Práctica Algoritmos Genéticos y Meméticos APC Metaheurísticas

Ignacio Aguilera Martos
DNI: 77448262V e-mail: nacheteam@correo.ugr.es
Grupo de prácticas 1 Lunes 17:30-19:30

Curso 2017-2018

Índice

1. Introducción del problema

Para el problema de clasificación partimos de un conjunto de datos dado por una serie de tuplas que contienen los valores de atributos para cada instancia. Esto es una n-tupla de valores reales en nuestro caso.
El objetivo del problema es obtener un vector de pesos que asocia un valor en el intervalo $[0, 1]$ indicativo de la relevancia de ese atributo. Esta relevancia va referida a lo importante que es en nuestro algoritmo clasificador ese atributo a la hora de computar la distancia entre elementos.
Resumiendo lo que tenemos es un algoritmo clasificador que utiliza el vector de pesos calculado para predecir la clase a la que pertenece una instancia dada. Este algoritmo clasificador es el KNN con k=1. Lo que hace es calcular según la distancia euclídea (o cualquier otra) la tupla más cercana a la que queremos clasificar ponderando cada atributo con el correspondiente peso del vector, es decir, la distancia entre dos elementos sería:

$$d(e, f) = \sqrt{\sum_{i=0}^{n} w_i * (e_i - f_i)}$$

Donde e y f son instancias del conjunto de datos, w el vector de pesos y n la longitud de e y f que es la misma.
La calificación que se le asigna al vector w depende de dos cosas: la tasa de aciertos y la simplicidad.
La tasa de aciertos se mide contando el número de aciertos al emplear el clasificador descrito y la simplicidad se mide como el número de elementos del vector de pesos que son menores que 0.2, ya que estos pesos no son empleados por el clasificador, o lo que es lo mismo, son sustituidos por cero. Por lo tanto las calificaciones siguen las fórmulas:

$$Tasa\_acierto = 100 \cdot \frac{n^o\ aciertos}{n^o\ datos}\ ,\ Tasa\_simplicidad = 100 \cdot \frac{n^o\ valores\ de\ w\ <\ 0{,}2}{n^o\ de\ atributos}$$

$$Tasa\_agregada = \frac{1}{2} \cdot Tasa\_acierto + \frac{1}{2} \cdot Tasa\_simplicidad$$

Cabe destacar que todas las tasas están expresadas en porcentajes, por lo tanto cuanto más cercano sea el valor a 100 mejor es la calificación.
De esta forma a través del algoritmo que obtiene el vector de pesos para el conjunto de datos dado y el clasificador obtenemos un programa que clasifica de forma automática las nuevas instancias de datos que se introduzcan.

2. Introducción de la práctica

En esta práctica he desarrollado algoritmos genéticos tanto estacionarios como generacionales con los dos operadores de cruce propuestos (aritmético y BLX) así como un algoritmo memético basado en el genético generacional.
Al igual que en la práctica anterior el objetivo es ejecutar estos algoritmos sobre los conjuntos de datos dados para observar su comportamiento y realizar una comparativa entre los mismos.
A los algoritmos mencionados anteriormente se les suman el 1NN con todos los pesos a uno, el greedy Relief y la búsqueda local (algoritmos implementados en la primera práctica).
Igual que en la práctica anterior he realizado un procesamiento de los datos para eliminar tuplas repetidas, de forma que ya tenemos implementado el leave one out para conjuntos distintos en el KNN y además, he implementado una versión más rápida de KNN usando la librería NumPy con la intención de reducir tiempos en la búsqueda local, en los algoritmos genéticos y en los meméticos.

En la práctica se desarrollará cómo he implementado los algoritmos genéticos (en sus dos variantes) incluyendo los operadores de cruce y mutación así como su adaptación a algoritmo memético.

3. Descripción común a todos los algoritmos

Los algoritmos empleados han sido el KNN, el algoritmo greedy Relief, la metaheurística de búsqueda local, un algoritmo genético estacionario, un algoritmo genético generacional y un memético basado en el algoritmo genético generacional.
Estos algoritmos comparten ciertos métodos y operadores que pasaré a explicar en esta sección.
Para empezar se debe destacar que la representación escogida para las soluciones es un vector de números reales, es decir, si n es el número de características:

$$w \in \mathbb{R}^n \ t.q. \ \forall i \ con \ 0 \leq i < n \ se \ tiene \ w_i \in [0,1]$$

O lo que es lo mismo, un vector de tamaño n con todas las posiciones rellenas con números del intervalo [0,1].
A estos números me referiré como pesos asociados a las características, ya que lo que nos indican es el grado de importancia de dicha característica a la hora de clasificar los datos, siendo 1 el máximo de relevancia y 0 el mínimo.
Así mismo cabe destacar que nuestra intención en este problema es obtener una buena calificación de dicho vector de pesos. Esto lo medimos mediante las tasas de acierto y simplicidad que se definen como:

$$Tasa\_acierto = 100 \cdot \frac{n^o \ aciertos}{n^o \ datos} \ , \ Tasa\_simplicidad = 100 \cdot \frac{n^o \ valores \ de \ w \ < \ 0{,}2}{n^o \ de \ atributos}$$

$$Tasa\_agregada = \frac{1}{2} \cdot Tasa\_acierto + \frac{1}{2} \cdot Tasa\_simplicidad$$

La tasa de aciertos lo que nos mide es en un porcentaje cuántas instancias hemos clasificado correctamente mediante el algoritmo KNN usando el vector de pesos w.
La tasa de simplicidad nos mide cuántos de los valores que tiene el vector de pesos son menores que 0.2. Esto se hace ya que, como imposición del problema, tenemos que si alguno de los pesos es menor que 0.2 no debemos usarlo, o lo que es lo mismo, debemos sustituirlo por un 0 en la función de la distancia que luego describiré. Midiendo esto obtenemos un dato de cuanto sobreajuste ha tenido nuestro algoritmo a la hora de obtener el vector de pesos. Cuantas menos características necesitemos para discernir la clase a la que pertenece una instancia de los datos, más simple será clasificar dicha instancia. Se expresa en porcentaje indicando 0 como ninguna simplicidad y 100 como la máxima simplicidad.
De esta forma combinando ambas tasas obtenemos la tasa agregada que nos hace la media entre ambas tasas, de forma que le asignamos la misma importancia a acertar en la clasificación de las instancias y a la simplicidad en la solución. Cabe destacar que es imposible obtener una tasa de un 100 % a no ser que los datos se compongan únicamente de un punto ya que ello implicaría que la simplicidad ha de ser un 100 % (todos las posiciones del vector menores que 0.2) y por tanto la distancia sería 0 en todos los casos. De esta forma aspiraremos a una calificación lo mas alta posible pero teniendo en cuenta las restricciones de la función objetivo construida.
Las funciones y operadores de uso común los he agrupado en un fichero llamado auxiliar.py. Este fichero contiene las funciones de lectura de datos, distancias, una función que devuelve el elemento más común de una lista, la norma euclídea, una función para dividir los datos en el número de particiones que queramos manteniendo el porcentaje de elementos de cada clase que había en el conjunto de datos original y el operador de mutación.

3.1. Generación de soluciones aleatorias

En los algoritmos genéticos y meméticos partimos de una población de soluciones aleatorias que generamos con una distribución uniforme, de forma que partimos en un inicio con una

población de 30 individuos con valores en los vectores de pesos entre 0 y 1 generados de forma aleatoria.
Nótese que en nuestro caso TAM_POBLACION=30.

Algorithm 1 generaPoblacionInicial(longitud)

```
poblacion ← [ ]
for i=0 , ... , TAM_POBLACION-1 do
  cromosoma ← [ ]
  for j=0 , ... , longitud-1 do
    cromosoma ← [cromosoma,uniforme(0,1)]
  end for
  poblacion ← [poblacion,cromosoma]
end for
return poblacion
```

3.2. Operador de cruce BLX-α

Este operador de cruce se usa tanto en los algoritmos genéticos como meméticos.
Se toman dos padres, de los que hallamos el elemento más grande de su vector de pesos y el más pequeño para poder obtener el máximo y mínimo de los dos. Esto nos va a dar un intervalo de valores para nuestro hijo.
El hijo se va a generar tomando valores aleatorios con una distribución uniforme que estén en el intervalo $[min\_padres - \delta, max\_padres + \delta]$, donde $\delta = max\_padres - min\_padres$. De esta forma podemos obtener el número que deseemos de hijos tan solo con dos padres, ya que los valores son aleatorios y por tanto los hijos obtenidos serán distintos.
Nótese que en nuestro caso $\alpha = 0{,}3$

Algorithm 2 cruceBLX(cromosoma1, cromosoma2)

```
hijo ← [ ]
max_c1 ← máximo(cromosoma1)
max_c2 ← máximo(cromosoma2)
min_c1 ← mínimo(cromosoma1)
min_c2 ← mínimo(cromosoma2)
max_intervalo ← máximo(max_c1,max_c2)
min_intervalo ← mínimo(min_c1,min_c2)
delta ← (max_intervalo − min_intervalo) · α
for i=0 , ... , longitud(cromosoma1) do
  hijo ← [hijo, uniforme(min_intervalo − δ, max_intervalo + δ)]
end for
Si hay alguna posición negativa en el hijo se trunca a 0.
Si hay alguna posición mayor a 1 en el hijo se trunca a 1.
return hijo
```

Nótese que hemos tenido que truncar las soluciones, ya que el valor δ utilizado en el algoritmo puede provocar que el hijo que obtengamos tenga valores fuera del intervalo [0,1], cosa que no tendría sentido para nuestro problema.

3.3. Operador de cruce Aritmético

El operador de cruce aritmético toma, igual que en el caso anterior, dos padres y devuelve un hijo. En este caso el hijo que obtenemos es único, ya que lo vamos a calcular haciendo la media posición a posición respecto a los dos padres. Esto nos va a garantizar que los hijos estén en el intervalo de definición, no como en el BLX.
Un inconveniente que puede presentar este algoritmo es que vamos a tener mucha menos posibilidad de obtener valores muy cercanos a 0 o a 1, ya que al realizar la media siempre vamos a ir alejándonos de estos valores.

Algorithm 3 cruceAritmetico(cromosoma1,cromosoma2)

```
hijo ← [ ]
for i=0 , ... , longitud(cromosoma1) do
  hijo ← [hijo,(cromosoma1[i]+cromosoma2[i])/2]
end for
return hijo
```

3.4. Función de mutación

Esta función recibe como entrada un vector de pesos y una posición que es la que se desea mutar, devolviendo como resultado el vector de pesos ya mutado y la posición aumentada en una unidad (se usa para el algoritmo de búsqueda local aunque puede ignorarse).

Algorithm 4 mutacion(w,pos)

```
incremento = gauss(mu=0,sigma=0.3)
posicion_nueva = pos+1
w[pos]+=incremento
Truncar el vector w (0 si es negativo y 1 si es mayor que 1).
return w,pos_nueva
```

Esta función es usada en búsqueda local y en todos los genéticos y meméticos a la hora de realizar la mutación de los cromosomas.

3.5. Torneo Binario

Para la selección de los dos padres utilizamos el torneo binario. Para ello cogemos dos individuos de la población y los comparamos entre sí cogiendo al mejor de los dos. Al repetir esta operación dos veces obtenemos los dos padres que necesitamos.

Algorithm 5 torneoBinario(data,poblacion,k,etiquetas,valoraciones)

```
individuos ← 2 números aleatorios entre 0 y TAM_POBLACION-1
valoracion_ind1 ← valoraciones[individuos[0]]
valoracion_ind2 ← valoraciones[individuos[1]]
if valoracion_ind1>valoracion_ind2 then
  return individuos[0]
else
  return individuos[1]
end if
```

4. Genético Estacionario

El algoritmo genético estacionario se basa en la dinámica de poblaciones como todos los algoritmos genéticos, con la salvedad de que sólo evolucionamos y cruzamos una única pareja de padres para obtener sólo dos hijos en el caso de BLX y para el aritmético haremos lo mismo con cuatro padres para obtener dos hijos de igual forma.
Partimos de una población inicial de 30 individuos generados de forma aleatoria tal y como se ha explicado en la parte común a todos los algoritmos.
Tras la obtención de los dos hijos hacemos una mutación con probabilidad de 0.001 en cada gen (posición del vector de pesos).
Estos dos hijos generados se sumarán a la población existente. Haremos una valoración de la población viendo cuáles son los 30 individuos con mejor puntuación y nos quedaremos con ellos. Esto puede implicar que si los dos hijos generados son peores que el resto de la población desecharemos a los 2 hijos generados y nos volveremos a quedar con la población que teníamos antes de realizar el cruce.
Haciendo esto vamos a ir obteniendo a cada iteración dos individuos nuevos que se introducirán en la población mejorándola de forma gradual.
Al final del algoritmo comprobamos cuál de los elementos de la población tiene mejor valoración para devolver al mejor individuo de nuestra población.
Nótese que la constante TAM_POBLACION es 30 en nuestro caso y MAX_EVALUACIONES es 15000

Algorithm 6 GeneticoEstacionario(data,k,operador_cruce)

```
num_padres ← 0
if operador_cruce == cruceAritmetico then
  num_padres ← 4
else if operador_cruce == cruceBLX then
  num_padres ← 2
else
  Error en el operador de cruce.
end if

poblacion ← generaPoblacionInicial(numero_caracteristicas)
valoraciones ← tasa_agregada + tasa_reduccion de cada individuo de la poblacion
evaluaciones ← TAM_POBLACION
while evaluaciones < MAX_EVALUACIONES do
  padres ← Padres escogidos por torneo binario según num_padres
  hijos ← Obtenemos los hijos según operador_cruce con los padres calculados.

  Muta cada gen de los hijos si uniforme(0,1) es menor que 0.001.
  poblacion ← [poblacion,hijos]
  valoraciones ← [valoraciones,valoraciones de los hijos]
  Obtener los índices que los 30 mejores individuos de la población y quedarse con ellos.
  Actualizar poblacion y valoraciones según los índices obtenidos.
  evaluaciones ← evaluaciones+2
end while
return Devolver al individuo con mayor valoración de la población.
```

Donde el parámetro data y k se emplean en la llamada a KNN para obtener una calificación de cada individuo de la población para poder compararlos.

5. Genético Generacional

El algoritmo genético generacional se basa en la dinámica de poblaciones al igual que el estacionario. Este algoritmo, al contrario que el anterior, intenta reemplazar toda o gran parte de la población a cada iteración del mismo de forma que a cada paso mejore una parte sustancial de la misma tras los cruces.
En nuestro caso el porcentaje escogido es de un 70 % por lo que a cada iteración se generarán $0{,}7 \cdot 30 = 21$ hijos que se introducirán en la población reemplazando a los padres.
Así mismo vamos a ir guardando a cada paso el mejor individuo de cada generación con la intención de que si el peor de la población generada en esta iteración es peor que el mejor de la anterior lo reemplacemos.
Esto también nos va a proveer de una forma sencilla de reconocer al mejor individuo una vez acabemos el algoritmo.
También debemos de tener en cuenta que al igual que en el anterior algoritmo genético vamos a mutar la nueva población generada con una probabilidad de 0.001. En este caso al ser un número mayor de genes vamos a ahorrarnos la generación de números aleatorios mutando siempre un número fijo de los mismos que será $21 \cdot num\_caracteristicas \cdot 0{,}001$.

Algorithm 7 GeneticoGeneracional(data,k,operador_cruce)

```
poblacion ← generaPoblacionInicial(num_caracteristicas)
mutaciones ← PROB_MUTACION*TAM_POBLACION*num_caracteristicas
num_parejas ← TAM_POBLACION*PROB_CRUCE
valoraciones ← valoraciones de la población
mejor_solucion ← Mejor solución de la población.
while evaluaciones < MAX_EVALUACIONES do
  for i=0 , ... , num_parejas-1 do
    if operador_cruce==cruceAritmetico then
      hijos ← [ ]
      padres ← genera 4 padres con torneoBinario
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
      hijos ← [hijos,operador_cruce(padres[2],padres[3])]
      hijos ← [hijos,operador_cruce(padres[0],padres[2])]
      hijos ← [hijos,operador_cruce(padres[1],padres[2])]
    else
      hijos ← [ ]
      padres ← genera 2 padres con torneoBinario
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
    end if
    Sustituye los padres por los hijos generados
  end for
  Muta la nueva población con probabilidad 0.001 con una distribución gauss(μ = 0, σ = 0,3)

  Actualiza las valoraciones de los individuos.
  Si el peor de la nueva población es peor que el mejor de la anterior lo sustituimos.
  Actualiza el mejor de la población.
end while
return Mejor de la población.
```

6. Meméticos

El algoritmo memético toma la misma estructura que el algoritmo genético generacional en cuanto a esquema de evolución, cruce y mutaciones, con la salvedad de que introducimos una fase de explotación en el mismo.
La variante implementada toma el mismo algoritmo genético generacional y aplica al 10 % de los 30 individuos una búsqueda local para maximizar su valoración. De esta forma vamos a tener un tercio de la población con una valoración mejor que el resto. Esto lo que va a intentar hacer es maximizar y hacer la convergencia mucho más rápida que en los algoritmos genéticos.
Así mismo este algoritmo está implementado con los dos operadores de cruce con la intención de comprobar cuál es el que nos da mejores resultados con respecto al otro.
Donde prob_bl es el porcentaje de la población al que queremos aplicar la búsqueda local.

7. KNN

Cabe notar que el número que devolvemos está entre 0 y 1, por lo que en los algoritmos de valoración debemos tener esto en cuenta para multiplicarlo por 100 y convertirlo en un porcentaje.

8. Relief

Algorithm 8 Memetico(data,k,operador_cruce,nGeneraciones,prob_bl,mejores=False)

poblacion ← generaPoblacionInicial(num_caracteristicas)
mutaciones ← PROB_MUTACION*TAM_POBLACION*num_caracteristicas
num_parejas ← TAM_POBLACION*PROB_CRUCE
valoraciones ← valoraciones de la población
mejor_solucion ← Mejor solución de la población.
contador_generaciones ← 1
while evaluaciones < MAX_EVALUACIONES **do**
  if contador_generaciones %nGeneraciones==0 **then**
    n_elem_bl ← prob_bl*TAM_POBLACION
    individuos ← [ ]
    if not mejores **then**
      individuos ← Tomar n_elem_bl de forma aleatoria desde 0,...,TAM_POBLACION-1
    else
      individuos ← Toma los 0.1*TAM_POBLACION mejores de la poblacion
    end if
    for ind en individuos **do**
      Aplica la búsqueda local a poblacion[ind]
      Actualiza el número de evaluaciones.
    end for
    Actualiza las valoraciones
    Actualiza las evaluaciones.
  end if

  for i=0 , ... , num_parejas-1 **do**
    if operador_cruce==cruceAritmetico **then**
      hijos ← [ ]
      padres ← genera 4 padres con torneoBinario
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
      hijos ← [hijos,operador_cruce(padres[2],padres[3])]
      hijos ← [hijos,operador_cruce(padres[0],padres[2])]
      hijos ← [hijos,operador_cruce(padres[1],padres[2])]
    else
      hijos ← [ ]
      padres ← genera 2 padres con torneoBinario
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
      hijos ← [hijos,operador_cruce(padres[0],padres[1])]
    end if
    Sustituye los padres por los hijos generados
  end for
  Muta la nueva población con probabilidad 0.001 con una distribución $gauss(\mu = 0, \sigma = 0{,}3)$

  Actualiza las valoraciones de los individuos.
  Si el peor de la nueva población es peor que el mejor de la anterior lo sustituimos.
  Actualiza el mejor de la población.
  contador_generaciones ← contador_generaciones + 1
end while
return Mejor de la población.

Algorithm 9 KNN(w,datos_test,datos_entrenamiento, etiquetas_entrenamiento, etiquetas_test, k, mismos_conjuntos)

tam_datos_entrenamiento ← longitud(datos_entrenamiento)
clases ← []
for i=0,...,longitud(datos_test) **do**
  p ← datos_test[i]
  w_m ← Repetir el vector w tantas veces como datos haya en datos_entrenamiento.
  p_m ← Repetir el vector p tantas veces como datos haya en datos_entrenamiento.
  dist ← $w\_m \cdot (p\_m - datos\_entrenamiento)^2$
  if mismos_conjuntos **then**
    dist[i] ← ∞
  end if
  mins ← Los k índices correspondientes a las distancias más pequeñas.
  clases ← [clases, masComun(etiquetas_entrenamiento[mins])]
end for
return $\frac{Numero\ de\ elementos\ de\ clases\ que\ han\ acertado\ con\ respecto\ a\ etiquetas\_test}{longitud(etiquetas\_test)}$

Algorithm 10 elementoMinimaDistancia(e,lista)

distancias ← [ ]
for l en lista **do**
  if l!=e **then**
    distancias ← [distancias, distancia(e,l,[1..1])]
  else
    distancias ← [distancias, max(distancias)]
  end if
end for
indice_menor_distancia ← índice del elemento de menor valor del vector distancias.
return lista[indice_menor_distancia]

Algorithm 11 Relief(data)

```
w ← vector de pesos a 0
for elemento en data do
  clase ← clase de elemento
  amigos ← [ ]
  enemigos ← [ ]
  for e en data do
    if e!=elemento AND e[longitud(e)-1]==clase then
      amigos ← [amigos, e]
    else
      enemigos ← [enemigos, e]
    end if
  end for
  amigo_cercano ← elementoMinimaDistancia(elemento, amigos)
  enemigo_cercano ← elementoMinimaDistancia(elemento, enemigos)
  resta_enemigo ← element-enemigo_cercano
  resta_amigo ← element-amigo_cercano
  w ← w + resta_enemigo - resta_amigo
  w_max ← máximo de w
end for
for i en [0..longitud(w)-1] do
  if w[i]<0 then
    w[i] ← 0
  else
    w[i] ← w[i] / w_max
  end if
end for
return w
```

9. Búsqueda Local

Algorithm 12 primerVector(n)

```
w ← [ ]
for i en [0..n-1] do
  w ← [w, random.uniforme(0,1)]
end for
return w
```

Algorithm 13 busquedaLocal(data,k)

```
MAX_EVALUACIONES ← 15000
MAX_VECINOS ← 20 · longitud(data[0])
vecinos ← 0
evaluaciones ← 0
posicion_mutacion ← 0
w ← primerVector(longitud(data[0]))
valoracion_actual ← Valoracion(data,data,k,w)
while evaluaciones<MAX_EVALUACIONES AND vecinos<MAX_VECINOS do
  evaluaciones ← evaluaciones+1
  vecinos ← vecinos+1
  vecino, posicion_mutacion ← mutacion(w,posicion_mutacion)
  valoracion_vecino ← Valoracion(data,data,k,vecino)
  if valoracion_vecino>valoracion_actual then
    vecinos ← 0
    w ← vecino
    valoracion_actual ← valoracion_vecino
    posicion_mutacion ← 0
  else if posicion_mutacion==longitud(w) then
    posicion_mutacion ← 0
  end if
end while
return w
```

10. Procedimiento de desarrollo de la práctica

11. Resultados

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	0.0000	35.9375	0.4403	76.3158	0.0000	38.1579	0.0546	70.5882	0.0000	35.2941	0.2561
Partición 2	84.3750	0.0000	42.1875	0.4341	81.5789	0.0000	40.7895	0.0527	77.9412	0.0000	38.9706	0.2959
Partición 3	71.8750	0.0000	35.9375	0.4329	94.7368	0.0000	47.3684	0.0543	67.6471	0.0000	33.8235	0.3161
Partición 4	81.2500	0.0000	40.6250	0.4340	73.6842	0.0000	36.8421	0.0530	60.2941	0.0000	30.1471	0.2984
Partición 5	85.9375	0.0000	42.9688	0.4484	76.7442	0.0000	38.3721	0.0586	66.2338	0.0000	33.1169	0.2715
Media	79.0625	0.0000	39.5313	0.4380	80.6120	0.0000	40.3060	0.0546	68.5409	0.0000	34.2704	0.2876

Cuadro 1: Resultados 1NN

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	98.6301	85.2526	0.9703	78.9474	47.8261	63.3867	0.1254	75.0000	46.6667	60.8333	0.4568
Partición 2	87.5000	97.2603	92.3801	1.0143	81.5789	26.0869	53.8330	0.1267	73.5294	53.3333	63.4314	0.6407
Partición 3	71.8750	97.2603	84.5676	1.0385	86.8421	47.8261	67.3341	0.1263	67.6471	60.0000	63.8235	0.7070
Partición 4	81.2500	95.8904	88.5702	1.0530	78.9474	52.1739	65.5606	0.1254	72.0588	31.1111	51.5850	0.6594
Partición 5	85.9375	98.6301	92.2838	0.9963	72.0930	43.4783	57.7856	0.1180	72.7273	48.8889	60.8081	0.3986
Media	79.6875	97.5342	88.6109	1.0145	79.6818	43.4783	61.5800	0.1244	72.1925	48.0000	60.0963	0.5725

Cuadro 2: Resultados Relief con K=1

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
Partición 1	71.8750	34.2466	53.0608	186.6400	78.9474	26.0870	52.5172	9.0305	70.5882	35.5556	53.0719	52.5939
Partición 2	76.5625	54.7945	65.6785	436.2914	81.5789	39.1304	60.3547	6.8897	82.3529	24.4444	53.3987	92.8766
Partición 3	70.3125	30.1370	50.2247	185.3929	86.8421	17.3913	52.1167	4.3802	70.5882	57.7778	64.1830	120.7886
Partición 4	82.8125	54.7945	68.8035	354.7868	78.9474	52.1739	65.5606	21.2622	63.2353	31.1111	47.1732	70.7596
Partición 5	84.3750	39.7260	62.0505	224.3229	74.4186	30.4348	52.4267	9.3806	64.9351	44.4444	54.6898	90.2908
Media	77.1875	42.7397	59.9636	277.4868	80.1469	33.0435	56.5952	10.1886	70.3400	38.6667	54.5033	85.4619

Cuadro 3: Resultados Búsqueda Local con K=1

	Ozone				Parkinsons				Spectf-Heart			
	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)	%_clas	%_red	Agr.	T (seg)
1-NN	79.0625	0.0000	39.5313	0.4380	80.6120	0.0000	40.3060	0.0546	68.5409	0.0000	34.2704	0.2876
Relief	79.6875	97.5342	88.6109	1.0145	79.6818	43.4783	61.5800	0.1244	72.1925	48.0000	60.0963	0.5725
BL	77.1875	42.7397	59.9636	277.4868	80.1469	33.0435	56.5952	10.1886	70.3400	38.6667	54.5033	85.4619

Cuadro 4: Resultados globales con K=1