalo I del cual generaremos
            aleatoriamente a los dos hijos:
        I = [Min(a^1_min, a^2_min) - (Max(a^1_max, a^2_max - Min(a^1_min, a^2_min) *0.3, Max(a^1_max, a^2_max) + (
            Max(a^1_max, a^2_max - Min(a^1_min, a^2_min) *0.3 ]

        //Creamos a los dos hijos h_1, h_2
        Para cada atributo a^i_1 ∈ h_1 y a^i_2 ∈ h_2
                a^i_1 = Aleatorio (I)
                a^i_2 = Aleatorio (I)
        fin para cada

        devolvemos a_1 y a_2
```

De esta forma para cada pareja de padres, obtenemos una pareja de hijos.

2.8. Operador cruce aritmético

Además del operador BLX, es necesario implementar el operador de cruce aritmético. A diferencia del anterior, para cada pareja de padres obtenemos un único descendiente (h_1), que posee la media aritmética de los genes de ambos padres (p_1 y p_2).

```
CruceAritmetico entre p1 y p2

        Para cada atributo a_i ∈ h1
                a_i = MediaAritmetica(a^i_p1 y a^i_p2)
        fin para cada

        devolvemos h1
```

2.9. Torneo Binario

La función torneo binario realiza un torneo entre dos padres (cromosomas) aleatorios, es decir, de entre esos dos padres, elige aquel que posea una mayor tasa de clasificación, o lo que es lo mismo, el mejor de los dos.
Genera tantos padres por torneo binario como se le indique. El pseudocódigo sería el siguiente, recibiendo una poblacion y sus tasas correspondientes:

```
TorneoBinario Poblacion, Tasas
        Desde i hasta N_padresagenerar
                //Obtenemos dos padres de forma aleatoria:
                p1 = aleatorio(Poblacion)
                p2 = aleatorio(Poblacion) distinto de p1

                //Ahora nos quedamos con el que tenga mejor tasa de los
                    dos:
                mejores padres ⇐ Max(Tasas(p1), Tasas(p2))
        Fin desde

        Devolver mejores padres
```

De esta forma obtenemos al final un número de padres deseados obtenidos mediante torneo binario, que emplearemos para un posterior cruce en los algoritmos genéticos y meméticos correspondientes.

3. Estructuras de los principales métodos de búsqueda

3.1. Búsqueda Local

La implementación de la búsqueda local consiste en emplear una técnica de búsqueda mediante explotación aplicada a nuestro problema, de forma que nos permita explotar una solución localmente, permitiendo así maximizar la tasa evaluación de una solución de forma local.

Como **criterio de parada** cuando se ejecute como algoritmo individual, se detendrá tras haber generado 20* (numero veces el número de características) vecinos, o cuando se hayan realizado 15000 evaluaciones (llamadas al 1-NN)
El pseudocódigo sería el siguiente:

```
Busqueda Local (Conjunto de datos: Data)
        //Primero generamos una solucion inicial de forma aleatoria
        S_inicial = GenerarSolucionAleatoria()
        //Y calculamos la tasa
        T_inicial = knn(Data, Data, S_inicial)


        //Inicializamos la primera como mejor solucion actual:
        S_mejor = S_inicial
        //Y guardamos la mejor tasa como esta:
        T_mejor = T_inicial

        //Llevamos cuenta del numero de evaluaciones que llevamos:
        numero evaluaciones = 1


        //Empezamos a generar vecindario con la solucion actual
        // hasta que se cumplan los criterios de parada:

        Mientras (numero evaluaciones < 15000) y (vecinos generados <
            20*(Numero genes cromosoma)

                //vamos generando el vecindario:
                Si hay_que_generar_nuevo_vecindario
                        //rellenamos una cola de indices aleatorios desde
                             con los genes a mutar
                        vecinos por generar = Shuffle(Indices de genes)
                        hay_que_generar_nuevo_vecindario = false
                Sino
                        //Generamos un vecino correspondiente sacando
                        // un elemento de la cola anterior para
                        //Indicar que gen mutar:
                        nuevo vecino = generarVecino(S_mejor, vecinos por
                            generar)
                        vecinos generados +1

                        Si (Tasa(nuevo vecino) > T_mejor)
                                S_mejor = nuevo vecino
                                T_mejor = knn(Data, Data, nuevo vecino)

                                numero evaluaciones+1

                                //Indicamos que genere un nuevo
                                    vecindario
                                hay_que_generar_nuevo_vecindario = true
                        Fin Si

                Fin Sino

                //Hemos recorrido un vecindario sin mejora
                Si estaVacia(vecinos por generar)
                        hay_que_generar_nuevo_vecindario = true
                Fin Si
```

```
        Fin Mientras

        Devolver S_mejor
```

3.2. Algortimos genéticos

3.2.1. AGG BLX

El AGG BLX se trata de un algoritmo que implementa un esquema de evolución generacional elitista, mediante un esquema de reemplazamiento generacional, es decir, la nueva generación reemplaza en cada iteración completamente a la generación actual.
Como operador de selección se implementará el torneo binario ya descrito anteriormente, para la elección de los mejores padres a cruzar.

El operador de cruce es el BLX, descrito anteriormente, para generar en cada cruce a partir de dos padres dos descendientes. La probabilidad de cruce será de 0.7.

Para la mutación, se empleará la función generarVecino descrita anteriormente, bajo una probabilidad de mutación en población de un 0.001.

Por último el criterio de parada del algoritmo es no superar las 15000 evaluaciones de la función objetivo (1-NN).

```
AGG BLX (Data)
        //Creamos una poblacion de 30 individuos aleatoria:
        Desde i = 0 hasta i < 30
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        Para cada P_i ∈ poblacion actual
                tasas actuales ⇐ knn(Data, P_i)
        Fin para cada

        //empezamos generacion tras generacion hasta cumplir
        //El criterio de parada:
        Mientras (num. evaluaciones < 15000)
                //Calculamos el mejor padre de la poblacion actual:
                Padre_mejor = Max(tasas poblacion actual)

                //Obtenemos los padres (parejas) a cruzar,
                //con una probabilidad de cruce del 70%
                parejas a cruzar = 0.7 * 30/2

                //Llamamos al torneo binario, obteniendo los padres
                    deseados
                padres a cruzar = torneoBinario(poblacion actual)
                //Realizamos los cruces, obteniendo dos hijos por cruce
                Desde i = 0 hasta i < parejas a cruzar
                        nueva poblacion ⇐ cruceBLX(pareja_i, pareja_{i+1})
```

```
                Fin Desde

                //Aniadimos los padres que no se han cruzado hasta llegar
                // a una nueva poblacion de 30:
                Mientras Tam(nueva poblacion) < 30
                        nueva poblacion ⇐ padres no se han cruzado
                Fin Mientras

                //Calculamos el numero de individuos a mutar, con
                // una probabilidad de 0.001:
                individuos a mutar = 0.001 * 30

                //Mutamos de forma aleatoria los n individuos a mutar:
                Desde i = 0 hasta i < individuos a mutar
                        nueva poblacion_aleatorio = generarVecino (nueva
                            poblacion_aleatorio)
                Fin Desde

                //Calculamos las tasas de la nueva poblacion
                Para cada P_i ∈ nueva poblacion
                        nuevas tasas ⇐ knn(Data, P_i)
                Fin para cada

                //Si hemos perdido al mejor padre lo cambiamos por
                //el peor de la poblacion.
                Si (hemos perdido al mejor padre)
                        Min(tasas actual) = mejor padre
                Fin Si

                //La nueva poblacion es ahora la actual:
                poblacion actual = nueva poblacion
                tasas actual = nuevas tasas


        Fin Mientras

        Devolver mejor individuo poblacion actual
```

3.2.2. AGG CA

El AGG CA se trata de un algoritmo que implementa un esquema de evolución generacional elitista, mediante un esquema de reemplazamiento generacional, es decir, la nueva generación reemplaza en cada iteración completamente a la generación actual.
Como operador de selección se implementará el torneo binario ya descrito anteriormente, para la elección de los mejores padres a cruzar.

El operador de cruce es el CA, descrito anteriormente, para generar en cada cruce a partir de dos padres un descendiente. La probabilidad de cruce será de 0.7.

Para la mutación, se empleará la función generarVecino descrita anteriormente, bajo una

probabilidad de mutación en población de un 0.001.

Por último el criterio de parada del algoritmo es no superar las 15000 evaluaciones de la función objetivo (1-NN).

```
AGG CA (Data)
        //Creamos una poblacion de 30 individuos aleatoria:
        Desde i = 0 hasta i < 30
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        Para cada P_i ∈ poblacion actual
                tasas actuales ⇐ knn(Data, Data, P_i)
        Fin para cada

        //empezamos generacion tras generacion hasta cumplir
        //El criterio de parada:
        Mientras (num. evaluaciones < 15000)
                //Calculamos el mejor padre de la poblacion actual:
                Padre_mejor = Max(tasas poblacion actual)

                //Obtenemos los padres (parejas) a cruzar,
                //con una probabilidad de cruce del 70%
                parejas a cruzar = 0.7 * 30

                //Llamamos al torneo binario, obteniendo los padres
                    deseados,
                // en este caso, el doble del tamanio de la poblacion
                    actual, puesto que
                // el operador de cruce CA devuelve un unico descendiente
                    .
                padres a cruzar = torneoBinario(poblacion actual)
                //Realizamos los cruces, obteniendo dos hijos por cruce
                Desde i = 0 hasta i < parejas a cruzar
                        nueva poblacion ⇐ cruceAritmetico(pareja_i, pareja_{i+1})
                Fin Desde

                //Aniadimos los padres que no se han cruzado hasta llegar
                // a una nueva poblacion de 30:
                Mientras Tam(nueva poblacion) < 30
                        nueva poblacion ⇐ padres no se han cruzado
                Fin Mientras

                //Calculamos el numero de individuos a mutar, con
                // una probabilidad de 0.001:
                individuos a mutar = 0.001 * 30

                //Mutamos de forma aleatoria los n individuos a mutar:
                Desde i = 0 hasta i < individuos a mutar
                        nueva poblacion_aleatorio = generarVecino (nueva
                            poblacion_aleatorio)
                Fin Desde
```

```
                //Calculamos las tasas de la nueva poblacion
                Para cada Pi ∈ nueva poblacion
                        nuevas tasas ⇐ knn(Data, Data, Pi)
                Fin para cada

                //Si hemos perdido al mejor padre lo cambiamos por
                //el peor de la poblacion.
                Si (hemos perdido al mejor padre)
                        Min(tasas actual) = mejor padre
                Fin Si

                //La nueva poblacion es ahora la actual:
                poblacion actual = nueva poblacion
                tasas actual = nuevas tasas


        Fin Mientras

        Devolver mejor individuo poblacion actual
```

3.2.3. AGE BLX

El AGE BLX se trata de un algoritmo que implementa un esquema de evolución estacionario, mediante un esquema de reemplazamiento en el que los dos hijos de la nueva generación compite con los dos peores individuos de la generación actual, reemplazando a estos en caso de ser mejores.
+
Como operador de selección se implementará el torneo binario ya descrito anteriormente, para la elección de los mejores padres a cruzar.

El operador de cruce es el BLX, descrito anteriormente, para generar en cada cruce a partir de dos padres dos descendientes. La probabilidad de cruce será de 0.7.

Para la mutación, se empleará la función generarVecino descrita anteriormente, bajo una probabilidad de mutación en población de un 0.001.

Por último el criterio de parada del algoritmo es no superar las 15000 evaluaciones de la función objetivo (1-NN).

```
AGE BLX (Data)
        //Creamos una poblacion de 30 individuos aleatoria:
        Desde i = 0 hasta i < 30
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        Para cada Pi ∈ poblacion actual
                tasas actuales ⇐ knn(Data, Pi)
        Fin para cada

        //empezamos generacion tras generacion hasta cumplir
```

```
//El criterio de parada:
Mientras (num. evaluaciones < 15000)
        //Calculamos el mejor padre de la poblacion actual:
        Padre_mejor = Max(tasas poblacion actual)

        //Obtenemos los padres (parejas) a cruzar,
        //con una probabilidad de cruce del 1, es decir
        // unicamente cruzamos 1 pareja:
        num parejas = 1

        //Llamamos al torneo binario, obteniendo los padres
            deseados
        padres a cruzar = torneoBinario(poblacion actual)
        //Realizamos los cruces, obteniendo dos hijos

         h_1, h_2 ⇐ cruceBLX(pareja_0, pareja_1)

        //Igualamos la nueva poblacion a la poblacion actual:
        nueva poblacion = poblacion actual
        nueva tasas = tasas actual
        //Calculamos el numero de individuos a mutar, con
        // una probabilidad de 0.001:
        individuos a mutar = 0.001 * 30

        //Mutamos de forma aleatoria los n individuos a mutar:
        Desde i = 0 hasta i < individuos a mutar
                nueva poblacion_aleatorio = generarVecino (nueva
                    poblacion_aleatorio)
        Fin Desde

        //Comparamos los dos hijos h_1 y h_2 con los dos peores de
            la poblacion:
        Si (Tasa(h_1) > peoresDosIndividuos(poblacion actual))
                //sustituimos los individuos:
                nueva poblacion(peor individuo1) = h_1
                // y sus tasas
                Tasa(peor individuo1) = Tasa(h_1)
        Fin Si

        Si (Tasa(h_2) > peoresDosIndividuos(poblacion actual))
                //sustituimos los individuos:
                nueva poblacion(peor individuo2) = h_2
                // y sus tasas
                Tasa(peor individuo2) = Tasa(h_2)
        Fin Si

        //La nueva poblacion es ahora la actual:
        poblacion actual = nueva poblacion
        tasas actual = nuevas tasas


Fin Mientras

Devolver mejor individuo poblacion actual
```

3.2.4. AGE CA

El AGE CA se trata de un algoritmo que implementa un esquema de evolución estacionario, mediante un esquema de reemplazamiento en el que los dos hijos de la nueva generación compite con los dos peores individuos de la generación actual, reemplazando a estos en caso de ser mejores.
+
Como operador de selección se implementará el torneo binario ya descrito anteriormente, para la elección de los mejores padres a cruzar.

El operador de cruce es el CA, descrito anteriormente, para generar en cada cruce a partir de dos padres dos descendientes. La probabilidad de cruce será de 0.7.

Para la mutación, se empleará la función generarVecino descrita anteriormente, bajo una probabilidad de mutación en población de un 0.001.

Por último el criterio de parada del algoritmo es no superar las 15000 evaluaciones de la función objetivo (1-NN).

```
AGE CA (Data)
        //Creamos una poblacion de 30 individuos aleatoria:
        Desde i = 0 hasta i < 30
                Poblacion actual <- generaSolucionAleatoria()
        Fin Desde

        Para cada P_i ∈ poblacion actual
                tasas actuales ⇐ knn(Data, P_i)
        Fin para cada

        //empezamos generacion tras generacion hasta cumplir
        //El criterio de parada:
        Mientras (num. evaluaciones < 15000)
                //Calculamos el mejor padre de la poblacion actual:
                Padre_mejor = Max(tasas poblacion actual)

                //Obtenemos los padres (parejas) a cruzar,
                //con una probabilidad de cruce del 1, es decir
                // unicamente cruzamos 2 parejas para obtener 2 hijos:
                num parejas = 2

                //Llamamos al torneo binario, obteniendo los padres
                    deseados
                padres a cruzar = torneoBinario(poblacion actual)
                //Realizamos los cruces, obteniendo dos hijos

                h_1 ⇐ cruceCA(pareja_0, pareja_1)
                h_2 ⇐ cruceCA(pareja_2, pareja_3)

                //Igualamos la nueva poblacion a la poblacion actual:
                nueva poblacion = poblacion actual
                nueva tasas = tasas actual
```

```
            //Calculamos el numero de individuos a mutar, con
            // una probabilidad de 0.001:
            individuos a mutar = 0.001 * 30

            //Mutamos de forma aleatoria los n individuos a mutar:
            Desde i = 0 hasta i < individuos a mutar
                    nueva poblacion_aleatorio = generarVecino (nueva
                        poblacion_aleatorio)
            Fin Desde

            //Comparamos los dos hijos h_1 y h_2 con los dos peores de
                la poblacion:
            Si (Tasa(h_1) > peoresDosIndividuos(poblacion actual))
                    //sustituimos los individuos:
                    nueva poblacion(peor individuo1) = h_1
                    // y sus tasas
                    Tasa(peor individuo1) = Tasa(h_1)
            Fin Si

            Si (Tasa(h_2) > peoresDosIndividuos(poblacion actual))
                    //sustituimos los individuos:
                    nueva poblacion(peor individuo2) = h_2
                    // y sus tasas
                    Tasa(peor individuo2) = Tasa(h_2)
            Fin Si

            //La nueva poblacion es ahora la actual:
            poblacion actual = nueva poblacion
            tasas actual = nuevas tasas


    Fin Mientras

    Devolver mejor individuo poblacion actual
```

3.3. Algoritmos meméticos

La implementación realizada para los algoritmos meméticos consiste en hibridar los algoritmos generacionales anteriormente analizados que mejor resultado me han dado, en mi caso, el AGG-CA, incluyendo en este un proceso de búsqueda local para obtener, aparte de la exploración proporcionada por el algoritmo genético, tener también una explotación mediante el uso de dicha búsqueda local.

Utilizaremos para los meméticos una población de tamaño 10.
Estudiaremos las tres posibilidades de hibridación siguientes:

3.3.1. AM-(10, 1.0)

Consiste en que dado el algoritmo AGG-CA mencionado, cada 10 generaciones aplicamos una búsqueda local sobre todos los cromosomas de la población, mejorando cada

cromosoma así de forma local.

El pseudocódigo sería el siguiente:

```
AM-(10, 1.0) (Data)
//Creamos una poblacion de 10 individuos aleatoria:
Desde i = 0 hasta i < 10
        Poblacion actual <- generaSolucionAleatoria()
Fin Desde

Para cada P_i ∈ poblacion actual
        tasas actuales ⇐ knn(Data, Data, P_i)
Fin para cada

//empezamos generacion tras generacion hasta cumplir
//El criterio de parada:
Mientras (num. evaluaciones < 15000)
        //Calculamos el mejor padre de la poblacion actual:
        Padre_mejor = Max(tasas poblacion actual)

        //Obtenemos los padres (parejas) a cruzar,
        //con una probabilidad de cruce del 70%
        parejas a cruzar = 0.7 * 10

        //Llamamos al torneo binario, obteniendo los padres deseados,
        // en este caso, el doble del tamanio de la poblacion actual,
            puesto que
        // el operador de cruce CA devuelve un unico descendiente.
        padres a cruzar = torneoBinario(poblacion actual)
        //Realizamos los cruces, obteniendo dos hijos por cruce
        Desde i = 0 hasta i < parejas a cruzar
                nueva poblacion ⇐ cruceAritmetico(pareja_i, pareja_{i+1})
        Fin Desde

        //Aniadimos los padres que no se han cruzado hasta llegar
        // a una nueva poblacion de 10:
        Mientras Tam(nueva poblacion) < 10
                nueva poblacion ⇐ padres no se han cruzado
        Fin Mientras

        //Calculamos el numero de individuos a mutar, con
        // una probabilidad de 0.001:
        individuos a mutar = 0.001 * 10

        //Mutamos de forma aleatoria los n individuos a mutar:
        Desde i = 0 hasta i < individuos a mutar
                nueva poblacion_aleatorio = generarVecino (nueva poblacion_aleatorio)
        Fin Desde

        //Calculamos las tasas de la nueva poblacion
        Para cada P_i ∈ nueva poblacion
                nuevas tasas ⇐ knn(Data, Data, P_i)
        Fin para cada
```

```
        //Si hemos perdido al mejor padre lo cambiamos por
        //el peor de la poblacion.
        Si (hemos perdido al mejor padre)
                Min(tasas actual) = mejor padre
        Fin Si

        //La nueva poblacion es ahora la actual:
        poblacion actual = nueva poblacion
        tasas actual = nuevas tasas


        //Si ya llevamos 10 generaciones aplicamos la busqueda local:
        Si (generaciones es multiplo de 10)
                Para cada c_i \in poblacion actual
                        c_i = BL (Data, c_1)
                        Tasas actual (c_i) = Tasa(c_i)
                Fin para cada
        Fin Si


Fin Mientras


Devolver mejor individuo poblacion actual
```

3.3.2. AM-(10, 0.1)

Consiste en que dado el algoritmo AGG-CA mencionado, cada 10 generaciones aplicamos una búsqueda local sobre un 10 % aleatorio de los cromosomas de la población.
El pseudocódigo sería el siguiente:

```
AM-(10, 0.1) (Data)
//Creamos una poblacion de 10 individuos aleatoria:
Desde i = 0 hasta i < 10
        Poblacion actual <- generaSolucionAleatoria()
Fin Desde

Para cada P_i \in poblacion actual
        tasas actuales \Leftarrow knn(Data, Data, P_i)
Fin para cada

//empezamos generacion tras generacion hasta cumplir
//El criterio de parada:
Mientras (num. evaluaciones < 15000)
        //Calculamos el mejor padre de la poblacion actual:
        Padre_{mejor} = Max(tasas poblacion actual)

        //Obtenemos los padres (parejas) a cruzar,
        //con una probabilidad de cruce del 70%
        parejas a cruzar = 0.7 * 10
```

```
        //Llamamos al torneo binario, obteniendo los padres deseados,
        // en este caso, el doble del tamanio de la poblacion actual,
            puesto que
        // el operador de cruce CA devuelve un unico descendiente.
        padres a cruzar = torneoBinario(poblacion actual)
        //Realizamos los cruces, obteniendo dos hijos por cruce
        Desde i = 0 hasta i < parejas a cruzar
                nueva poblacion ⇐ cruceAritmetico(pareja_i, pareja_{i+1})
        Fin Desde

        //Aniadimos los padres que no se han cruzado hasta llegar
        // a una nueva poblacion de 10:
        Mientras Tam(nueva poblacion) < 10
                nueva poblacion ⇐ padres no se han cruzado
        Fin Mientras

        //Calculamos el numero de individuos a mutar, con
        // una probabilidad de 0.001:
        individuos a mutar = 0.001 * 10

        //Mutamos de forma aleatoria los n individuos a mutar:
        Desde i = 0 hasta i < individuos a mutar
                nueva poblacion_{aleatorio} = generarVecino (nueva poblacion_{aleatorio})
        Fin Desde

        //Calculamos las tasas de la nueva poblacion
        Para cada P_i ∈ nueva poblacion
                nuevas tasas ⇐ knn(Data, Data, P_i)
        Fin para cada

        //Si hemos perdido al mejor padre lo cambiamos por
        //el peor de la poblacion.
        Si (hemos perdido al mejor padre)
                Min(tasas actual) = mejor padre
        Fin Si

        //La nueva poblacion es ahora la actual:
        poblacion actual = nueva poblacion
        tasas actual = nuevas tasas


        //Si ya llevamos 10 generaciones aplicamos la busqueda local:
        Si (generaciones es multiplo de 10)
                Para cada c_i \in 0.1* mejores poblacion actual
                        c_i = BL (Data, c_1)
                        Tasas actual (c_i) = Tasa(c_i)
                Fin para cada
        Fin Si


Fin Mientras

Devolver mejor individuo poblacion actual
```

3.3.3. AM-(10, 0.1mej)

Consiste en que dado el algoritmo AGG-CA mencionado, cada 10 generaciones aplicamos una búsqueda local sobre los 0.1*N mejores cromosomas de la población.
El pseudocódigo sería el siguiente:

```
AM-(10, 0.1mej) (Data)
//Creamos una poblacion de 10 individuos aleatoria:
Desde i = 0 hasta i < 10
        Poblacion actual <- generaSolucionAleatoria()
Fin Desde

Para cada P_i ∈ poblacion actual
        tasas actuales ⇐ knn(Data, Data, P_i)
Fin para cada

//empezamos generacion tras generacion hasta cumplir
//El criterio de parada:
Mientras (num. evaluaciones < 15000)
        //Calculamos el mejor padre de la poblacion actual:
        Padre_mejor = Max(tasas poblacion actual)

        //Obtenemos los padres (parejas) a cruzar,
        //con una probabilidad de cruce del 70%
        parejas a cruzar = 0.7 * 10

        //Llamamos al torneo binario, obteniendo los padres deseados,
        // en este caso, el doble del tamanio de la poblacion actual,
           puesto que
        // el operador de cruce CA devuelve un unico descendiente.
        padres a cruzar = torneoBinario(poblacion actual)
        //Realizamos los cruces, obteniendo dos hijos por cruce
        Desde i = 0 hasta i < parejas a cruzar
                nueva poblacion ⇐ cruceAritmetico(pareja_i, pareja_{i+1})
        Fin Desde

        //Aniadimos los padres que no se han cruzado hasta llegar
        // a una nueva poblacion de 10:
        Mientras Tam(nueva poblacion) < 10
                nueva poblacion ⇐ padres no se han cruzado
        Fin Mientras

        //Calculamos el numero de individuos a mutar, con
        // una probabilidad de 0.001:
        individuos a mutar = 0.001 * 10

        //Mutamos de forma aleatoria los n individuos a mutar:
        Desde i = 0 hasta i < individuos a mutar
                nueva poblacion_aleatorio = generarVecino (nueva poblacion_aleatorio)
        Fin Desde

        //Calculamos las tasas de la nueva poblacion
        Para cada P_i ∈ nueva poblacion
```

```
                nuevas tasas ⇐ knn(Data, Data, P_i)
        Fin para cada

        //Si hemos perdido al mejor padre lo cambiamos por
        //el peor de la poblacion.
        Si (hemos perdido al mejor padre)
                Min(tasas actual) = mejor padre
        Fin Si

        //La nueva poblacion es ahora la actual:
        poblacion actual = nueva poblacion
        tasas actual = nuevas tasas


        //Si ya llevamos 10 generaciones aplicamos la busqueda local:
        Si (generaciones es multiplo de 10)
                Para cada c_i \in 0.1*poblacion actual
                        c_i = BL (Data, c_1)
                        Tasas actual (c_i) = Tasa(c_i)
                Fin para cada
        Fin Si


Fin Mientras


Devolver mejor individuo poblacion actual
```

4. Algoritmo de comparación

En el dominio del problema a tratar, la implementación de un método de comparación se realiza mediante el método RELIEF.
Se trata de una solución de tipo Greedy.
Dicho método partimos de un vector de pesos inicializado a 0 inicialmente y en cada paso modificamos dichos pesos en función, para cada uno de los ejemplos del conjunto de entrenamiento, de su enemigo más cercano y su amigo más cercano.

Definimos el enemigo más cercano como aquel ejemplo con clase diferente que se encuentra a menor distancia euclídea; y el amigo más cercano de la misma forma pero que tiene la misma clase que él. NUNCA COMPARAMOS LA DISTANCIA ENTRE EL CONSIGO MISMO (Leave one out), sino obtendríamos siempre una tasa cercana al 100 %
Tras el cálculo de W, y si es necesario, se normaliza W.
El pseudocódigo sería el siguiente:

```
RELIEF(train, test)
        //Inicializamos los pesos a 0:
        Para todo w_i ∈ w
                w_i = 0
        Fin para todo
```

```
//Recorremos el conjunto test:
Para todo a_i ∈ Test
        //inicializamos las distancias minimas al enemigo
        // y al amigo a infinito
        d_min^amigo = ínf
        d_min^enemigo = ínf

        //Ahora recorremos el conjunto train
        // calculando las distancias, buscando el min amigo
        // y enemigo:

        Para todo b_i ∈ Train
                Si a_i ≠ b_i  //leave one out
                        Si etiqueta(b_i ≠ a_i) && (calculaDistancia(a_i
                            , b_i) < d_min^enemigo)
                                d_min^enemigo = calculaDistancia(a_i, b_i)

                        Sino Si(calculaDistancia(a_i, b_i) < d_min^enemigo)
                                d_min^amigo = calculaDistancia(a_i, b_i)
                Fin para todo
        Fin para todo

        //Ahora actualizamos los pesos
        w = w + |a_i − a^enemigo| − |a_i − a_amigo|

Fin para todo


Devolver W
```

5. Procedimiento para la realización de la práctica

Para la realización de la práctica no he hecho uso de ningúna implementación ni código externos, todo ha sido desarrollado por mi mismo.

5.1. Manual de usuario

Si se desea realizar una réplica de las pruebas aquí descritas serán necesarios los siguientes pasos:

1. En primer lugar, será necesario ejecutar el script en R "practica1.R", cuya función es leer los archivos que se encuentran en formato '.arff', normalizarlos, y escribirlos en ficheros '.csv' (almacenados en la carpeta "datos") para facilitar de esta forma la futura lectura.

2. Una vez tenemos los datos normalizados, ejecutar el makefile incluido, de esta forma obtendremos un ejecutable "practica1".

3. Por último lanzar dicho programa, indicando como argumento el nombre del conjunto de datos que se desea utilizar para la realización de los algoritmos, por ejemplo: "./practica1 cancer", de esta forma utilizaríamos la base de datos de cáncer de mama.

4. Tras esto se ejecutarán TODOS los algoritmos indicando tanto la tasa obtenida para cada uno, como los tiempos, la tasa media de todas las particiones y su correspondiente tiempo medio.

6. Experimentos y análisis de resultados

Para la realización de los experimento utilizaremos 3 conjuntos de datos distintos, uno para predecir si se posee cáncer de mama, otro para detectar posible correo 'spam' y un tercero, para clasificar datos de un sonar.

Cómo método de validación utilizaremos el método 5x2-cross validation (cv). Para ello, se generarán 5 particiones distintas del conjunto de datos proporcionado en subconjuntos de entrenamiento y prueba realizadas al 50 %. Para cada partición, se aprenderá primero el clasificador con una parte y se validará con la otra y viceversa, resultando en 10 ejecuciones de cada algoritmo sobre cada conjunto de datos.

Cómo medida de validación, utilizaremos la tasa de acierto del clasificador 1-NN sobre el conjunto de prueba. El resultado final será la media de los 10 valores obtenidos, uno para cada ejecución del algoritmo.

También emplearemos cómo método de comparación el tiempo de ejecución empleado por cada uno. Las semillas utilizadas para la realización de todas las pruebas son las siguientes:

```
default_random_engine generator (5); //para los números aleatorios
normal_distribution<double> distribution (0.0,0.3); //para la distribución normal
```

Los tiempos obtenidos y las tasas para cada algoritmo de forma individual son las siguientes:

Figura 6.1: Resultados 1NN

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	83,6538	0,00061	93,3333	0,0027	75,6522	0,0028
Partición 1-2	74,0385	0,00047	94,7183	0,0028	76,0870	0,0023
Partición 2-1	84,6154	0,00060	93,6842	0,0027	76,0870	0,0026
Partición 2-2	82,6923	0,00061	95,7746	0,0027	78,6957	0,0029
Partición 3-1	90,3846	0,00060	95,0877	0,0027	75,6522	0,0029
Partición 3-2	76,9231	0,00060	93,6620	0,0026	78,6957	0,0030
Partición 4-1	86,5385	0,00060	93,3333	0,0025	77,3913	0,0029
Partición 4-2	83,6538	0,00060	94,3662	0,0020	78,2609	0,0030
Partición 5-1	82,6923	0,00060	96,1404	0,0024	81,7391	0,0029
Partición 5-2	80,7692	0,00059	94,7183	0,0026	78,6957	0,0030
Media	82,5962	0,00059	94,4818	0,0026	77,6957	0,0028

Figura 6.2: Resultados RELIEF

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	79,8077	0,0036	94,3860	0,0141	74,7826	0,0156
Partición 1-2	70,1923	0,0028	95,0704	0,0133	77,8261	0,0184
Partición 2-1	89,4231	0,0041	93,6842	0,0138	80,8696	0,0193
Partición 2-2	83,6538	0,0029	95,0704	0,0139	75,2174	0,0160
Partición 3-1	95,1923	0,0049	95,4386	0,0188	76,9565	0,0208
Partición 3-2	78,8462	0,0040	94,7183	0,0156	80,0000	0,0169
Partición 4-1	86,5385	0,0037	93,3333	0,0156	77,3913	0,0184
Partición 4-2	84,6154	0,0048	95,4225	0,0149	83,4783	0,0184
Partición 5-1	81,7308	0,0028	95,0877	0,0152	83,0435	0,0188
Partición 5-2	82,6923	0,0039	95,7746	0,0149	79,5652	0,0184
Media	83,2692	0,0038	94,7986	0,0150	78,9131	0,0181

Figura 6.3: Resultados BL

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	79,8077	0,7194	93,6842	1,5384	76,9565	3,3082
Partición 1-2	78,8462	0,7248	95,4225	1,5336	80,0000	3,3365
Partición 2-1	84,6154	0,7249	91,2281	1,5457	72,6087	3,2632
Partición 2-2	81,7308	0,7184	93,6620	1,5400	77,8261	3,2875
Partición 3-1	89,4231	0,7234	94,3860	1,5289	78,6957	3,2653
Partición 3-2	76,9231	0,7465	95,7746	1,5269	78,2609	3,2933
Partición 4-1	84,6154	0,7122	91,5789	1,5744	74,7826	3,3166
Partición 4-2	84,6154	0,7044	94,7183	1,5482	78,6957	3,2756
Partición 5-1	77,8846	0,7108	95,0877	1,5179	80,4348	3,2813
Partición 5-2	77,8846	0,7130	92,2535	1,5260	75,2174	3,2742
Media	81,6346	0,7198	93,7796	1,5380	77,3478	3,2902

Figura 6.4: Resultados AGG-BLX

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	76,9231	8,8300	93,6842	38,4026	76,9565	43,5713
Partición 1-2	75,9615	8,8738	94,7183	38,3077	79,5652	43,4449
Partición 2-1	86,5385	8,7820	90,5263	38,3193	77,8261	42,9739
Partición 2-2	81,7308	8,7714	95,0704	38,4520	79,1304	43,0604
Partición 3-1	90,3846	8,8839	95,0877	38,7841	76,5217	42,9486
Partición 3-2	81,7308	8,8413	93,3099	38,1193	76,9565	42,9567
Partición 4-1	75,9615	8,7965	94,3860	38,5654	76,9565	43,2797
Partición 4-2	83,6538	8,8314	94,3662	38,2005	83,0435	43,1012
Partición 5-1	84,6154	8,7806	95,4386	38,5096	80,0000	42,6656
Partición 5-2	80,7692	8,8016	95,4225	38,4573	81,3043	43,0038
Media	81,8269	8,8192	94,2010	38,4118	78,8261	43,1006

Figura 6.5: Resultados AGG-CA

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	79,8077	8,7625	93,3333	38,1796	75,6522	43,4818
Partición 1-2	76,9231	8,6952	95,4225	38,1895	77,3913	43,5018
Partición 2-1	83,6538	8,7087	93,6842	38,2629	77,3913	42,8629
Partición 2-2	80,7692	8,7559	95,0704	38,2096	77,8261	43,1403
Partición 3-1	88,4615	8,8520	95,4386	38,6657	76,9565	42,9882
Partición 3-2	77,8846	8,8235	93,6620	38,1347	78,6957	42,9263
Partición 4-1	85,5769	8,7919	92,9825	38,9644	78,6957	43,1858
Partición 4-2	80,7692	8,8148	93,6620	39,1913	78,2609	42,9395
Partición 5-1	80,7692	8,7163	96,1404	38,5565	82,1739	42,6303
Partición 5-2	78,8462	8,7696	94,0141	38,4111	78,2609	42,8151
Media	81,3461	8,7690	94,3410	38,4765	78,1305	43,0472

Figura 6.6: Resultados AGE-BLX

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	81,7308	8,8940	92,6316	38,5765	77,8261	43,6500
Partición 1-2	74,0385	8,8662	95,4225	38,3101	78,2609	43,6048
Partición 2-1	81,7308	8,8569	94,0351	38,3448	78,2609	42,9619
Partición 2-2	82,6923	8,8683	91,9014	38,4618	79,1304	43,1084
Partición 3-1	87,5000	8,9361	93,6842	38,8297	80,4348	43,2135
Partición 3-2	75,9615	8,9704	94,3662	38,3249	75,2174	43,0196
Partición 4-1	85,5769	8,9122	92,9825	38,7402	78,6957	43,3752
Partición 4-2	85,5769	8,9255	95,0704	38,3095	81,3043	43,2506
Partición 5-1	78,8462	8,9193	94,0351	38,6321	82,6087	42,8764
Partición 5-2	77,8846	8,9068	93,3099	38,3831	80,8696	43,1885
Media	81,1539	8,9056	93,7439	38,4913	79,2609	43,2249

Figura 6.7: Resultados AGE-CA

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	79,8077	8,8710	93,3333	38,3510	77,3913	5,0000
Partición 1-2	70,1923	8,8391	96,1268	38,2720	79,5652	43,6149
Partición 2-1	86,5385	8,8652	93,6842	38,4760	76,0870	43,1672
Partición 2-2	76,9231	8,8490	95,7746	38,4370	79,1304	43,3074
Partición 3-1	89,4231	8,8964	94,3860	38,7800	76,9565	43,1629
Partición 3-2	77,8846	8,9164	94,3662	38,2010	76,0870	43,1479
Partición 4-1	85,5769	8,8695	92,9825	38,7210	75,2174	43,4442
Partición 4-2	84,6154	8,9157	94,7183	38,4870	81,7391	43,0141
Partición 5-1	81,7308	8,9079	95,4386	38,7050	83,4783	42,6802
Partición 5-2	80,7692	8,8922	94,3662	38,3060	80,0000	43,0972
Media	81,3462	8,8822	94,5177	38,4736	78,5652	39,3636

Figura 6.8: Resultados AM-1

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	76,9231	14,3574	93,6842	46,9917	78,2609	67,0086
Partición 1-2	79,8077	14,2807	93,3099	46,8734	80,4348	67,1830
Partición 2-1	86,5385	14,3445	92,2807	46,9325	75,6522	65,9955
Partición 2-2	80,7692	14,2671	95,0704	46,9279	77,3913	66,4277
Partición 3-1	87,5000	14,5204	95,0877	47,4047	80,8696	66,3099
Partición 3-2	76,9231	14,4097	92,2535	46,6747	80,4348	66,3237
Partición 4-1	81,7308	14,3837	93,3333	47,2194	75,2174	66,7749
Partición 4-2	79,8077	14,5036	93,6620	46,8756	80,4348	66,4382
Partición 5-1	83,6538	14,3285	94,7368	47,2038	83,4783	65,7203
Partición 5-2	81,7308	14,2973	93,6620	46,8812	79,5652	66,1662
Media	81,5385	14,3693	93,7081	46,9985	79,1739	66,4348

Figura 6.9: Resultados AM-01

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	84,6154	9,2448	93,3333	39,4646	76,9565	47,1028
Partición 1-2	75,0000	9,2136	94,0141	39,3695	74,7826	47,0861
Partición 2-1	77,8846	9,2256	93,3333	39,5893	79,5652	46,0999
Partición 2-2	81,7308	9,3042	93,6620	39,6184	76,0870	46,5718
Partición 3-1	91,3462	9,3380	93,6842	39,7924	78,6957	46,3926
Partición 3-2	77,8846	9,2898	93,6620	39,3678	76,5217	46,5583
Partición 4-1	85,5769	9,2684	92,2807	39,7536	75,6522	46,7849
Partición 4-2	80,7692	9,2262	92,9577	39,3684	78,2609	46,4618
Partición 5-1	79,8077	9,2755	94,7368	39,7074	79,1304	45,9172
Partición 5-2	82,6923	9,1843	94,7183	39,3059	79,5652	46,4023
Media	81,7308	9,2570	93,6382	39,5337	77,5217	46,5378

Figura 6.10: Resultados AM-01MEJ

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	84,6154	9,2432	91,9298	39,5570	76,5217	46,7848
Partición 1-2	76,9231	9,2291	91,5493	39,4840	74,7826	46,9261
Partición 2-1	80,7692	9,2300	92,6316	39,5350	75,6522	46,2066
Partición 2-2	82,6923	9,2482	95,0704	39,5690	81,7391	46,4188
Partición 3-1	90,3846	9,2705	93,3333	39,9630	76,5217	46,3816
Partición 3-2	79,8077	9,2928	95,0704	39,3370	79,5652	46,3662
Partición 4-1	84,6154	9,2291	92,9825	39,8920	75,2174	46,5992
Partición 4-2	83,6538	9,2672	95,0704	39,5010	79,1304	46,4398
Partición 5-1	72,1154	9,1295	94,7368	39,6800	81,7391	46,0196
Partición 5-2	80,7692	9,2413	93,3099	39,4900	80,4348	46,4262
Media	81,6346	9,2381	93,5684	39,6008	78,1304	46,4569

Ahora pasemos a realizar una tabla comparativa con las medias, tanto de tiempo como de tasas obtenidas para cada uno de los algoritmos para poder realizar así un análisis de forma global:

Figura 6.11: Comparativa global

	Sonar		Wdbc		Spambase	
	%_clas	T	*%_clas*	T	*%_clas*	T
1-NN	82,5962	0,0006	94,4818	0,0026	77,6957	0,0028
RELIEF	83,2692	0,0038	94,7986	0,0150	78,9131	0,0181
BL	81,6346	0,7198	93,7796	1,5380	77,3478	3,2902
AGG-BLX	81,8269	8,8192	94,2010	38,4118	78,8261	43,1006
AGG-CA	81,3461	8,7690	94,3410	38,4765	78,1305	43,0472
AGE-BLX	81,1539	8,9056	93,7439	38,4913	79,2609	43,2249
AGE-CA	81,3462	8,8822	94,5177	38,4736	78,5652	39,3636
AM-(10,1.0)	81,5385	14,3693	93,7081	46,9985	79,1739	66,4348
AM-(10,0.1)	81,7308	9,2570	93,6382	39,5337	77,5217	46,5378
AM-(10,0.1mej)	81,6346	9,2381	93,5684	39,6008	78,1304	46,4569

Realicemos ahora un anánisis de los datos obtenidos e intentemos razonar el porqué de dichos datos.

Si analizamos a primera vista las tasas obtenidas según los tres distintos conjuntos, lo primero que nos damos cuenta es de la gran importancia que tiene el mero hecho de que el conjunto de datos que usamos sea o no represntativo.
Por ejemplo, nos damos cuenta de que el conjunto de datos de cancer (Wdbc) es el que posee los datos más representativos de los tres, lo cuál facilita el aprendizaje. Es por eso que es en este en el que se obtienen las mayores tasas de clasificación, seguidas por las de Sonar y por último las de Spam.

6.1. Análisis genéticos

Si nos centramos concretamente en la familia de algoritmos genéticos podemos observar, evidentemente, la necesidad elevada de cálculo en comparación con métodos Greedy como RELIEF, lo cual influye significativamente en el tiempo del algoritmo.
Por otra parte, los genéticos nos ofrecen la posibilidad de realizar exploración en el espacio de búsqueda, pudiendo de esta forma evitar máximos locales, como puede pasar con el RELIEF.

Es interesante observar con más profundidad los resultados obtenidos de algoritmos genéticos según su esquema de evolución:

6.2. Generacionales VS Estacionarios

Si comparamos generacionales y estacionarios estrictamente observando la tabla, podríamos pensar que ambos son iguales, puesto que tanto los tiempos como las tasas obtenidas

en los tres conjuntos no difieren apenas en unas décimas o incluso centésimas entre ellos.

Pero si analizamos la funcionalidad de ambos, podemos deducir que la principal diferencia entre estacionarios y generacionales es la diversidad que producen.
En esquema estacional, únicamente se reemplazan dos individuos como máximo en una nueva generación, e incluso puede pasar que no se reemplace ninguno (puesto que los hijos son peores que los peores individuos de la población actual) y obtengamos una generación igual a la anterior.

En cambio en el modelo generacional, el porcentaje de la población reemplazada por los descendientes es mucho mayor, en nuestro caso un 0.7 como hemos visto. Esto puede generar mayor diversidad a la hora de la generación de descendientes, ampliando así el espacio de búsqueda.
En cualquier caso, dependerá del tipo de problema que deseemos tratar, para así poder adecuar mejor un esquema de evolución u otro.

6.3. BLX VS CA

En una vista apriori sin comparar datos de los dos operadores de cruce, podríamos pensar que el operador BLX es mejor que el Cruce Aritmético por el hecho de ser más complejo y diversificar, en parte, con cierta componente aleatoria la reproducción de los padres.

Pero en la práctica, a la hora de la verdad, podemos ver que las tasas obtenidas por ambos son similares, incluso dependiendo del conjunto de datos aquellos con operador de cruce aritmético obtienen ligeramente mejores tasas, lo cuál nos lleva a hacernos la siguiente pregunta: ¿Complejidad implica mejora?, en este caso la más simple nos da casi las misma soluciones, pero con tiempos ligeramente menores, lo que implica que a veces lo sencillo es lo mejor, siguiendo el principio de la navaja de Ockham.

6.4. Exploración VS explotación

Si comparamos los resultados obtenidos por nuestro algoritmo de explotación (BL) con nuestros algoritmos genéticos, vemos que las tasas obtenidas por los genéticos son una o dos décimas superiores a las obtenidas por la búsqueda local. La principal razón de esto son los máximos locales.
El principal problema de la búsqueda local son los máximos locales, puesto que se trata de un algoritmo de explotación, es factible que se de el caso de que estemos explotando una determinada zona, obteniendo buenos resultados, pero estemos atascados en una cota local, de forma que no podamos llegar nunca a soluciones que se encuentran en distintas regiones del espacio de búsqueda.

Esto mismo tratan de solucionar los algoritmos de exploración. Se trata básicamente de sacrificar la obtención de una buena solución en una única región concreta del espacio de

búsqueda, por una exploración más amplia del mismo.

Entonces, ¿que es mejor, sacrificar precisión por amplitud de miras, o lo contrario?.
Una buena solución es una hibridación entre explotación y exploración. Los algoritmos meméticos implementados nos permiten justo eso, realizar una exploración primero mediante un algoritmo genético, permitiendo así dirigirse a una región del espacio de búsqueda con buenas soluciones, y una vez allí aplicar una búsqueda local para realizar una exploración de este. De hecho, las tasas obtenidas por los meméticos son superiores a las obtenidas por una simple búsqueda local individual.

7. Conclusión

Como hemos visto en el análisis, la elección de un buen algoritmo de búsqueda es importantísima a la hora de la resolución de un problema, y se encuentra muy ligada a el ámbito de el mismo.

No existe un claro vencedor, puesto que como hemos dicho entran muchas variables en juego, dependientes en gran parte del dominio del problema.