METAHEURÍSTICAS
Grado en Ingeniería Informática
Universidad de Granada

Práctica 1.b: Técnicas de Búsqueda basadas en Poblaciones para el Problema del Aprendizaje de Pesos en Características

---

Carlos Manuel Sequí Sánchez
DNI: 20 48 69 26
Grupo de prácticas 2.
Horario: viernes, 17:30-19:30

2016-2017

Algoritmos considerados:

- KNN.
- RELIEF.
- Local Search.
- AGG-BLX.
- AGG-CA.
- AGE-BLX.
- AGE-CA.
- AM-(10,1.0).
- AM-(10,0.1).
- AM-(10,0.1mej).

Índice

Índice de figuras

1. Descripción del problema: APC

Nota: la partición y normalización de datos han sido realizadas en el lenguaje de programación R, de tal manera que paso los datos a un fichero CSV normalizado con la columna de la clase de cada ejemplo puesta al principio. El resto de código (lectura de fichero CSV, algoritmos principales, programa principal) está hecho en C++.

El problema del aprendizaje de pesos en características (APC) es un problema que optimiza el rendimiento del clasificador KNN mediante la asignación de valores reales (pesos) a las características ponderando de tal manera la relevancia de cada una de ellas en el problema de aprendizaje.
Dichos pesos vienen dados por un vector de valores reales en el intervalo [0,1] con un número de datos igual a la cantidad de atributos de las componentes.

El problema de clasificación consiste en el aprender a clasificar en distintas clases un conjunto de datos a partir de unos datos de entrenamiento (Train). El conjunto de datos que pretendemos clasificar es el conjunto de datos Test, con el cual podremos sacar más tarde la tasa de acierto del algoritmo utilizado para aprender los pesos de las características.
El objetivo principal del problema es ajustar el conjunto de ponderaciones o pesos asociados al conjunto total de elementos, con el fin de que los clasificadores que se construyan a partir de él sean óptimos.

Como clasificador KNN usaremos el 1NN para calcular la clase de un elemento a a partir del vecino más cercano b, considerando como vecino más cercano al que tenga los valores más próximos para cada uno de los atributos.

2. Código en común

Comenzamos con la descripción del código común a todos los algoritmos considerados en el problema.
Para comenzar, podemos decir que mi código consta de una clase Algortimos.cpp donde se encuentran todos los algoritmos (valga la redundancia) que se piden y demás métodos útiles creados para la realización de la práctica.
La representación de los datos es una simple matriz (matrizDatos) la cual contiene los valores numéricos leídos de los ficheros CSV y, un vector de string (etiquetas) que contiene la clase original de cada una de las características de la matriz, también leídas desde los ficheros CSV.
A continuación describo los métodos usados en común:

2.1. Generador de soluciones aleatorias

No tengo un método en sí para la generación de soluciones aleatorias, simplemente he hecho el siguiente código cada vez que lo he necesitado.

INICIO

Desde i = 0 hasta cantidadAtributos
num = generarNumeroAleatorio;
normalizar(num);
solucionInicial.insertar(num);
FIN(2)

FIN(1)

2.2. Normalización de datos

Una vez leídos los datos de los ficheros arff, estos han de ser normalizados dentro del rango [0,1] para que no halla unos atributos con mucho más peso o importancia que otros.

Max_j = máximo valor del atributo j para todas las características.
Min_j = mínimo valor del atributo j para todas las características.
x_ij = atributo j de la característica i

Para la normalización:

$x_ij = (x_ij$ - $Min_j)$ / $(Max_j$ - $Min_j)$

2.3. Torneo binario

Método utilizado para elegir los N mejores padres de una población, de manera que escogemos durante X iteraciones 2 padres de forma aleatoria y, fijándonos en sus tasas, cogemos el que tenga la mayor para añadirlo al vector de "Mejores padres".

```
INICIO
    Desde i = 0 hasta cantidadPadresAGenerar
        padre1 = escogerPadreAleatorio;
        padre2 = escogerPadreAleatorio;
        Mientras (padre1 == padre2)
            padre2 = escogerPadreAleatorio;
        si(tasa(padre1) >tasa(padre2))
            padresGanadores.insertar(padre1);
        sino
            padresGanadores.insertar(padre2);

    Devolver padresGanadores;

FIN(1)
```

2.4. Operador de cruce BLX

Operador para los cruces entre padres en los algoritmos genéticos.
Este operador calcula los valores máximo y mínimo entre los dos padres para, posteriormente junto con el valor I y el el valor α, crear el intervalo de generación de valores aleatorios para obtener los hijos.

INICIO

MAX_{p1} = máximo elemento del padre 1
MAX_{p2} = máximo elemento del padre 2

MIN_{p1} = mínimo elemento del padre 1
MIN_{p2} = mínimo elemento del padre 2

MAX_{total} = máximo entre MAX_{p1} y MAX_{p2}
MIN_{total} = mínimo entre MIN_{p1} y MIN_{p2}

$\alpha = 0.3$;

I = MAX_{total} - MIN_{total};

//Creamos los hijos en el intervalo siguiente:

//[MIN_{total}-I*α, MAX_{total}+I*α]
//En el código simplemente hacemos un bucle insertando cada
//uno de los valores del nuevo vector de pesos.

hijo1 = crearHijo(padre1, padre2, intervalo);
hijo2 = crearHijo(padre1, padre2, intervalo);

Devolver hijos;

FIN(1)

2.5. Operador de cruce aritmético

Operador para los cruces entre padres en los algoritmos genéticos.
Este operador, a diferencia del BLX, itera sobre cada uno de los atributos de los padres $P1_j$ y $P2_j$, "depositando" en el hijo la media aritmética entre esos dos valores.

Para cada valor de $P1_j$ y $P2_j$
    hijo.insertar(($P1_j$+$P2_j$) / 2);

2.6. Cálculo de la Distancia

Método común que se encarga de calcular la distancia Euclidea entre un vecino A y un vecino B teniendo en cuenta la dimensión del vector de pesos W.
En el código lo he desarrollado como un desenrollado de bucles con el fin de mejorar los tiempos de ejecución.

INICIO

distancia = 0;
si(tamaño de W es impar)
    distancia += distancia entre A_0 y B_0;
Para cada elemento de A y B
    distancia += distancia euclidea entre A_i y B_i;
    distancia += distancia euclidea entre A_{i+1} y B_{i+1};

FIN

2.7. Generar Vecino

La generación de un nuevo vecino consiste en calcular un número aleatorio (posModificar) entre 0 y el número de genes del individuo W (siendo los genes la cantidad de elementos que tiene el vector de pesos) y dicha posición será relaculada mediante un número aleatorio generado por una distribución normal en el rango [0,1]. Para cumplir el rango, en caso de que el valor generado sobrepase las cotas, simplemente truncamos para que las cumpla.

INICIO

```
numeroAleatorio = generarNumero;
normalizarNumAleatorio(rango[0,1]);
W[posModificar] = numAleatorio;
```

FIN

2.8. Creación de particiones

Para la generación de las particiones simplemente he creado las que se exigen en el guión, 5 particiones distintas generadas de forma aleatoria, las cuales son divididas en dos subparticiones: Train (datos de entrenamiento) y Test (datos de evaluación).

3. Principales algoritmos y métodos de búsqueda

3.1. Clasificador 1-NN

El objetivo del clasificador 1NN, tal como indica su nombre, es clasificar los elementos del conjunto de datos T_1 (Train, datos de entrenamiento), con respecto a los elementos del conjunto de datos T_2 (Test, los datos de evaluación), basándose siempre en el vecino más cercano, es decir, el que presente menor distancia Euclidea para cada uno de los elementos. De tal manera, un elemento i del conjunto de datos Train tomará el valor de la clase del vecino más cercano e del conjunto de datos Test. Tras clasificar el conjunto de datos Train al completo, calculamos la tasa de acierto teniendo en cuenta los datos verdaderos y la devolvemos.

```
INICIO

    Para cada elemento i de T1
        Para cada elemento e de T2
            si(i es distinto de e) //Leave one out
                distanciaVecino = calcularDistancia(i,e);
                si(distanciaVecino <minimaDistanciaActual)
                    vecinoMasCercano = e;
                    minimaDistanciaActual = distanciaVecino;
        FIN(3)
        si(clase(vecinoMasCercano) == clase(i))
            cantidadElementosBienClasificados++;
    FIN(2)
    //Calculamos la tasa de clasificación
    tasaClasificacion = 100*(cantidadElementosBienClasificados/Tamaño(T1))

Devolver tasaClasificacion;

FIN(1)
```

3.2. Local Search

El principal objetivo del algoritmo de búsqueda local es explotar un reducido espacio de búsqueda con el fin de obtener máximos locales (la mejor solución del entorno de la solución actual). Para ello recibe un vector inicial de pesos (W_{ini}) y a partir de este va generando nuevos vecinos (V_i) con el fin de encontrar el mejor en dicho espacio de búsqueda. Mediante la función de clasificación 1NN ya descrita comprobamos sucesivamente si cada uno de los vecinos generados se comporta mejor que la solución actual(S_{act}) mediante sus respectivas tasas de clasificacion: T_{act} y T_v, en dicho caso, la solución actual es actualizada.
La condición de parada consiste en generar en total 20*N vecinos (siendo i el vecino actual) o bien realizar 15000 evaluaciones con el clasificador 1NN.

INICIO

inicializar($S_{act} = W_{ini}$);
Mientras(i <20*N && i <15000)
posModificar = número aleatorio;
V_i = generarVecino(S_{act}, posModificar);
T_v = 1NN(V_i);

si($T_{act} < T_v$) //comparamos tasas
$S_{act} = V_i$;
$T_{act} = T_v$;

sino, si(hemos generado el vecindario de V_i entero)
//continuamos generando nuevos vecinos, poniendo el
//contador de posiciones modificadas a 0;

FIN(2)

FIN(1)

3.3. AGG-BLX

Entramos en el bloque de algoritmos genéticos con el AGG-BLX, el cual es genético generacional basado en el operador de cruce BLX. El objetivo principal de los algoritmos genéticos es, a partir de una población(P) de individuos(P_i) (soluciones iniciales) ir creando sucesivas generaciones de estos mediante operadores de cruce entre los individuos de las poblaciones actuales con el fin de obtener generaciones con mejores resultados a través de la evolución. Para escoger qué padres se cruzan utilizamos el torneo binario de modo que creamos una población de mejores padres a cruzar (PMP), los cuales serán los mejores. Además de los cruces entre individuos también se utiliza el operador de mutación para realizar pequeñas mejoras puntuales sobre individuos de manear azarosa.
Este en concreto, al ser generacional, a la hora de crear una nueva población, esta sustituirá al completo (salvo al mejor padre MP) a la población actual. Para evitar que el algoritmo retroceda en la búsqueda de la solución óptima, como ya he dicho, el mejor de los padres deberá estar presente en toda generación siempre y cuando sea mejor que el peor de los individuos de las sucesivas nuevas generaciones. Concretamente en este problema, se pide una población de individuos (tamaPoblacion) de tamaño 30, una probabilidad de cruce (probCruce) de 0.7 y una probabilidad de mutación (probMut) de 0.001. La condición de parada la situamos en 15000 evaluaciones con la función 1-NN. cantidadGenes es el parámetro que nos indica la cantidad de elementos de un vector de pesos (genes en un individuo)

INICIO

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)
        //calculamos el mejor de los padres (MP) fijándonos en el vector de tasas.
        MP = calcularIndiceMejorPadre( tasasPoblacionActual );

        //Aplicamos el torneo binario para obtener la población de mejores
        //padres (PMP)
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas
        PMP = torneoBinario(tasasPoblacionActual)
```

```
        //creamos la nueva generación de individuos a partir de los mejores padres
        Para (cada pareja (P_i,P_{i+1}) de padres calculada en el torneo binario)
            parHijos = operadorCruceBLX(P_i,P_{i+1});
            nuevaPoblacion.introducir(parHijos);

        //Rellenamos la poblacion con el resto de mejores padres
        //que no se hayan cruzado
        Mientras (no se haya completado la nueva población)
            nuevaPoblacion.introducir(PMP[i])
            //con el fin de detectar si se ha perdido al mejor padre:
            si(i == MP)
                mejorPadrePerdido = false;

        //Procedemos a realizar la mutación
        cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
        //como solo sale un individuo a mutar no aplicamos un bucle
        individuoAMutar = rand() %tamaPoblacion;
        genAMutar = rand %cantidadGenes;
        Poblacion[individuoAMutar] = generarVecino(P_[individuoAMutar])

        //Procedemos a evaluar a los individuos de la nueva población (P_nueva)
        //almacenando las tasas en un vector de tasas de la nueva población.
        Para cada elemento P_i de P_nueva
            tasasPoblacionNueva += 1-NN(P_i);

        //cogemos al mejor padre de la población anterior para sustituirlo
        //por el peor de la nueva generación de individuos.
        si(mejorPadrePerdido)
        P_nueva[peorIndividuo] = P_act[mejorIndividuo];

        //actualizamos la población actual
        P_act = P_nueva
        tasasPoblacionActual = tasasPoblacionNueva;

    FIN(2)

    //Calculamos el mejor de los individuos de la población final obtenida
    MP = calcularIndiceMejorPadre( tasasPoblacionActual );

    devolver MP;

FIN(1)
```

3.4. AGG-CA

Continuamos con los algoritmos genéticos generacionales pero esta vez, cambiamos el operador de cruce BLX por el CA, de manera que ahora dos padres generarán tan solo un hijo, el cual surje mediante la media aritmética de los padres. El resto del algoritmo es completamente igual que el anteriormente descrito. Ahora en lugar de escoger a los 30 mejores padres como en el algoritmo anterior, escogemos a los 60 mejores padres, por el hecho ya comentado de que dos padres generan un solo hijo.

INICIO

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)
        //calculamos el mejor de los padres (MP) fijándonos en el vector de tasas.
        MP = calcularIndiceMejorPadre( tasasPoblacionActual );

        //Aplicamos el torneo binario para obtener la población de mejores
        //padres (PMP)
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas. Esta vez cogemos 60 mejores padres.
        PMP = torneoBinario(tasasPoblacionActual)

        //creamos la nueva generación de individuos a partir de los mejores padres
        Para (cada pareja (P_i,P_i+1) de padres calculada en el torneo binario)
                nuevoHijo = operadorCruceBLX(P_i,P_i+1);
                nuevaPoblacion.introducir(nuevoHijo);

        //Rellenamos la poblacion con el resto de mejores padres
        //que no se hayan cruzado
        Mientras (no se haya completado la nueva población)
                nuevaPoblacion.introducir(PMP[i])
                //con el fin de detectar si se ha perdido al mejor padre:
                si(i == MP)
                        mejorPadrePerdido = false;
```

//Procedemos a realizar la mutación
cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
//como solo sale un individuo a mutar no aplicamos un bucle
individuoAMutar = rand() %tamaPoblacion;
genAMutar = rand %cantidadGenes;
Poblacion[individuoAMutar] = generarVecino($P_{[individuoAMutar]}$)

//Procedemos a evaluar a los individuos de la nueva población (P_{nueva})
//almacenando las tasas en un vector de tasas de la nueva población.
Para cada elemento P_i de P_{nueva}
    tasasPoblacionNueva += 1-NN(P_i);

//cogemos al mejor padre de la población anterior para sustituirlo
//por el peor de la nueva generación de individuos.
si(mejorPadrePerdido)
P_{nueva}[peorIndividuo] = P_{act}[mejorIndividuo];

//actualizamos la población actual
P_{act} = P_{nueva}
tasasPoblacionActual = tasasPoblacionNueva;

FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre(tasasPoblacionActual);

devolver MP;

FIN(1)

3.5. AGE-BLX

Cambiamos de tipo de algoritmo genético, pasando a los estacionarios, los cuales se basan en cambiar tan solo 2 individuos de la población actual por nuevos individuos (no como en los generacionales que sustituíamos a la poblacióni entera), solo en caso de que estos nuevos sean mejores que los dos peores de la población actual. Concretamente en este usaremos el operador de cruce BLX ya comentado anteriormente para generar nuevos individuos a partir de los mejores padres.

INICIO

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)
        //calculamos los dos peores padres PP_1 y PP_2
        PP_1 = calcularIndicePeorPadre1
        PP_2 = calcularIndicePeorPadre2

        //Aplicamos el torneo binario para obtener a los dos mejores
        //padres MP_1 y MP_2
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas
        torneoBinario(tasasPoblacionActual, MP_1, MP_2)

        //Calculamos los dos nuevos hijos a partir de los dos mejores padres
        parHijos = operadorCruceBLX(MP_1, MP_2);

        //Comprobamos si los hijos son mejores que los peores padres.
        //Para ello comparamos el mejor de los hijos(H_1) con el peor de los
        //dos peores padres(PP_2), así nos aseguramos de que al menos
        //halla un cambio.
        //(también comparamos el peor de los dos hijos con el mejor de los
        //dos peores padres).
        si (tasa(H_1) >tasa(PP_2))
                poblacion[PP_2] = H_1
```

si (tasa(H_2) >tasa(PP_1))
poblacion[PP_1] = H_2

//Procedemos a realizar la mutación
cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
Desde 0 hasta cantidadMutaciones
individuoAMutar = rand() %tamaPoblacion;
genAMutar = rand %cantidadGenes;
Poblacion[individuoAMutar] = generarVecino($P_{[individuoAMutar]}$)
//Evaluamos las tasas de los nuevos individuos(P_{nueva})
tasasPoblacionActual[individuoAMutar] = 1-NN(IndividuoAMutar);
FIN(3)
FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre(tasasPoblacionActual);

devolver MP;

FIN(1)

3.6. AGE-CA

Como segundo algoritmo estacionario, de la misma manera que para los generacionales, tenemos el correspondiente algoritmo genético con operador de cruce, el cual hace uso de cuatro mejores padres (calculados con el torneo binario) para generar dos nuevos hijos y proceder exactamente de la misma forma que en el anterior algoritmo estacionario a partir de ese momento.

INICIO

//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
    GenerarIndividuo(rand);

//Procedemos a evaluar a los individuos de la población actual (P_{act})
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_{act}
    tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)
    //calculamos los dos peores padres PP_1 y PP_2
    PP_1 = calcularIndicePeorPadre1
    PP_2 = calcularIndicePeorPadre2

    //Aplicamos el torneo binario para obtener a los cuatro mejores
    //padres esta vez MP_1, MP_2, MP_3 y MP_4
    //con torneos entre padres escogidos de forma aleatoria y teniendo
    //en cuenta las tasas
    torneoBinario(tasasPoblacionActual, MP_1, MP_2)

    //Calculamos los dos nuevos hijos a partir de los cuatro mejores padres
    parHijos = operadorCruceBLX(MP_1, MP_2, MP_3, MP_4);

    //Comprobamos si los hijos son mejores que los peores padres.
    //Para ello comparamos el mejor de los hijos(H_1) con el peor de los
    //dos peores padres(PP_2), así nos aseguramos de que al
    //menos halla un cambio.
    //(también comparamos el peor de los dos hijos con el mejor de los
    //dos peores padres).
    si (tasa(H_1) >tasa(PP_2))
        poblacion[PP_2] = H_1
    si (tasa(H_2) >tasa(PP_1))

poblacion[PP_1] = H_2

//Procedemos a realizar la mutación
cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
Desde 0 hasta cantidadMutaciones
individuoAMutar = rand() %tamaPoblacion;
genAMutar = rand %cantidadGenes;
Poblacion[individuoAMutar] = generarVecino($P_{[individuoAMutar]}$)
//Evaluamos las tasas de los nuevos individuos(P_{nueva})
tasasPoblacionActual[individuoAMutar] = 1-NN(IndividuoAMutar);
FIN(3)
FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre(tasasPoblacionActual);

devolver MP;

FIN(1)

3.7. AMCA-1.0

Empezamos con el bloque de algoritmos meméticos, los cuales son básicamente algoritmos de exploración (genéticos) y explotación (local search) al mismo tiempo. He decidido realizar los algoritmos meméticos basándome en el operador de cruce aritmético, no por algo en especial, simplemente los tanto los tiempos de ejecución como las tasas de acierto son prácticamente identicos y al final me he decidido por el operador CA. Comenzamos con el algoritmo memético con probabilidad de aplicación de la local search PLS = 1, es decir, le aplicamos la local search a todos y cada uno de los individuos de la población cada 10 generaciones tal como se indica en el guión para conseguir mejores resultados mediante la explotación.

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)

        //llevamos la cuenta del número de generación
        generacion++;

        //calculamos el mejor de los padres (MP) fijándonos en el vector de tasas.
        MP = calcularIndiceMejorPadre( tasasPoblacionActual );

        //Aplicamos el torneo binario para obtener la población de mejores
        //padres (PMP)
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas. Esta vez cogemos 60 mejores padres.
        PMP = torneoBinario(tasasPoblacionActual)

        //creamos la nueva generación de individuos a partir de los mejores padres
        Para (cada pareja (P_i,P_i+1) de padres calculada en el torneo binario)
                nuevoHijo = operadorCruceBLX(P_i,P_i+1);
                nuevaPoblacion.introducir(nuevoHijo);

        //Rellenamos la poblacion con el resto de mejores padres
        //que no se hayan cruzado
```

Mientras (no se haya completado la nueva población)
nuevaPoblacion.introducir(PMP[i])
//con el fin de detectar si se ha perdido al mejor padre:
si(i == MP)
mejorPadrePerdido = false;

//Procedemos a realizar la mutación
cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
//como solo sale un individuo a mutar no aplicamos un bucle
individuoAMutar = rand() %tamaPoblacion;
genAMutar = rand %cantidadGenes;
Poblacion[individuoAMutar] = generarVecino($P_{[individuoAMutar]}$)

//Procedemos a evaluar a los individuos de la nueva población (P_{nueva})
//almacenando las tasas en un vector de tasas de la nueva población.
Para cada elemento P_i de P_{nueva}
tasasPoblacionNueva += 1-NN(P_i);

//cada 10 generaciones realizamos explotación con la búsqueda local
//sobre la población entera
si(generacion == 10)
generacion = 0;
para cada individuo P_i de la población P
P_i = LocalSearch(P_i)
cantidadEvaluaciones1NN += llamadas desde localSearch;
FIN(3)

//cogemos al mejor padre de la población anterior para sustituirlo
//por el peor de la nueva generación de individuos.
si(mejorPadrePerdido)
P_{nueva}[peorIndividuo] = P_{act}[mejorIndividuo];

//actualizamos la población actual
P_{act} = P_{nueva}
tasasPoblacionActual = tasasPoblacionNueva;

FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre(tasasPoblacionActual);
devolver MP;
FIN(1)

3.8. AMCA-0.1

Esta vez la probabilidad de aplicación de la local search se reduce a 0.1, por lo que si la población es de 10 individuos solamente se le aplicara a un individuo de forma aleatoria. El resto del algoritmo es exactamente igual que el anterior.

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//nos creamos un vector con los indices de cada individuo de la poblacion
//para usarlo al decidir a quien se le aplica la localSearch
Mientras i=0 <tamaPoblacion
        vectorIndicesPoblacion.introducir(i);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)

        //llevamos la cuenta del número de generación
        generacion++;

        //calculamos el mejor de los padres (MP) fijándonos en el vector de tasas.
        MP = calcularIndiceMejorPadre( tasasPoblacionActual );

        //Aplicamos el torneo binario para obtener la población de mejores
        //padres (PMP)
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas. Esta vez cogemos 60 mejores padres.
        PMP = torneoBinario(tasasPoblacionActual)

        //creamos la nueva generación de individuos a partir de los mejores padres
        Para (cada pareja (P_i,P_i+1) de padres calculada en el torneo binario)
                nuevoHijo = operadorCruceBLX(P_i,P_i+1);
                nuevaPoblacion.introducir(nuevoHijo);

        //Rellenamos la poblacion con el resto de mejores padres
        //que no se hayan cruzado
        Mientras (no se haya completado la nueva población)
```

nuevaPoblacion.introducir(PMP[i])
//con el fin de detectar si se ha perdido al mejor padre:
si(i == MP)
mejorPadrePerdido = false;

//Procedemos a realizar la mutación
cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
//como solo sale un individuo a mutar no aplicamos un bucle
individuoAMutar = rand() %tamaPoblacion;
genAMutar = rand %cantidadGenes;
Poblacion[individuoAMutar] = generarVecino($P_{[individuoAMutar]}$)

//Procedemos a evaluar a los individuos de la nueva población (P_{nueva})
//almacenando las tasas en un vector de tasas de la nueva población.
Para cada elemento P_i de P_{nueva}
tasasPoblacionNueva += 1-NN(P_i);

//cada 10 generaciones realizamos explotación con la búsqueda local
//sobre el 10 % de la población. si(generacion == 10)
shuffle(vectorIndicesPoblacion); //para coger individuos aleatorios
generacion = 0;
para cada individuo P_i de la población P hasta llegar al 10 %
P_i = LocalSearch(P_i)
cantidadEvaluaciones1NN += llamadas desde localSearch;
FIN(3)

//cogemos al mejor padre de la población anterior para sustituirlo
//por el peor de la nueva generación de individuos.
si(mejorPadrePerdido)
P_{nueva}[peorIndividuo] = P_{act}[mejorIndividuo];

//actualizamos la población actual
P_{act} = P_{nueva}
tasasPoblacionActual = tasasPoblacionNueva;

FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre(tasasPoblacionActual);

devolver MP;

FIN(1)

3.9. AMCA-0.1 mej

Por último tenemos el memético con probabilidad de aplicación 0.1 pero que además, es aplicado al 10 % de individuos mejores de la población (en este caso tan solo uno).

```
//Inicializamos la población con individuos con genes aleatorios
//en el rango [0,1]
Desde 0 hasta tamaPoblacion
        GenerarIndividuo(rand);

//nos creamos un vector con los indices de cada individuo de la poblacion
//para usarlo al decidir a quien se le aplica la localSearch
Mientras i=0 <tamaPoblacion
        vectorIndicesPoblacion.introducir(i);

//Procedemos a evaluar a los individuos de la población actual (P_act)
//almacenando las tasas en un vector de tasas.
Para cada elemento P_i de P_act
        tasasPoblacionActual += 1-NN(P_i);

cantidadCruces = cantidadGenes / 2 * probCruce;
Mientras(cantidadEvaluaciones1NN <15000)

        //llevamos la cuenta del número de generación
        generacion++;

        //calculamos el mejor de los padres (MP) fijándonos en el vector de tasas.
        MP = calcularIndiceMejorPadre( tasasPoblacionActual );

        //Aplicamos el torneo binario para obtener la población de mejores
        //padres (PMP)
        //con torneos entre padres escogidos de forma aleatoria y teniendo
        //en cuenta las tasas. Esta vez cogemos 60 mejores padres.
        PMP = torneoBinario(tasasPoblacionActual)

        //creamos la nueva generación de individuos a partir de los mejores padres
        Para (cada pareja (P_i,P_i+1) de padres calculada en el torneo binario)
                nuevoHijo = operadorCruceBLX(P_i,P_i+1);
                nuevaPoblacion.introducir(nuevoHijo);

        //Rellenamos la poblacion con el resto de mejores padres
        //que no se hayan cruzado
        Mientras (no se haya completado la nueva población)
                nuevaPoblacion.introducir(PMP[i])
```

```
        //con el fin de detectar si se ha perdido al mejor padre:
        si(i == MP)
            mejorPadrePerdido = false;

    //Procedemos a realizar la mutación
    cantidadMutaciones = probMut * tamaPoblacion * cantidadGenes;
    //como solo sale un individuo a mutar no aplicamos un bucle
    individuoAMutar = rand() %tamaPoblacion;
    genAMutar = rand %cantidadGenes;
    Poblacion[individuoAMutar] = generarVecino(P_[individuoAMutar])

    //Procedemos a evaluar a los individuos de la nueva población (P_nueva)
    //almacenando las tasas en un vector de tasas de la nueva población.
    Para cada elemento P_i de P_nueva
        tasasPoblacionNueva += 1-NN(P_i);
        //para poder coger los mejores en la local search
        poblacionOrdenadaPorTasas.introducir(P_i, tasa(P_i));

    //cada 10 generaciones realizamos explotación con la búsqueda local
    //sobre el 10 % de la población.          si(generacion == 10)
        shuffle(vectorIndicesPoblacion); //para coger individuos aleatorios
        generacion = 0;
        para cada individuo P_i de la población ordenada por tasas
        P hasta llegar al 10 %
            P_i = LocalSearch(P_i)
            cantidadEvaluaciones1NN += llamadas desde localSearch;
        FIN(3)

    //cogemos al mejor padre de la población anterior para sustituirlo
    //por el peor de la nueva generación de individuos.
    si(mejorPadrePerdido)
    P_nueva[peorIndividuo] = P_act[mejorIndividuo];

    //actualizamos la población actual
    P_act = P_nueva
    tasasPoblacionActual = tasasPoblacionNueva;

FIN(2)

//Calculamos el mejor de los individuos de la población final obtenida
MP = calcularIndiceMejorPadre( tasasPoblacionActual );

devolver MP;
```

FIN(1)

4. Algoritmo de comparación: RELIEF

Consiste en un algoritmo voraz de cálculo del vector de pesos para el APC, digamos, algoritmo de fuerza bruta. Partiendo de un vector de pesos inicializado a cero (W_{ini}), consiste en, para cada elemento T_i del conjunto de datos Train (T_1), calcular su amigo A_i y su enemigo E_i del subconjunto Test (T_2) para, en función de ellos, calcular la i-ésima posición del vector de pesos.

INICIO

W = inicializar(W_{ini});

Para cada elemento i de T_1
Para cada elemento e de T_2
si(i es distinto de e) //Leave one out
distanciaVecino = calcularDistancia(i,e);

si(calse(i) == clase(e)) //para los amigos
si(distanciaVecino <distanciaMinAmigo)
A_i = e;
distanciaMinAmigo = distanciaVecino;
sino //para los enemigos
si(distanciaVecino <distanciaMinEnemigo)
E_i = e;
distanciaMinEnemigo = distanciaVecino;
W[i] = actualizar peso teniendo en cuenta al amigo y al enemigo.
//en el código he creado un método aparte para actualizar W.

//Antes de devolver W lo normalizamos para que contenga valores en el
//intervalo [0,1].

devolver W;

FIN

5. Procedimiento considerado para desarrollar la práctica

La inmensa mayoría del código que entrego está desarrollado por mí mismo, exceptuando la lectura de archivos csv por ejemplo, para la cual me he ayudado un poco de internet, así como el uso de páginas como stackOverflow o cPlusPLus para consultar información sobre contenedores de la STL, generación de números aleatorios, uso de la biblioteca ctime para la toma de tiempos...
. No he utilizado nada de código del proporcionado en la web de la asignatura.
El procedimiento seguido para el desarrollo de la práctica es tal cual el que se indica en el guión: lectura de ficheros .arff, normalización de los datos y exportación a .csv en R, lectura de ficheros .csv en C++, generación de las 5 particiones más los datos para Train y para Test, generación del algoritmo 1NN, algoritmo RELIEF, Local Search, los AGG, los AGE y por último los tres algoritmos meméticos. Tras la realización de los algoritmos realizamos la recopilación de datos y posteriormente documentación y análisis.

6. Tablas de resultados de ejecución

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	93,6620	0,008729	81.739100	0.010123	78.846200	0.002310
Partición 1-2	95,4386	0,008753	85.652200	0.010236	81.730800	0.002268
Partición 2-1	95,0704	0,008689	82.608700	0.010216	85.576900	0.002337
Partición 2-2	94,3860	0,008738	83.478300	0.010168	86.538500	0.002266
Partición 3-1	97,1831	0,008671	81.304300	0.010102	85.576900	0.002267
Partición 3-2	94,7368	0,008665	85.217400	0.010170	82.692300	0.002285
Partición 4-1	96,1268	0,008680	83.043500	0.010130	84.615400	0.002277
Partición 4-2	94,7368	0,008650	83.913000	0.010110	80.769200	0.002272
Partición 5-1	95,0704	0,008654	85.652200	0.010140	85.576900	0.002277
Partición 5-2	97,1930	0,008688	84.782600	0.010130	80.769200	0.002290
Media	95,3604	0,008692	83.739100	0.010154	83.269200	0,002285

Figura 6.1: Tabla 1-NN

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	94,0141	00,0085	82,1739	00,0108	78.8462	0.002843
Partición 1-2	97,8947	00,0084	86,5217	00,0106	79.8077	0.002268
Partición 2-1	95,7747	00,0084	83,4783	00,0107	81.7308	0.002312
Partición 2-2	95,4386	00,0084	86,5217	00,0108	79.8077	0.002278
Partición 3-1	95,7747	00,0085	87,3913	00,0110	89.4231	0.0023
Partición 3-2	94,7368	00,0084	89,1304	00,0106	80.7692	0.002284
Partición 4-1	95,4225	00,0085	86,0870	00,0105	79.8077	0.002285
Partición 4-2	96,4912	00,0084	88,6957	00,0109	76.9231	0.002277
Partición 5-1	92,6056	00,0085	87,8261	00,0109	78.8462	0.002276
Partición 5-2	95,7895	00,0084	87,3913	00,0106	00,7500	0.002287
Media	95,3942	00,0084	86,5217	00,0107	80.0962	0.002341

Figura 6.2: Tabla RELIEF

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	95,0704	4,020820	90.434800	9.315020	88.461500	2.095600
Partición 1-2	97,5439	3,997430	90.434800	9.277700	92.307700	2.001780
Partición 2-1	96,1268	3,968910	90.869600	9.349250	88.461500	1.996210
Partición 2-2	97,1930	3,989290	89.565200	9.266010	86.538500	1.993370
Partición 3-1	98,2394	3,962020	93.043500	9.281260	91.346200	2.023930
Partición 3-2	95,7895	3,992570	90.434800	9.154810	84.615400	2.003700
Partición 4-1	97,5352	3,958320	87.826100	9.196520	93.269200	1.995130
Partición 4-2	95,0877	3,997460	92.173900	9.217660	87.500000	1.999080
Partición 5-1	95,7747	3,976240	88.260900	9.290630	87.500000	1.998730
Partición 5-2	96,1404	3,987330	84.782600	9.143110	90.384600	1.987760
Media	96,4501	3,985040	89.782600	9.249200	89.038500	2,009530

Figura 6.3: Tabla Local Search

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	94,3662	99,059100	82.173900	122.536	82.692300	24.906400
Partición 1-2	97,8947	99,742200	86.521700	122.311	88.461500	25.070300
Partición 2-1	94,3662	98,957500	82.608700	122.904	84.615400	24.965600
Partición 2-2	94,7368	99,671200	79.565200	122.345	88.461500	25.033800
Partición 3-1	96,8310	98,832400	85.652200	122.473	86.538500	25.152000
Partición 3-2	95,0877	99,616700	86.956500	122.536	85.576900	25.005000
Partición 4-1	97,5352	98,847700	83.043500	122.488	88.461500	24.909500
Partición 4-2	94,3860	99,641900	86.087000	122.493	79.807700	25.001000
Partición 5-1	95,0704	98,968000	86.521700	122.709	78.846200	24.968800
Partición 5-2	96,8421	99,480600	83.913000	122.897	77.884600	24.901900
Media	95,7116	99,281700	84.304300	122.569	84.134600	24,991400

Figura 6.4: Tabla AGGBLX

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	93,6620	98,955600	80.869600	122.322	78.846200	24.841100
Partición 1-2	96,4912	99,667500	84.347800	123.460	82.692300	25.010300
Partición 2-1	95,7747	98,860500	83.043500	124.228	85.576900	24.965000
Partición 2-2	95,4386	99,665800	81.304300	122.870	84.615400	24.971700
Partición 3-1	96,8310	98,827000	82.608700	123.837	87.500000	25.051900
Partición 3-2	94,3860	99,471000	86.956500	122.920	80.769200	24.960500
Partición 4-1	96,4789	98,765800	82.173900	122.707	85.576900	25.228300
Partición 4-2	94,7368	99,565700	86.087000	123.120	81.730800	24.951200
Partición 5-1	95,0704	98,883600	85.652200	123.904	83.653800	24.971300
Partición 5-2	98,2456	99,681800	87.826100	122.151	80.769200	25.026900
Media	95,7115	99,234400	84.087000	123.152	83.173100	24,997800

Figura 6.5: Tabla AGGCA

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	95,0704	296,4840	83.9130	241.3960	83.6538	49.7915
Partición 1-2	97,8947	298,7700	84.3478	241.2020	79.8077	50.0069
Partición 2-1	94,3662	296,6670	86.0870	241.5760	84.6154	49.8045
Partición 2-2	95,7895	299,1560	82.6087	241.3420	87.5000	49.8509
Partición 3-1	95,4225	296,7130	84.3478	242.0340	85.5769	50.0832
Partición 3-2	95,4386	298,3010	80.4348	242.3000	79.8077	49.8468
Partición 4-1	96,4789	296,1050	85.2174	240.6280	87.5000	49.6726
Partición 4-2	95,0877	298,5500	80.8696	240.7160	80.7692	49.9266
Partición 5-1	96,1268	296,4970	85.2174	240.9140	87.5000	49.8497
Partición 5-2	98,5965	298,3530	82.1739	240.9930	81.7308	49.7682
Media	96,0272	297,5600	83.5217	241.3100	83.8462	49,8601

Figura 6.6: Tabla AGEBLX

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	93,3099	296,3960	83.0435	241.9290	81.7308	49.6308
Partición 1-2	96,4912	298,7730	86.0870	243.3150	81.7308	49.8876
Partición 2-1	96,1268	296,5770	85.6522	244.3040	85.5769	49.7380
Partición 2-2	94,7368	298,3780	86.5217	242.4370	87.5000	49.7689
Partición 3-1	94,7183	295,9770	84.7826	246.3750	87.5000	50.0676
Partición 3-2	94,7368	298,1760	03.9130	242.1950	80.7692	49.7035
Partición 4-1	96,4789	296,0950	85.2174	242.3550	89.4231	49.7043
Partición 4-2	95,0877	298,8290	87.3913	242.9550	81.7308	49.8161
Partición 5-1	92,6056	296,6370	83.9130	245.1730	83.6538	49.7340
Partición 5-2	97,8947	298,3510	87.3913	241.6310	78.8462	49.6533
Media	95,2187	297,4190	85.3913	243.2670	83.8462	49,7704

Figura 6.7: Tabla AGECA

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	94.0141	99.4795	82,6087	121,1550	80,7692	25,9665
Partición 1-2	96.4912	99.8684	85,6522	121,0940	83,6538	26,1925
Partición 2-1	94.7183	99.0657	83,4783	122,8490	83,6538	26,1684
Partición 2-2	95.7895	99.8204	83,4783	120,8910	87,5000	26,1436
Partición 3-1	94.7183	99.1551	85,2174	122,1400	88,4615	26,2537
Partición 3-2	94,3860	100,2170	86,0870	120,5850	80,7692	26,1123
Partición 4-1	94.7183	99.4272	83,9130	120,4680	87,5000	26,0284
Partición 4-2	94,3860	100,3030	84,7826	120,5490	78,8462	26,1543
Partición 5-1	93,6620	99.5357	86,9565	120,9640	85,5769	26,0778
Partición 5-2	97.8947	100.32	85,6522	120,1380	82,6923	26,0793
Media	95.0778	99.7191	84,7826	121,0830	83,9423	26,1177

Figura 6.8: Tabla AMCA1

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	94.7183	99.6841	83,9130	121,1920	77,8846	24,8819
Partición 1-2	96.8421	100,4570	83,4783	122,1750	81,7308	25,0304
Partición 2-1	94.3662	100,2370	84,3478	123,1650	85,5769	24,9416
Partición 2-2	94.7368	100,7460	80,8696	121,4970	85,5769	25,0402
Partición 3-1	97.5352	99.5793	85,2174	121,7940	87,5000	25,0960
Partición 3-2	94,3860	100,6380	86,5217	121,2990	81,7308	24,9592
Partición 4-1	95.7747	99.3254	82,6087	121,1030	88,4615	24,9190
Partición 4-2	95.0877	100,6470	86,0870	121,7730	81,7308	24,9958
Partición 5-1	94.0141	99.9037	85,6522	122,2220	83,6538	24,9826
Partición 5-2	98.5965	100,1830	86,0870	121,7840	84,6154	24,9162
Media	95.6058	100.14	84,4783	121,8010	83,8462	24,9763

Figura 6.9: Tabla AMCA2

	Wdbc		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
Partición 1-1	93,6620	100,6170	83,4783	121,0120	82,6923	24,8985
Partición 1-2	96.1404	100,5120	86,0870	122,7560	81,7308	25,1074
Partición 2-1	94.7183	100,0020	82,6087	122,7350	85,5769	24,9629
Partición 2-2	94.7368	100.29	83,0435	121,2460	86,5385	25,0210
Partición 3-1	96,8310	99,5390	86,0870	121,8140	86,5385	25,1515
Partición 3-2	94,3860	100,5050	85,6522	121,2360	85,5769	25,0037
Partición 4-1	95.0704	99.8879	83,4783	121,0560	84,6154	24,9261
Partición 4-2	94.0351	101,0320	83,0435	121,7700	80,7692	25,0426
Partición 5-1	95.4225	100,1020	82,1739	122,7710	82,6923	24,9701
Partición 5-2	97,1930	100,0860	83,9130	120,6060	79,8077	24,9418
Media	95.2195	100,2570	83,9565	121,7000	83,6538	25,0026

Figura 6.10: Tabla AMCA3

	WDBC		Spambase		Sonar	
	%_clas	T	*%_clas*	T	*%_clas*	T
1-NN	95.3604	0.0086917	83.7391	0.0101537	83.2692	0.0022849
RELIEF	95.1845	0.0086738	87.5217	0.0101428	83.0769	0.0022921
BL	96.4501	3.98504	89.7826	9.2492	89.0385	2.00953
AGG-BLX	95.7116	99.2817	84.3043	122,5690	84.1346	24.9914
AGG-CA	95.7115	99.2344	84,0870	123,15200	83.1731	24.9978
AGE-BLX	96.0272	297.56	83.5217	241.31	83.8462	49.8601
AGE-CA	95.2187	297,4190	85.3913	243,2670	83.8462	49.7704
AM-(10,1.0)	95.0778	99.7191	84,7826	121,0830	83,9423	26,1177
AM-(10,0.1)	95.6058	100.14	84,4783	121,8010	83,8462	24,9763
AM-(10,0.1mej)	95.2195	100,2570	83,9565	121,7000	83,6538	25,0026

Figura 6.11: Tabla final de medias

7. Análisis de resultados obtenidos.

A primera una vista de la tabla 6.11, parece que en la relación tasa de acierto/tiempo de ejecución, el algoritmo que sale ganando por encima de los demás es la búsqueda local con su carácter de explotación del espacio de búsqueda.
En cuanto al tiempo de ejecución, podemos decir que tiene ventaja sobre los algoritmos genéticos y meméticos en el hecho de que no ha de realizar tantas cuentas pesadas y sucesivos accesos a contenedores de datos para obtener por ejemplo el mejor de los padres de una población (como se hace en los algoritmos genéticos generacionales para conservar el mejor de los padres), o para realizar el torneo binario para la selección de los mejores padres en cada iteración de dichos algoritmos, o bien para obtener nuevas generaciones de individuos con el fin de encontrar la solución óptima. Estas son, como he dicho, operaciones costosas en tiempo que hacen que, aun consiguiendo tasas de clasificación similares a los de la búsqueda local, den peores resultados a la hora de tener en cuenta el tiempo de ejecución de cada algoritmo.
Continuando con el análisis de tiempos, la búsqueda local es únicamente superada (con creces) por los algoritmos 1-NN y RELIEF; es evidente el por qué de este hecho, pues en dichos algoritmos no se realizan ni de lejos la cantidad de operaciones que realiza la búsqueda local en su código, de hecho, es lógico, pues la búsqueda local hace uso del propio 1-NN, calculando la tasa de clasificación de los 20*N vecinos generados.

Los tiempos de ejecución obtenidos por lo general son resultados coherentes (salvo los algoritmos genéticos estacionarios, los cuales pienso que deberían ser más rápidos que los generacionales, pues en estos últimos se realizan cambios de la población entera en lugar de cambiar tan solo a los dos peores padres por los nuevos hijos como hacen los estacionarios, pero no encuentro el cuello de botella en el código), cosa que no ocurre exactamente con las tasas de clasificación, me explico:
Pienso que los resultados obtenidos (en cuanto a las tasas me refiero) dependen en gran medida de las particiones de datos obtenidas de las bases de datos, quizás por la semilla escogida para ejecutar los algoritmos, o porque ha dado la casualidad de que han salido particiones favorables para el algoritmo BL, puesto que me parece extraño que sea la búsqueda local la que mejores resultados obtenga, ya que por ejemplo, el algoritmo memético con probabilidad de mutación 1.0 sobre los mejores individuos de la población tiene mejor pinta en la teoría debido a la mezcla de los mejores padres de cada generación a la explotación de los mejores individuos.

7.1. Comparación de las distintas metaheurístias entre sí y con los algoritmos RELIEF y 1NN.

- **Algoritmos genéticos** : Todos los algoritmos obtienen tasas de clasificación casi idénticas, 1 % o 1 % abajo todas obtienen un porcentaje de acierto en la clasificación similar, aunque un poco por debajo del algoritmo RELIEF, el cual utiliza al amigo y al enemigo para saber hacia que solución le conviene aproximarse y de cual alejarse.
  Todos están a la misma altura del algoritmo 1-NN con respecto a las tasas de clasificación.

- **Algoritmos meméticos**: de la misma forma que los genéticos, estos también son prácticamente idénticos entre sí con respecto a las tasas de clasificación. Quizás en tiempo de ejecución es donde varían un poco más pero, casi nada, apenas un segundo entre ellos. Exceptuando al RELIEF en la Spambase, estos algoritmos coinciden casi a la perfección en las tasas de clasificación tanto con RELIEF como con 1-NN.

Como vemos, con los resultados obtenidos poca diferencia a la hora de medir las tasas puede apreciarse entre los algoritmos desarrollados. Sí puede decirse que entre los dos algoritmos a comparar (1-NN frente a RELIEF) sale mejor el segundo tanto en tasa de acierto como en tiempo de ejecución, seguramente debido a lo ya comentado previamente de que el RELIEF se basa en amigo y enemigo para decidirse sobre el mejor vector de pesos, al contrario que el 1-NN que solo lo hace sobre el vecino más cercano.
En definitiva, la calidad de los algoritmos esta prácticamente nivelada entre unos y otros y, como he dicho al principio del análisis, si tuviese que basarme en estas tablas comparativas entre algoritmos, escogería la búsqueda local porque, sea el azar de la elección de los datos o la capacidad de explotación del espacio de búsqueda local que he implementado, es el que mejor resultados obtiene con diferencia tanto en tiempo (con respecto a los genéticos y meméticos) como en tasas de acierto (con respecto a todos).