

Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características

Alejandro Borrego Megías¹

07/04/2022

¹26504975M, alejbormeg@correo.ugr.es, Grupo 3 Martes 17:30-19:30

Contents

1	Descripción del problema APC.	2
2	Descripción de la aplicación de los algoritmos.	3
	Métricas	3
	Validación Cruzada	3
	Clasificador KNN	4
	Bases de datos a usar	5
	Detalles técnicos	5
3	Descripción de los Métodos de Búsqueda	7
	Algoritmo RELIEF	7
	Algoritmo de Búsqueda Local	9
	Algoritmo Genéticos	11
	Elementos comunes a todos los AG	11
	Algoritmo Genético Generacional (AGG)	11
	Algoritmo Genético Estacionario (AGE)	16
	Algoritmo Memético	19
4	Procedimiento	22
	Manual de uso	22
5	Experimentos y análisis de resultados	23
	Casos del problema empleados y parámetros utilizados.	23
	Resultados Obtenidos	23
6	Referencias Bibliográficas	31

Chapter 1

Descripción del problema APC.

El problema que trataremos es el APC (Aprendizaje de Pesos en Características). Antes de explicar el problema tendremos en cuenta los siguientes puntos:

- Conjunto de datos: Utilizaremos tres bases de datos que se explicarán más adelante y se representarán como una sucesión de elementos sobre los cuales se han llevado a cabo una serie de mediciones o se aporta una serie de características y tienen asociada una etiqueta. Por lo tanto estamos en el ámbito de problemas de Aprendizaje Supervisado.
- Clasificador: Utilizaremos el clasificador de los K vecinos más cercanos con $K=1$ para clasificar ejemplos, de esta manera la etiqueta que asignaremos al ejemplo que clasificamos será la del elemento más cercano a él en el conjunto de datos del que disponemos.
- Distancia: Para medir la distancia entre elementos del conjunto usaremos la distancia euclídea ponderada:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

Como podemos ver, cada elemento del sumatorio viene multiplicado por una constante w_i .

De esta manera, establecido el contexto del problema, nuestro objetivo será hallar ese vector w de pesos que dará prioridad a unos atributos sobre otros, para ello usaremos diversas técnicas aprendidas en clase.

Por lo tanto, el APC es un problema que pretende optimizar el rendimiento del clasificador KNN.

Chapter 2

Descripción de la aplicación de los algoritmos.

Métricas

En todos los algoritmos que empleemos vamos a utilizar un clasificador 1-NN (como se dijo anteriormente). De esta forma las métricas que usaremos para comprobar la bondad del vector de pesos w obtenido por cada algoritmo serán:

- **Tasa de clasificación:** que medirá el porcentaje de instancias correctamente clasificadas aplicando validación sobre el conjunto T de datos correspondiente (lo haremos generalmente sobre el conjunto de datos que usamos para entrenar). Su expresión es la siguiente:

$$tasa_{clas} = 100 \cdot \frac{n \text{ instancias bien clasificadas en } T}{n \text{ instancias en } T}$$

- **Tasa de reducción:** medirá el porcentaje de características descartadas, es decir, aquellas cuyo peso esté cercano a 0. Pondremos el umbral en 0.1. Su expresión es la siguiente:

$$tasa_{red} = 100 \cdot \frac{n \text{ valores } < 0.1 \text{ en } W}{n \text{ características}}$$

Finalmente, las dos métricas anteriores se combinarán en la que será nuestra **función objetivo**, que intentaremos maximizar, y que tiene como expresión la siguiente:

$$F(W) = \alpha \cdot tasa_{clas}(W) + (1 - \alpha) \cdot tasa_{red}(W)$$

En nuestro caso $\alpha = 0.5$ de manera que daremos la misma importancia a la tarea de reducción como la de clasificación.

Validación Cruzada

Por otro lado, como sabemos de las técnicas de Aprendizaje Automático, para diseñar un clasificador, es necesario realizar dos tareas: **Aprendizaje y Validación**.

De esta forma, el conjunto de los ejemplos que tratamos se divide en dos, en conjunto de entrenamiento (que usaremos para entrenar el clasificador) y en el conjunto de Validación (que lo usaremos para validarlo).

Para mayor seguridad y garantías de generalización, se suelen realizar varias particiones Entrenamiento-Validación sobre los datos originales, es por ello que nosotros usaremos la técnica denominada **Cross Validation 5-Fold**, que consiste en los siguiente:

- El conjunto de datos se divide en 5 partes (5-fold) disjuntas al 20% con distribución de clases equilibrada.
- Aprendemos un clasificador con el 80% de los datos (es decir, con 4 de las 5 particiones) y se valida en el 20% restante (la partición que no hemos usado). Esto lo realizaremos 5 veces cambiando cada vez el conjunto de validación.
- De esta manera obtenemos 5 valores distintos de porcentaje de clasificación, por lo que para medir la calidad del método se realizará la media de los 5 obtenidos.
- Lo explicado anteriormente es el método general, pero en nuestro caso en cada iteración obtendremos un resultado de $tasa_{clas}$, $tasa_{red}$, del tiempo de ejecución del algoritmo y de la función objetivo. Por lo que nosotros al final promediaremos todas estas cantidades para obtener la bondad de nuestro clasificador final.

Clasificador KNN

Finalmente, como ya se ha comentado, usaremos el **clasificador KNN** con $K=1$, por lo que en esta última parte de la sección vamos a explicar algunos detalles del clasificador.

El proceso de aprendizaje consiste en mantener en memoria una tabla con los ejemplos de entrenamiento junto con la clase que tienen asociada, de esta forma, dado un nuevo ejemplo se calculará la distancia con los n elementos de la tabla y se escogen los k elementos más cercanos, de esta forma la clase que asignaremos al nuevo ejemplo sería la mayoritaria entre estas k clases. En nuestro caso como $K=1$ se simplifica el procesp, pues asignamos la clase del elemento más cercano.

La descripción en Pseudocódigo sería:

Clasificador1NN(datos, elemento):

cmin=clase del primer elemento en entrenamiento e1
dmin=distancia entre e1 y el nuevo ejemplo

Para i=2 hasta m hacer:

Se calcula la distancia entre ei y el nuevo ejemplo.

Se comprueba si la nueva distancia es menor que dmin:

- Si es menor se elige cmin=clase de ei y se actualiza dmin.
- Si no es menor se continúa.

Devolver cmin.

Por otro lado la distancia que usaremos para el clasificador será la comentada en el apartado anterior, que denominaremos **distancia euclídea ponderada**, recordamos su expresión:

$$d(e_1, e_2) = \sqrt{\sum_i w_i (e_1^i - e_2^i)^2}$$

Finalmente usaremos dos técnicas diferentes para comprobar la precisión de nuestro clasificador según nos encontremos en la fase de entrenamiento o la de validación en cada partición realizada en el 5-Fold Cross Validation:

- Entrenamiento: Usaremos el método **Leave One Out**, su pseudocódigo es el siguiente:

LeaveOneOut(Entrenamiento):

Para cada elemento e en Entrenamiento:

etiqueta_predicha=Clasificador1NN(Entrenamiento-{e},e)

Si etiqueta_predicha=etiqueta_real de e

```

    correctos ++

    return 100*(correctos/num_datos)

```

Es decir, en cada paso eliminamos un elemento del conjunto de datos de entrenamiento y tratamos de clasificarlo, si las etiquetas predicha y real coinciden sumamos un acierto, si no se continúa. Finalmente se devuelve el porcentaje de acierto.

- Validación: En este caso método será intentar clasificar cada elemento del conjunto de validación usando los del conjunto de entrenamiento:

```

Evaluacion(Entrenamiento, Validacion):
    Para cada elemento e en Validacion:
        etiqueta_predicha=Clasificador1NN(Entrenamiento,e)

        Si etiqueta_predicha=etiqueta_real de e
            correctos ++

    return 100*(correctos/num_datos)

```

Para este método es muy importante que los datos estén normalizados entre [0,1] para no priorizar unos atributos sobre otros. Por ello, para cualquier conjunto de datos que utilicemos, lo primero que se hará será normalizar los datos y mezclarlos (para evitar que las clases estén desbalanceadas). El algoritmo para normalizar es el siguiente:

```

Normalizar(Datos):
    Para cada atributo a:
        Se calcula el máximo y mínimo de entre todos los datos

    Para cada elemento del conjunto de datos:
        Cambiar valor de atributo a por (x-min)/(max-min)

```

Bases de datos a usar

Las bases de datos que utilizaremos son:

- **Ionosphere:** datos de radar recogidos por un sistema en Goose Bay, Labrador. Los objetivos eran electrones libres en la ionosfera. Los “buenos” retornos de radar son aquellos que muestran evidencia de algún tipo de estructura en la ionosfera. Los retornos “malos” son aquellos que no lo hacen.
Tiene 352 ejemplos con 34 atributos (señales procesadas) y 2 clases (good o bad).
- **Parkinsons:** contiene datos para distinguir entre la presencia y la ausencia de la enfermedad de Parkinson en una serie de pacientes a partir de medidas biomédicas de la voz. Tiene 195 ejemplos con 23 atributos (incluyendo la clase). Las clases están desbalanceadas (147 enfermos 48 sanos). Algunos atributos son: Frecuencia mínima, máxima y media de la voz, medidas absolutas y porcentuales de variación de la voz, medidas de ratio de ruido en las componentes tonales.
- **Spectf-heart:** contiene atributos calculados a partir de imágenes médicas de tomografía computerizada (SPECT) del corazón de pacientes humanos. La tarea consiste en determinar si la fisiología del corazón analizado es correcta o no. Tiene 267 ejemplos, 45 atributos (incluyendo la clase) y 2 clases (sano o con patología).

Detalles técnicos

En nuestro problema, el conjunto de datos se va a representar por medio de un vector de pares (atributos, etiqueta).

El esquema de representación de soluciones será un vector de doubles que representará el vector de pesos w solución, de manera que si una característica tiene asociado un peso de 1, significa que se considera completamente en el cálculo de la distancia, si tiene un 0.1 o menos no se considera y para cualquier otro valor intermedio pondera la importancia del atributo que tiene asociado.

Chapter 3

Descripción de los Métodos de Búsqueda

En esta sección vamos a describir las distintas técnicas que emplearemos para resolver el problema APC.

Algoritmo RELIEF

Se trata de un algoritmo Voraz (greedy) de búsqueda secuencial, en el cual se parte de un vector de pesos inicializado a 0 que incrementa cada componente en función del enemigo más cercano a cada ejemplo y la reduce en función del amigo más cercano a cada ejemplo.

Entendemos el enemigo más cercano como el elemento más próximo al que estemos considerando que tiene asociada una clase distinta a la suya. Del mismo modo consideramos el amigo más cercano como aquel que es más próximo al elemento que estamos considerando y tiene misma clase asociada.

La implementación del algoritmo es la siguiente en pseudocódigo:

```
RELIEF(Entrenamiento,w):  
    Inicializamos w a 0  
  
    Para cada elemento e en el conjunto de entrenamiento E:  
        se calcula distancia componente a componente al enemigo más cercano de e en E  
        se calcula distancia componente a componente al amigo más cercano de e en E  
        se calcula  $w=w+distancia\_enemigo-distancia\_amigo$  componente a componente  
  
    Normalizamos w  
  
    return w
```

Por su lado las implementaciones para las funciones que calculan las distancias al enemigo y amigo más cercano son las siguientes:

```
DistanciaEnemigoMasCercano(Entrenamiento,a):  
    elemento de entrada a  
    dmin=distancia euclídea entre a y el primer elemento de entrenamiento  
    distancia=0  
  
    Para cada elemento e en entrenamiento E:  
        distancia=distancia euclídea entre e y a
```



```
Si distancia < dmin && clase de e!= clase de a
    dmin=distancia
    enemigo_mas_cercano=e
```

Calculamos distancias componente a componente entre a y enemigo_mas_cercano

```
return distancias_componente_componente
```

Para el amigo más cercano sería:

DistanciaAmigoMasCercano(Entrenamiento,a):

elemento de entrada a

dmin=distancia euclídea entre a y el primer elemento de entrenamiento

distancia=0

Para cada elemento e en entrenamiento E:

distancia=distancia euclídea entre e y a

```
Si distancia < dmin && clase de e== clase de a
```

```
    dmin=distancia
```

```
    amigo_mas_cercano=e
```

Calculamos distancias componente a componente entre a y amigo_mas_cercano

```
return distancias_componente_componente
```

En el código, la estructura de datos que hemos empleado han sido los vectores de la STL. Además hemos tenido que sobrecargar los operadores de suma y resta para realizar sumas y restas componente a componente entre dos vectores.

Algoritmo de Búsqueda Local

En segundo lugar hemos empleado la técnica de búsqueda por trayectorias simples para implementar el algoritmo de **búsqueda local del primer mejor**.

Antes de describir el algoritmo vamos a aclarar algunos puntos necesarios para su comprensión:

- Definimos el entorno de una solución w como el conjunto formado por las soluciones accesibles desde ella a través de un movimiento, que en nuestro caso será una mutación de una componente del vector w por medio de un valor aleatorio generado por una Distribución normal de media 0 y desviación típica $\sigma = 0.3$.

$$Mov(W, \sigma) = W' = (w_1, \dots, w_i + z_i, \dots, w_n)$$

$$z_i \sim N(0, \sigma^2)$$

- Para asegurarnos de que tras aplicar una mutación, nuestro vector resultante siga cumpliendo las restricciones del problema debemos truncar la componente modificada para que su valor se encuentre entre $[0,1]$.

El tamaño del entorno de cada solución es infinito por ser un problema de codificación real. Para solucionar este problema vamos a mutar cada componente del vector w en un orden aleatorio y sin repetición hasta que **haya mejora en la función objetivo** o se hayan mutado todas las componentes. Si se produce mejora aceptamos la solución vecina y comenzamos de nuevo. Si no se produce ninguna mejora tras mutar las n componentes del vector se vuelve a repetir el proceso sobre la solución actual.

Este método que hemos descrito se denomina **Búsqueda Local del primer Mejor**, y en nuestro caso concreto partimos de un vector cuyas componentes son valores aleatorios de una distribución uniforme en el intervalo $[0,1]$ y repetiremos el proceso descrito anteriormente hasta que se hayan realizado un máximo de 15000 llamadas a la función de evaluación o bien hasta que se hayan realizado un máximo de $20 \cdot n$ mutaciones sobre la solución actual sin que haya mejora (se han visitado $20 \cdot n$ vecinos sin que haya mejora).

Vamos a presentar a continuación los pseudocódigos del algoritmo y sus funciones.

- Inicialización Búsqueda Local: Es la función para inicializar el vector w antes del algoritmo. Utilizamos el generador de números pseudoaleatorios basado en el algoritmo de Marsenne Twister, que funciona muy bien como generador de números aleatorios¹.

```
InicializacionBL(int dimension, int i){
    Declaramos el vector w
    Inicializamos el generador mt19937 con la semilla i
    Inicializamos la distribución uniforme en [0,1]

    Mientras que i<dimension:
        elem_generado=dist(gen)
        añadimos a w elem_generado

    return w
}
```

- Movimiento: es la función que realiza la generación de vecinos en el entorno de la solución actual, volvemos a usar el generador de números pseudoaleatorios Marsenne Twister.

```
Mov(vector w, double sigma, int pos, int i){
    Inicializamos el generador mt19937 con la semilla i
    Inicializamos la distribución Normal(0,sigma^2)
```

¹se denomina mt19937 debido a que se basa en el primo $2^{19937} - 1$

```

z=elemento generado por la distribucion Normal
w[pos]=w[pos]+z

Si (w[pos]>1.0){
    w[pos]=1.0
}

Si(w[pos]<0.0){
    w[pos]=0.0;
}
}

```

- Algoritmo de BL: Como aclaración al pseudocódigo, el vector w viene ya inicializado con la función anterior.

```

BusquedaLocal(vector de pares datos, vector w, semilla){
    Establecemos semilla(semilla)
    creamos vector con orden de mutaciones
    contador_evaluaciones=contador_mutaciones=0
    mejpra=false
    Vector w_mutado=w

    Mientras(contador_evaluaciones<15000 && contador_mutaciones<20*tam(w))
        mezclar orden de mutaciones
        mejora=false

        for(i=0; i<tam(w)&& mejora==false; i++)
            w_mutado=w
            Mov(w_mutado,0.3,orden_mutaciones[i])
            contador_mutaciones++
            Obtenemos precision en entrenamiento con LeaveOneOut
            Obtemenos tasa de reduccion
            Obtenemos valor de función evaluación
            contador_ev++

        Si mejora la funcion evaluacion
            w=w_mutado
            mejora=true

        contador_mutaciones=0
}

```

Algoritmo Genéticos

Elementos comunes a todos los AG

Los siguientes algoritmos que usaremos serán algoritmos genéticos, Antes de presentar el esquema de estos algoritmos vamos a definir los siguientes elementos que utilizaremos:

En este tipo de algoritmos partimos de una población inicial $P(t)$ donde t es el contador de generaciones, así $P(0)$ por ejemplo sería la población inicial (en la primera generación).

Por otro lado la población está compuesta por un conjunto de cromosomas C_1, \dots, C_M y a su vez cada cromosoma tiene una serie de genes G_1, \dots, G_N .

La inspiración biológica de este algoritmo se refleja en que trataremos de hallar la solución simulando el comportamiento de las distintas especies en la naturaleza para adaptarse a su entorno. Para ello realizaremos cruces entre cromosomas para generar nuevos miembros de la población y dichos nuevos miembros tendrán mutaciones en sus genes (en algunos casos mejorarán su rendimiento y en otros no, como ocurre en la naturaleza). De esta forma, los mejores miembros (aquellos cuyos genes sean mejores) serán los que sobrevivan y se encuentren en la población de la siguiente generación.

Con este enfoque se diseñan dos algoritmos para resolver el problema APC, un **Algoritmo Genético Generacional** y un **Algoritmo Genético Estacionario** con dos tipos de cruce distintos:

- **Cruce BLX- α :** Es un método de cruce en el cual partiendo de dos padres $C1 = (c_{11}, \dots, c_{1n})$ y $C2 = (c_{21}, \dots, c_{2n})$ se generan dos descendientes $h_k = (h_{k1}, \dots, h_{kn})$ con $k = 1, 2$. Así, cada gen de los dos hijos generados se genera aleatoriamente en el intervalo $[C_{min} - l\alpha, C_{max} + l\alpha]$ donde:

- $C_{min} = \min\{c_{1i}, c_{2i}\}.$
- $C_{max} = \max\{c_{1i}, c_{2i}\}.$
- $l = C_{max} - C_{min}.$
- $\alpha \in [0, 1].$

La peculiaridad de este método de cruce es que aunque en su mayoría de veces, el valor aleatorio generado cae en el intervalo $[C_{min}, C_{max}]$ la constante $l\alpha$ permite realizar una “exploración” fuera de ese intervalo y probar con otros valores para el gen, lo que en ciertos casos puede ayudar a converger a una mejor solución para el problema.

- **Cruce Aritmético:** Se realiza una media ponderada según un factor $\alpha \in [0, 1]$ de manera que partiendo de dos padres $C1 = (c_{11}, \dots, c_{1n})$ y $C2 = (c_{21}, \dots, c_{2n})$ se generan dos descendientes $h_k = (h_{k1}, \dots, h_{kn})$ con $k = 1, 2$ de manera que $h_{ki} = \alpha c_{1i} + (1 - \alpha)c_{2i}$. En este caso todos los valores generados estarán en el intervalo $[\max(c_{1i}, c_{2i}), \min(c_{1i}, c_{2i})]$, por lo que no se llevará a cabo la “exploración” del método anterior. Esto conlleva que los genes tenderán siempre a decrecer.

El criterio de parada del algoritmo será alcanzar las 15000 evaluaciones de la función objetivo.

Algoritmo Genético Generacional (AGG)

Basado en la idea anteriormente explicada, este algoritmo sigue el siguiente esquema:

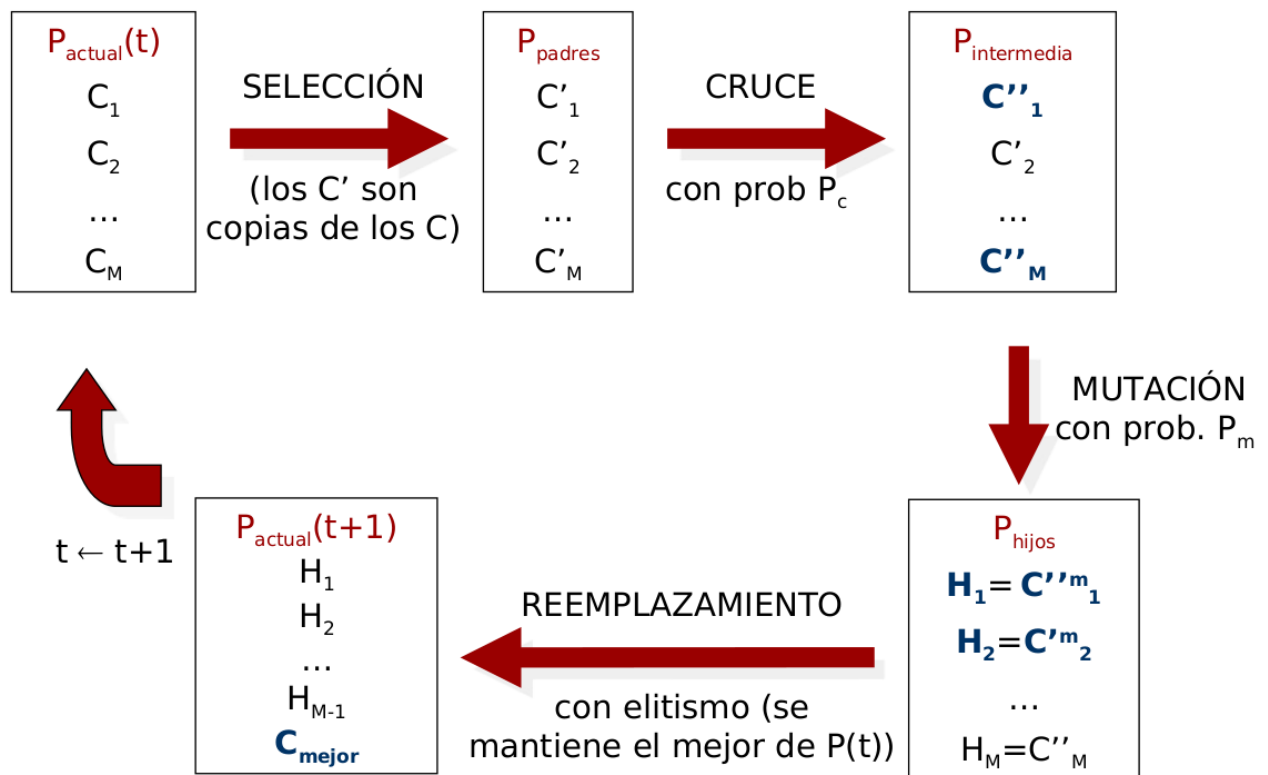


Figure 3.1: Esquema AGG

Que traducido a pseudocódigo sería:

```
AlgoritmoGeneticoGeneracional(datos, vector w , int tam_poblacion, int tipo_cruce, generador mt19937){  
  
    dim=num_atributos  
    contador_evaluaciones=0  
    definimos vectores de dim tam_poblacion x dim: poblacion, seleccion, cruce  
  
    Inicializar poblacion  
    Evaluar poblacion  
  
    while (contador_evaluaciones<15000){  
        Seleccionamos Padres  
        Cruzamos padres con BLX (si tipo=1) o con cruce Aritmetico (si tipo=2)  
        Mutamos los cruces  
        Reemplazamos la población original  
        Evaluamos la nueva población  
    }  
  
    w=mejor cromosoma de la poblacion  
}
```

La inicialización de la población se realiza de la siguiente manera:

```
Inicializar(tam_poblacion, dim, generador numeros aleatorios){  
  
    poblacion= matriz de dimensión tam_poblacion x dim  
  
    for(int k=0; k < tam_poblacion; k++){  
        for(int i=0; i < dim; i++){  
            elemento=generamos un valor aleatorio de una distribución uniforme (0,1)  
            poblacion[k,i]=elemento  
        }  
    }  
  
    return poblacion  
}
```

La función de evaluación es la siguiente:

```
Evaluacion( matriz poblacion, matriz datos, int solucion, vector vfitness){  
  
    for(int i=0; i < poblacion.size(); i++){  
        Calculamos tasa de clasificación de poblacion[i]  
        Calculamos tasa reducción de poblacion[i]  
        Calculamos función objetivo de población[i]  
  
        vfitness[i]=fitness calculado  
  
        Si el fitness es el máximo actualmente  
            solucion=i  
    }  
}
```

Como aclaración, para acelerar tiempos de ejecución usaremos un vector que contiene para cada elemento de la población su valor de la función objetivo, dicho vector es que llamamos *vfitness*.

La función encargada del proceso de selección utilizará el torneo binario como criterio, es decir, seleccionará en cada paso dos cromosomas y seleccionará al mejor de ellos (con repetición), el pseudocódigo es el siguiente:

```
Seleccion (matriz datos, matriz poblacion, matriz seleccion, generador num aleatorios,
vector vfitness){
```

```
    for(int i=0; i < poblacion.size(); i++){
        indice1=generamos valor aleatorio entre [0,tam-1]
        indice2=generamos valor aleatorio entre [0,tam-1]

        if(vfitness[indice1]>vfitness[indice2])
            metemos poblacion[indice1] en seleccion
        else
            metemos poblacion[indice2] en seleccion
    }
}
```

Tras esto llevamos a cabo el cruce entre los cromosomas de la selección:

```
Cruce(matriz seleccion,int tipo, double alpha, double probabilidad_cruce,
matriz cruce, generador de num aleatorios){
    cruces=probabilidad_cruce*num_cromosomas/2
```

```
    if(tipo==1){
        Para cada parejas de padres consecutiva en seleccion:
            Si sus índices están por debajo de cruces
                Cruce BLX con factor alpha y
                se insetan los hijos en la matriz cruce
            Si los índices son posteriores a cruces
                Se insertan en la matriz cruce sin alterar
    }else if(tipo==2){
        Para cada parejas de padres consecutiva en seleccion:
            Si sus índices están por debajo de cruces
                Cruce ARITMETICO con factor alpha y
                se insetan los hijos en la matriz cruce
            Si los índices son posteriores a cruces
                Se insertan en la matriz cruce sin alterar
    }
}
```

La función encargada del cruce BLX- α es:

```
BLX(vector padre1, vector padre2, matriz cruce, double alpha, generador){
```

```
    vector hijo1, hijo2;

    for(int i=0; i<padre1.size(); i++){
        max=maximo entre padre1[i] y padre2[i]
        min=minimo entre padre1[i] y padre2[i]
        l=max-min
        Creamos distribucion uniforme (min-l*alpha, max+l*alpha)
        Generamos dos valores aleatorios
```

```

    hijo1.push_back(elemento_generado1)
    hijo2.push_back(elemento_generado2)
}

```

```

metemos en cruce tanto hijo1 como hijo2
}

```

La función para el cruce Aritmético la realizamos con un parámetro α (que es distinto al que se usa en BLX- α) para que no hagamos la media aritmética exacta de dos genes, pues de ser así crearíamos dos veces el mismo hijo, en su lugar generamos un alpha aleatorio y la media se hace como $\alpha \cdot g_{1i} + (1 - \alpha) \cdot g_{2i}$ sería:

```

ARITMETICO(vector padre1, vector padre2,matriz cruce, generador){
    Creamos distribución uniforme en (0,1)
    vector hijo

    for (int k=0; k<2; k++){
        generamos alpha
        for(int i=0; i< tam_padre ; i++){
            elemento=alpha*padre1[i] + (1-alpha)*padre2[i]
            metemos elemento en hijo
        }
        metemos hijo en la matriz de cruce
        vaciamos hijo
    }
}

```

Tras esto se producen las mutaciones en los genes:

```

Mutacion(matriz cruce, double pm, generador_num_aleatorios){
    Calculamos el numero de cromosomas num_cromosomas
    calculamos el numero de genes num_genes
    int num_mutaciones=pm*num_cromosomas*num_genes
    int fila, col;

    for(i=0; i<num_mutaciones; i++){
        fila=generamos num aleatorio%num_cromosomas
        col=generamos num aleatorio%num_genes
        Mov(cruce[fila],0.3,col,generador_num_aleatorios)
    }
}

Mov(vector w, double sigma, int pos, generador_num_aleatorios){
    Inicializamos distribucion normal (0.0,sigma)

    z=elemento generado por la distribucion Normal
    w[pos]=w[pos]+z

    Si (w[pos]>1.0){
        w[pos]=1.0
    }

    Si(w[pos]<0.0){
        w[pos]=0.0;
    }
}

```


Finalmente los elementos de la población anterior son reemplazados y se evalúa la nueva población. En este caso se realizarán conjuntamente estas dos etapas:

```
ReemplazarYEvaluar(matriz poblacion, matriz mutaciones, datos, vector w,  
int solucion, vector vfitness){
```

```
    mejor_fitness_anterior=vfitness[solucion]  
    vector auxiliar v
```

```
    limpiamos poblacion  
    limpiamos vfitness
```

```
    for(int i=0; i< mutaciones.size(); i++){  
        v=mutaciones[i]  
        calculamos fitness de v  
        metemos el fitness en vfitness
```

```
        Si es el mejor fitness por ahora{  
            solucion=i  
            mejor_fitness=fitness de v  
        }
```

```
        Si es el peor fitness por ahora{  
            peor=i  
            peor_fitness=fitness  
        }  
    }
```

```
    Si(peor_fitness>mejor_fitness_anterior)  
        poblacion=mutaciones  
    Si no{  
        poblacion=mutaciones  
        poblacion[indice_peor]=w  
    }
```

```
    Si(mejor_fitness > mejor_fitness_anterior)  
        w=poblacion[solucion]  
}
```

Como aclaración, w es el vector solución de la población anterior, que en caso de ser mejor que el peor de la nueva población lo mantenemos una generación más.

Algoritmo Genético Estacionario (AGE)

En este caso, el esquema a seguir es el siguiente:

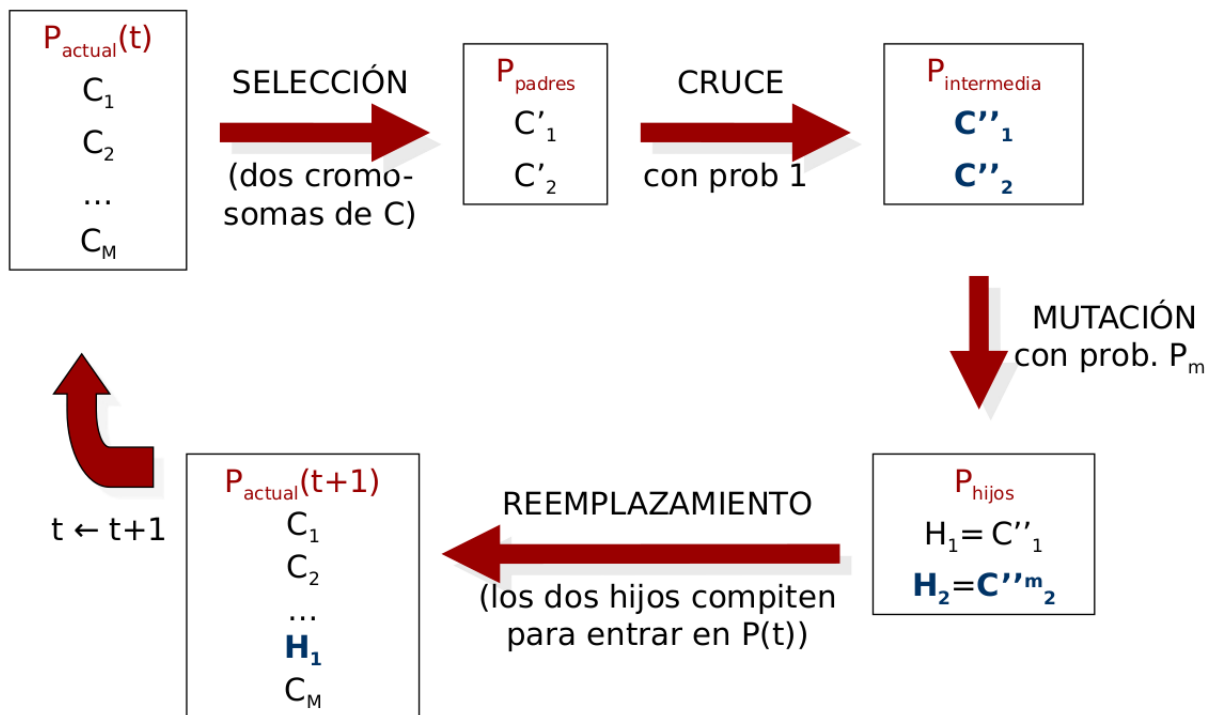


Figure 3.2: Esquema AGE

Como vemos, en esencia es similar al caso del AGG, las dos diferencias son que en este caso sólo elegimos dos cromosomas como padres en cada iteración, el cruce entre estos dos padres se realiza con probabilidad 1 (pero sigue siendo con BLX- α o con cruce Aritmético), y que los dos hijos generados compiten con los dos peores de la población actual para entrar en la nueva generación (los dos cromosomas con mejor fitness pasan).

Así, el pseudocódigo del algoritmo AGE sería similar al del AGG, con la salvedad de que algunas funciones se realizan de forma distinta.

Para empezar, la función de Evaluación de la población al comienzo del algoritmo (tras la inicialización de la población), se va a reemplazar por la siguiente función:

CalculaFitness(matriz poblacion, datos, vector vfitness, **int** solucion, **int** peor, **int** segundo peor){

Para cada vector v de la poblacion:

Calculamos el fitness de v

Lo almacenamos en vfitness

Si(fitness de v > mejorfitness){

mejorfitness=fitness de v

pos_mejor= pos de v

}

Si (fitness de v < peorfitness){

peorfitness=fitness de v

pos_peor=pos de v

}

//Calculamos segundo peor de vfitness

vfitness[pos_peor]=vfitness[pos_peor]+100.0

pos_segundo_peor=CalculaPeor(vfitness);

vfitness[pos_peor]=vfitness[pos_peor]-100.0;

}

int CalculaPeor(vector vfitness){

min=100.0

Para cada elemento en vfitness{

si(elemento < min){

min=elemento

peor=pos_elemento

}

}

return peor

}

La función selección permanece idéntica al caso AGG, con la excepción de que se le especifica por parámetros que solo tome dos elementos como padres.

Del mismo modo, las funciones de cruce y mutación permanecen idénticas con la excepción de que en el cruce se especifica que la probabilidad de cruce es de 1.0, así nos aseguramos que se cruzan los dos padres seleccionados.

Finalmente la función ReemplazarYEvaluar del caso AGG se sustituye por la siguiente:

```
ReemplazamientoCompetitivo(poblacion, mutaciones, datos, vfitness, pos_mejor,
pos_peor, pos_segundo_peor){
```

```
    Para cada hijo h en mutaciones{
        fitness1=Calculamos el fitness de h
        if(fitness1>vfitness[pos_segundo_peor]){
            vfitness[pos_segundo_peor]=fitness1
            poblacion[pos_segundo_peor]=h

            //Vemos si es mejor que el actual mejor
            Si (vfitness[pos_mejor]<fitness1){
                pos_mejor=pos_segundo_peor
            }

            //Calculamos segundo peor de nuevo
            vfitness[pos_peor]=vfitness[pos_peor]+100.0
            pos_segundo_peor=CalculaPeor(vfitness);
            vfitness[pos_peor]=vfitness[pos_peor]-100.0;
        }
        else if(vfitness[pos_peor]<fitness1){ //Está entre los dos peores
            vfitness[pos_peor]=fitness1;
            poblacion[pos_peor]=h; //Sigue siendo el peor
        }
    }
}
```

Algoritmo Memético

Utilizaremos también **algoritmos meméticos**, que son un híbrido entre el algoritmo de *búsqueda local* y *los algoritmos genéticos*. La justificación de esta hibridación reside en que los algoritmos de búsqueda local son malos exploradores (es decir tienden a quedar atrapados en extremos locales) pero buenos explotadores (alcanzamos con precisión el extremo local), en cambio los genéticos son buenos exploradores (no quedan atascados en extremos locales) y malos explotadores (no logran alcanzar extremos con precisión). Es por ello que esta hibridación permite explotar al máximo lo que mejor sabe hacer cada algoritmo, el esquema será por tanto:

1. Ejecutar un algoritmo genético durante un número t de generaciones establecido (realizando una buena exploración).
2. Ejecutar un algoritmo de búsqueda local a la población actual (realizando una buena explotación).
3. Repetir el proceso anterior hasta alcanzar un máximo de evaluaciones de la función objetivo.

Con esta nueva propuesta realizaremos varios experimentos, primero cambiaremos el tamaño de la población de 30 a 10, la probabilidad de cruce y mutación permanecerán similares al AGG. Por otro lado las versiones que implementaremos serán:

- **Algoritmo Memético (10,1.0):** Cada 10 generaciones se lleva a cabo una búsqueda local (de $2 \cdot \text{numgenes}$ iteraciones) de todos los cromosomas de la población. Se repite el proceso hasta alcanzar las 15000 evaluaciones.
- **Algoritmo Memético (10,0.1):** Cada 10 generaciones se lleva a cabo una búsqueda local (de $2 \cdot \text{numgenes}$ iteraciones) del 10% de los cromosomas de la población, esta versión es más rápida que la anterior pero a cambio solo se mejora un cromosoma aleatorio de la población. Se repite el proceso hasta alcanzar las 15000 evaluaciones.

- **Algoritmo Memético (10,0.1Mej):** Cada 10 generaciones se lleva a cabo una búsqueda local (de $2 \cdot numgenes$ iteraciones) del 10% de los mejores cromosomas de la población, versión mejorada de la anterior que busca centrar esfuerzos en mejorar únicamente los mejores cromosomas de la población. Se repite el proceso hasta alcanzar las 15000 evaluaciones.

La elección de cada tipo de algoritmo se realizará por parámetros que pasaremos al algoritmo, pues los tres algoritmos son iguales en esencia, así habrá un parámetro *pls* que nos da la probabilidad de que el cromosoma haga la búsqueda local que valdrá 1.0 en el primer algoritmo y 0.1 en los dos siguientes y un parámetro booleano llamado *mejor* que indica si la búsqueda local es en los mejores o en cualquier elemento:

De esta forma el Pseudocódigo del algoritmo será el siguiente:

```

AlgoritmoMemetico(datos, vector w , int tam_poblacion, int tipo_cruce, int tipo, double pls, bool mejores,

    int cont_generaciones=0
    int dim=num_atributos
    int contador_evaluaciones=0
    int ncromosomas=tam_pob*pls; //Esperanza matematica
    definimos vectores de dim tam_poblacion x dim: poblacion, seleccion, cruce
    vector vfitness //fitness de los cromosomas de la poblacion actual

    Inicializar poblacion
    Evaluar poblacion

    while (contador_evaluaciones<15000){
        Seleccionamos Padres
        Cruzamos padres con BLX (si tipo=1) o con cruce Aritmetico (si tipo=2)
        Mutamos los cruces
        Reemplazamos la población original
        Evaluamos la nueva población
        contador_generaciones++
        vector pos_mejores //para la tercera versión

        Si(contador_generaciones%10==0){
            contador_generaciones=0
            //Si no es la versión de mejorar mejores
            Si(!mejor){
                Para cada i in ncromosomas{
                    BusquedaLocal(datos, poblacion[i],evaluaciones, generador_num_aleatorios)
                }
            }Si no{
                pos_mejores=CalculaMejores(vfitness,ncromosomas);
                Para cada i in pos_mejores{
                    BusquedaLocal(datos, poblacion[i],evaluaciones, generador_num_aleatorios)
                }
            }
        }
    }

    w=mejor cromosoma de la poblacion
}

```

```

vector Calculamejores (vfitness, int n){
    vector posiciones
    vector vfitness_ordenado

    vfitness_ordenado= Ordenamos de menor a mayor el vector de vfitness

    Para cada i<n{
        //tomamos el elemento i-esimo maximo
        elem=vfitness_ordenado[vfitness.size()-i-1]
        posiciones.push_back(indice correspondiente a elem en vfitness)
    }

    return posiciones
}

```

Todas las funciones que se emplean son idénticas a las de AGG y la Búsqueda Local es igual a la ya explicada con la salvedad de que el criterio de parada es hacer $2 \cdot numgenes$ mutaciones.

Chapter 4

Procedimiento

Para el desarrollo de la práctica se ha optado por implementar todo en el lenguaje c++ y a partir de código propio, únicamente haciendo uso de funciones y estructuras de datos de la STL y de la librería random para los procesos aleatorios.

Se ha estructurado todo en una carpeta denominada Software que contiene a su vez una carpeta includes con los ficheros de cabecera empleados: utilidades.h, RELIEF.h y BL.h.

El fichero utilidades.h tiene funciones utilizadas por todos los algoritmos como el clasificador 1NN, Leave One Out u otras. Mientras que las otras dos contienen las funciones específicas de los otros dos algoritmos.

También se dispone de una carpeta src dónde se enciendan las implementaciones de los ficheros anteriores así como el programa principal.

Por otro lado encontramos la carpeta instancias_APC con las bases de datos.

Finalmente, para compilarlo todo se dispone de un Makefile que además de la compilación contiene las reglas run y clean, para ejecutar y limpiar los directorios respectivamente.

Manual de uso

Para compilar y ejecutar el proyecto se tiene un makefile con reglas para compilar (make) y ejecutar (make run_parkinsons, make run_ionosphere y make run_heart) no es necesario recompilar para cambiar la base de datos pues se pasa como argumento al ejecutable según la orden del makefile que usemos.

Chapter 5

Experimentos y análisis de resultados

Casos del problema empleados y parámetros utilizados.

En la resolución del problema hemos optado por mezclar los datos y normalizarlos sea cual sea la base de datos empleada, así nos aseguramos la efectividad del clasificador y que se reduzca el problema de clases desbalanceadas como ocurre en la base de datos de **Parkinsons**. Por otro lado, para que los procesos aleatorios, como la generación de vectores aleatorios que siguen una distribución uniforme al comienzo de la Búsqueda Local, sean reproducibles en cualquier ordenador, hemos establecido una semilla al llamar a cada una de estas funciones que coincidía con la iteración de 5-fold Cross Validation en la que se encuentre el programa en ese momento, por lo que las semillas usadas son 1,2,3,4,5. Estas se usan también para inicializar el generador de números pseudoaleatorios mt19937 cuando lo usamos en las mutaciones de la Búsqueda local, solo que en lugar de utilizar las iteraciones de Cross Validation usaremos el índice del bucle for que recorre los atributos de w realizando las mutaciones aleatorias.

Resultados Obtenidos

Obtenemos los resultados para el algoritmo **RELIEF**:

Table 5.1: Resultados en el Dataset Ionosphere para RELIEF

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	85.714	2.941	44.327	3.394
Partición 2	90	2.941	46.470	3.272
Partición 3	90	2.941	46.470	4.259
Partición 4	84.285	2.941	43.613	3.316
Partición 5	88.732	2.941	45.836	3.485
Media	87.746	2.941	45.343	3.545

Table 5.2: Resultados en el Dataset Parkinsons para RELIEF

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	94.871	0	47.435	1.199
Partición 2	94.871	0	47.435	1.017

Particiones	Parkinsons			
Partición 3	97.435	0	48.717	1.213
Partición 4	92.307	0	46.153	1.328
Partición 5	100	0	50	1.005
Media	95.897	0	47.948	1.152

Table 5.3: Resultados en el Dataset Spectf_heart para RELIEF

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	79.71	0	39.855	4.122
Partición 2	82.608	0	41.304	4.498
Partición 3	76.811	0	38.405	4.172
Partición 4	85.507	0	42.753	4.781
Partición 5	90.411	0	45.2055	3.952
Media	83.009	0	41.504	4.303

Obtenemos los resultados para el algoritmo de **Búsqueda Local**:

Table 5.4: Resultados en el Dataset Ionosphere para BL

Particiones	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
Partición 1	85.714	58.823	72.268	8825.46
Partición 2	87.142	55.882	71.512	3509.74
Partición 3	87.142	79.411	83.277	8396.57
Partición 4	91.428	58.823	75.126	3881.64
Partición 5	87.323	64.705	76.014	8485.97
Media	87.750	63.529	75.64	6619.88

Table 5.5: Resultados en el Dataset Parkinsons para BL

Particiones	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
Partición 1	100	50	75	788.805
Partición 2	76.923	77.272	77.097	1018.39
Partición 3	94.871	72.727	83.799	965.224
Partición 4	97.435	68.181	82.808	426.535
Partición 5	97.435	68.181	82.808	804.335
Media	93.333	67.272	80.303	800.658

Table 5.6: Resultados en el Dataset Spectf_heart para BL

Particiones	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
Partición 1	79.71	72.727	76.218	5768.42
Partición 2	78.260	54.545	66.403	5768.42
Partición 3	81.159	65.909	73.534	9876.57
Partición 4	86.956	59.090	73.023	6979.56
Partición 5	86.301	56.818	71.559	11173.2
Media	82.477	61.818	72.147	9528.56

Los resultados anteriores se han obtenido en cada iteración del proceso de 5-fold Cross Validation y como podemos observar, por regla general el método **RELIEF** consigue unas tasas muy elevadas de Precisión sobre el conjunto de Test en todas las bases de datos utilizadas, **pero en cambio este método no consigue reducir atributos** o al menos no sirve para este cometido (a excepción de la base de datos de **Ionosphere** dónde se reduce en un 3% aproximadamente), por lo que no consigue una evaluación muy alta de la función objetivo, ya que nuestro propósito era reducir el mayor número de atributos posible a la vez que intentar mantener una elevada precisión al clasificar.

Sin embargo, el método de **Búsqueda local** si que tiene un comportamiento que se ajusta mejor a lo que queríamos obtener, pues como podemos observar, aunque las tasas de clasificación son siempre menores o iguales que las obtenidas con el método RELIEF, obtenemos unas tasas de reducción muy elevadas, por encima del 60% en la mayoría de los casos, y logrando un aumento muy considerable en los valores de la función objetivo en comparación con los obtenidos con RELIEF (normalmente por encima de los 70). Este hecho puede deberse a la naturaleza del algoritmo, pues las mutaciones se realizan en cada paso sobre un único atributo en lugar de todos a la vez como en RELIEF, por lo que es más fácil encontrar atributos relevantes y no relevantes.

Por otro lado, comparando los tiempos empleados por ambos algoritmos podemos ver que el método RELIEF es mucho más rápido que el de Búsqueda Local, pues en media tarda unos 2-3ms en entrenar, en cambio el de Búsqueda Local está muy condicionado a si las mutaciones mejoran más o menos la función objetivo lo que en un caso extremo podría llevar a ejecutar 15000 evaluaciones de la función objetivo, es por ello que los tiempos son mayores, aunque muy razonables para la notable mejora conseguida con respecto a RELIEF (menos de 10 segundos normalmente).

Como resumen tenemos:

Table 5.7: Resumen resultados en el Dataset Ionosphere para todos los algoritmos

Algoritmos	Ionosphere			
	% clas	% red	Agr.	Tiempo ms
1-NN	86.599	0	43.299	2.653
RELIEF	87.746	2.941	45.343	3.545
Búsqueda Local	87.750	63.529	75.64	6619.88

Table 5.8: Resumen resultados en el Dataset Parkinsons para todos los algoritmos

Algoritmos	Parkinsons			
	% clas	% red	Agr.	Tiempo ms
1-NN	93.333	67.272	80.303	800.658
RELIEF	95.897	0	47.948	1.152
Búsqueda Local	93.333	67.272	80.303	800.658

Table 5.9: Resumen resultados en el Dataset Spectf_heart para todos los algoritmos

Algoritmos	Spectf_heart			
	% clas	% red	Agr.	Tiempo ms
1-NN	96.923	0	48.461	0.58
RELIEF	83.009	0	41.504	4.303
Búsqueda Local	82.477	61.818	72.147	9528.56

En estas tablas podemos ver mejor todo lo comentado anteriormente, y en vista de los resultados obtenidos por el método de Búsqueda Local podemos concluir:

- No todos los atributos recogidos son necesarios para obtener una alta tasa de clasificación, pues eliminando más de la mitad en cada base de datos se obtienen valores de clasificación muy elevados también.
- El tiempo es muy razonable, por lo que con apenas unos segundos más que en RELIEF obtenemos resultados mucho mejores.

No obstante, hay un hecho importante que no se aprecia en las tablas en el método de búsqueda local, y que se aprecia cuando se imprime por pantalla en cada iteración de la validación cruzada el vector solución obtenido, y es que podría decirse que hay atributos que son mutuamente excluyentes, en el sentido de que si uno está muy próximo a 1, entonces el otro está muy próximo a 0 y viceversa sin que se vea afectada en exceso la tasa de clasificación.

Por ejemplo, en el caso de la base de datos **Parkinsons**, obtenemos las siguientes soluciones por pantalla al ejecutar el algoritmo:

Particion: 1

Solucion obtenida:

0.781141, 0, 0.0552385, 0.999041, 0.0200457, 0, 0,
0.0950612, 0.935539, 0.846311, 0.0972302, 0.524548, 0,
0.00326139, 0, 0.913962, 0.752749, 0.841574, 0.939128,
0.0387805, 0.715971, 1,

Particion: 2

Solucion obtenida:

0.0635937, 0, 1, 0, 0, 0.0329383, 0.698863, 0, 0,
0.0580529, 0.0398286, 0.998241, 0, 0.0121121, 0.719754, 0,
0.0819994, 0, 0.0600942, 1, 0.080975, 0.0718255,

Particion: 3

Solucion obtenida:

0.0707249, 0.839949, 0.0552385, 0, 0, 0.018748, 0.0406307,
0.0100419, 0.0935515, 1, 0, 0, 0, 0, 0.913301, 0.696564,
0.398963, 0, 0, 0.0903173, 0.863963,

Particion: 4

Solucion obtenida:

0.900621, 0, 0.855621, 0.0343507, 0.0228713, 0, 0.0595821,
0.0869721, 0.903179, 0.0582782, 0.00515915, 0.572356, 0,
0.0294471, 0.728605, 0.811948, 0.0979347, 0.0504305,
0.0692294, 0.961353, 0.0190247, 0.0397804,

Particion: 5

Solucion obtenida:

0.0551801, 0.831328, 0, 0.979445, 0.089821, 0, 0, 0,
0.990821, 0.808282, 0.074774, 0.819473, 0.0264971, 0,
0.0580757, 0, 0.0875768, 0.877903, 0.0697856, 0.673025, 0,
0.0698192,

Si nos fijamos, los atributos ponderados por encima de 0.7 (valor a partir del cual considero que un atributo es muy relevante) en la primera iteración serían: 1,4,9,10,17,18,19,21,22

Sin embargo en la segunda iteración son: 3,12,15,20

Como vemos, ninguno de los ponderados en la primera iteración por encima de 0.7 se repite en la segunda iteración.

Si seguimos ahora con la tercera iteración: 2, 10, 16, 22

Como vemos en este caso se ponderan algunos atributos comunes con la primera iteración y otros no ponderados hasta ahora por encima de 0.7.

En cambio, los valores de clasificación son todos muy similares y elevados.

Y este hecho se ve en cada iteración, lo que conlleva que hay ciertos atributos que podrían tener información redundante a la hora de clasificar y por lo tanto mejora la función objetivo el hecho de reducirlos si su “contrario” se incrementa. Es por ello que no considero apropiado devolver como resultado final al problema la media de los 5 vectores obtenidos, pues en cada caso se ponderan atributos diferentes y si mostramos el vector promedio:

Solucion obtenida:

0.374252, 0.334255, 0.39322, 0.402567, 0.0265476,
0.0103373, 0.159815, 0.038415, 0.584618, 0.554185,
0.0433984, 0.582924, 0.00529942, 0.00896412, 0.301287,
0.527842, 0.343365, 0.433774, 0.227647, 0.534632,
0.181258, 0.409078,

Como vemos no hemos conseguido reducir a tantos atributos por debajo de 0.1 como en las anteriores iteraciones de Cross Validation, solo aquellos que son verdaderamente irrelevantes en todas las iteraciones porque nunca o casi nunca se han ponderado, como ocurre en los atributos: 5,6,8,11,13,14, que no han salido en ninguna de las iteraciones anteriores analizadas ponderados por encima de 0.7.

Además, al tener este comportamiento excluyente, el resto de atributos tienden a quedarse en términos cercanos a 0.4-0.5, pues en una iteración se ven incrementados notablemente y en la siguiente son prácticamente 0 por haber aumentado otros, y ninguno supera el 0.7, por lo que con esta solución no reflejamos el hecho de que hay atributos mucho más importantes que otros y mucho menos el hecho de que algunos atributos son **redundantes**.

Este hecho no ocurre tanto con el algoritmo **RELIEF**, pues si analizamos la salida de cada iteración del mismo modo en que hemos hecho con el de Búsqueda Local:

Particion: 1

Solucion obtenida:

1, 0.388353, 0.84248, 0.401231, 0.378912, 0.402414,
0.436481, 0.402141, 0.574087, 0.473696, 0.581067, 0.512275,
0.395842, 0.581054, 0.297156, 0.716484, 0.645781, 0.880921,
0.751947, 0.890509, 0.547066, 0.928198,

Particion: 2

Solucion obtenida:

0.619862, 0.310448, 0.560894, 0.2776, 0.302762, 0.294823,
0.366709, 0.294531, 0.470386, 0.382529, 0.502991, 0.425973,
0.309465, 0.502861, 0.174225, 0.401239, 0.585643, 0.949089,
0.717267, 0.759623, 0.34737, 1,

Particion: 3

Solucion obtenida:

1, 0.297727, 0.843674, 0.27533, 0.273018, 0.308313,
0.376326, 0.308253, 0.730687, 0.612213, 0.796042,
0.649244, 0.463112, 0.796029, 0.123959, 0.606604,
0.647272, 0.820244, 0.732885, 0.793197, 0.461053, 0.868964,

Particion: 4

Solucion obtenida:

0.897252, 0.27068, 0.801999, 0.266364, 0.300469, 0.31335,
0.370151, 0.313183, 0.539956, 0.465823, 0.575747, 0.480578,
0.346269, 0.575613, 0.155191, 0.578079, 0.751897, 1, 0.715037,
0.854044, 0.555776, 0.941265,

Particion: 5

Solucion obtenida:

0.751032, 0.270351, 0.577874, 0.375149, 0.363059, 0.419926,
0.505587, 0.419608, 0.580489, 0.49853, 0.617586, 0.533746,
0.373091, 0.617322, 0.197938, 0.549694, 0.65334, 0.795322,
0.791317, 0.856609, 0.553759, 1,

MEDIA RELIEF:

0.853629, 0.307512, 0.725384, 0.319135, 0.323644, 0.347765,
0.411051, 0.347543, 0.579121, 0.486558, 0.614686, 0.520363,
0.377556, 0.614576, 0.189694, 0.57042, 0.656787, 0.889115,
0.741691, 0.830796, 0.493005, 0.947685,

Como vemos, los atributos por encima de 0.7 en la primera iteración serían:1,3,16,18,19,20,22

En la segunda iteración serían: 18, 19 , 20 y 22.

En la tercera iteración: 1,3,9,11,14,18,19,20,22

Como vemos coinciden en su gran mayoría en cada iteración, y si nos fijamos en el vector pormedio solución, podemos ver que en este caso si hay valores por encima de 0.7 que son los que por regla general más se han ponderado en cada iteración.

La explicación a este comportamiento tan distinto entre el algoritmo RELIEF y el algoritmo de Búsqueda Local, en mi opinión, puede estar en la aleatoriedad que conlleva el algoritmo de búsqueda local, pues al mutar aleatoriamente cada componente si comienza aumentando el peso de unas, la función de evaluación le obliga a reducir el de otras pues como hemos visto son excluyentes.

Chapter 6

Referencias Bibliográficas

En esta práctica el material utilizado ha sido por regla general el proporcionado en el Seminario 2, así como el Tema 2 de teoría.

Otros enlaces utilizados:

- Generador mt19937: <https://www.cplusplus.com/reference/random/mt19937/>
- Distribución Uniforme: https://www.cplusplus.com/reference/random/uniform_real_distribution/
- Distribución normal: https://www.cplusplus.com/reference/random/normal_distribution/