UNIVERSIDAD DE GRANADA

E.T.S. de Ingenierías Informática y de Telecomunicaciones

Práctica 2:

Problema del Aprendizaje de Pesos en Características (APC)

DECSAI

Metaheuristicas. Grupo 1: Lunes de 17:30 a 19:30

Autor:
Iván Lopez Justicia. DNI: 26514896J. email:
ivanlopez27@correo.ugr.es

Índice

1. Descripción del problema

El problema del Aprendizaje de Pesos en Características (APC) es un problema de búsqueda con codificación real en el espacion n-dimensional, para n características. Consiste en obtener un vector de pesos que permita ponderar las características asociadas a un dato con la intención de obtener un mejor porcentaje de clasificación de datos futuros.

Tenemos una muestra de datos, $X = x_1, x_2, ..., x_n$, en la que cada dato x_i está formado por un conjunto de características $c_1, c_2, ..., c_n$, además, también contiene la clase de cada dato, al final del conjunto de características. Con estos datos, tenemos que obtener un vector W de pesos $w_1, w_2, ..., w_n$ donde cada w_i pondera la característica c_i y que el vector de pesos nos permita obtener un alto porcentaje de otro conjunto de datos nuevos o desconocidos hasta la comprobación de la calidad de nuestro vector de pesos W.

En esta práctica se van a realizar dos tareas:

1. Obtener el vector de pesos (aprendizaje)
2. Valorar la calidad del vector (validación).

Para la validación vamos a utilizar la técnica de validación cruzada **5-fold cross validation**, que consiste en dividir el conjunto de datos en 5 particiones disjuntas al 20 %, con la distribución de clases equilibrada. Aprenderemos un clasificador utilizando el 80 % de los datos (4 particiones) y validaremos con el 20 % restante.

Con ésto, obtenemos 5 valores de porcentaje de clasificación en el conjunto de prueba, uno para cada partición. La calidad del método de clasificación se medirá con el valor correspondiente a la media de los 5 porcentajes de clasificación del conjunto prueba.

Como clasificador se ha implementado el kNN, utilizando k = 1, el cual le asigna a cada dato la clase del vecino más cercano a dicho dato. El vecino más cercano será aquel con el que se obtenga menor distancia euclídea calculada de la siguiente forma:

$$d_e(e_1, e_2) = \sqrt{\sum_n^{i=1}(e_1^i - e_2^i)^2}$$

La medida de validación será la tasa de acierto del clasificador 1-NN sobe el conjunto de prueba. El resultado final será la media de los 5 valores obtenidos, uno para cada ejecución del algoritmo. Para la comparación de algoritmos utilizaremos cuatro estadísticos diferentes denominados *Tasa_clas*, *Tasa_red*, *Agregado* y *Tiempo*.

Los algoritmos implementados son el clasificador 1-NN, el algoritmo Relief, un algoritmo de busqueda local, cuatro algoritmos genéticos y tres meméticos. Los conjuntos de datos a utilizar serán *parkinsons*, *ozone-320* y *spectf-heart*. Hay que tener en cuenta que éstos datos hay que normalizarlos, lo que se realiza automáticamente tras cargar los datos.

2. Descripción de la aplicación de los algoritmos empleados al problema

Como solución a nuestro problema utilizaremos un vector de pesos, W, que permite realizar una clasificación de calidad. Para nuestro problema, utilizaremos un vector de n componentes (n es el nº de características de cada dato del conjunto. En cada componente tendremos un valor real en el rango [0,1]. El valor 1 en una componente w_i indica que esa característica se considera completamente al calcular la distancia, si es el valor 0, no se considera, y si es un valor intermedio gradúa el peso asociado a cada característica y pondera su importancia.

Para algoritmos que traten una población de soluciones, como en los genéticos, utilizaremos un vector de vectores solución.

Como función objetivo mediremos el rendimiento del clasificador 1-NN con los pesos calculados por cada algoritmo. El procedimiento será:

1. Se ejecuta el algoritmo para unos datos de entrenamiento, *train* y obtenemos W.
2. Ejecutamos el clasificador con un conjunto de prueba, *test*, y el vector obtenido en el paso anterior, devolviendo el conjunto de etiquetas, *clasificacion*, asociado al conjunto *test*.
3. Comparamos las etiquetas del clasificador con las reales del conjunto, y medimos el porcentaje de acierto.

Todos los datos necesarios están alojados en la carpeta *Data*, de la que se cargan al ejecutar el programa.

Algorithm 1 Clasificador 1-NN

Require: Conjunto entrenamiento (*train*, clases del conjunto (*clases*, conjunto de test (*test*, vector de pesos (*pesos*)

1: **for** (i = 0 ; i <test.size() ; i++) **do**
2: $vecino_mas_cercano = vecinoMasCercano(train, test[i], i, pesos)$
3: $clasificacin[i] = clases[vecino_mas_cercano]$
4: **end for**

El cálculo del vecino más cercano se realiza utilizando la *distancia Euclídea*. El vecino elegido será el de consiga una mejor distancia con respecto al dato que estamos clasificando.

Algorithm 2 VecinoMasCercano

Require: Conjunto de datos (*data*), dato actual (*actual*, posicion del dato (*posicion*), vector de pesos (*pesos*)

```
1: mejor_dist = 99999
2: for (i = 0 ; i <data.size() ; i++) do
3:     if i ≠ posicion then
4:         dist_actual = distanciaEuclidea(datos[i], actual)
5:         if dist_actual < mejor_dist then
6:             mejor_dist = dist_actual
7:             vecino_mas_cercano = i
8:         end if
9:     end if
10: end for
11: return vecino_mas_cercano
```

Una vez obtenida la clasificación de los datos de *test* en función del vector de pesos calculado, debemos evaluar la calidad de la clasificación. Ésto lo hacemos comparando la clasificación obtenida con la real obtenida del conjunto de datos y ver que porcentaje se han clasificado correctamente, tal y como nos indica la siguiente fórmula *tasa_clas*. A mayor tasa de clasificación, mejor es el vector de pesos generado.

$$tasa\_clas = 100 \cdot \frac{n^{o}instanciasbienclasificadas}{n^{o}instanciastotales}$$

También vamos a evaluar la tasa de reducción asociada al número de características utilizadas por el clasificador con respecto al número total de características. Ésta se calcula tal y como nos indica la formula *tasa_red*.

$$tasa\_red = 100 \cdot \frac{n^{o}valoresw_i<0,2}{n^{o}caractersticas}$$

Por último, calcularemos una agregación que combina ámbos resultado en único valor. La función objetivo será:

$$F(W) = \alpha \cdot tasa\_clas(W) + (1 - \alpha) \cdot tasa\_red(W)$$

Donde:

1. $W = (w_i, ..., w_n)$ es una solución al problema que consiste en un vector de números reales entre 0 y 1 de tamaño n que define el peso que pondera o filtra a cada característica.

2. 1-NN es el clasificador k-NN con k=1 vecino generado a partir del conjunto de datos inicial utilizando los pesos en W que se asocian a las n características.

3. α pondera la importancia entre el acierto y la reducción de la solución encontrada.

Todos los algoritmos parten de una solución inicial que se genera de manera aleatoria, creando un vector de pesos para cada componente con un numero real entre 0 y 1. Sea cual sea el algoritmo debemos generar nuevas soluciones o modificar las que tenemos, por lo que debemos tener un operar de generación de vecino/mutación. Para ello modificamos el gen i del cromosoma j siguiiente una distribución normal de media 0 y varianza σ^2. Hay que tener en cuenta que esta distribución puede genera valores negativos, los cuales debemos truncar a 0 ya que no tiene sentido tener un peso por debajo de 0. Para ello utilizamos la siguiente función:

Algorithm 3 Truncar

Require: numero

$salida = numeor$
if $numero < 0$ **then**
    $salida = 0$
else if $numero > 1$ **then**
    $salida = 1$
end if
return $salida$

Para la búsqueda local se ha implementado el siguiente algoritmo:

Algorithm 4 LocalSearch

Require: train, clases_train, pesos, num_eval

1: $indices = \{1, 2, 3, ..., num_caracteristicas\}$
2: $pesos = solIncialAleatoria(num\_caracteristicas)$
3: $KNN(train, clases\_train, train, clasificacion, pesos)$
4: $porcentaje_ant = tasaAgregacion(clases\_train, clasificacion, pesos, 0{,}5$
5: $i\_aux = 0$
6: **while** $num\_eval < 15000 \&\& num\_vecinos < 20 num\_caracteristica$ **do**
7: $mejora = false$
8: **for** $i = 0; i < num\_caracteristicas \&\& !mejora; i++$ **do**
9: $sol\_nueva = pesos$
10: $elegido = random \in [i\_auxiliar, indices.size() - 1]$
11: $posicion\_auxiliar = indices[i\_auxiliar]$
12: $indices[i\_auxiliar] = incides[elegido]$
13: $indices[elegido] = posicion\_auxiliar$
14: $modificarPesos(sol\_nueva, indices[i\_auxiliar])$
15: $num\_vecinos++$
16: $i\_auxiliar++$
17: **if** $i\_auxiliar == num\_caracteristicas$ **then**
18: $i\_auxiliar = 0$
19: **end if**
20: $KNN(train, clases\_train, train, clasificacion, sol\_nueva)$
21: $porcentaje\_nuevo = tasaAgregacion(clases\_train, clasificacion, sol\_nueva, 0{,}5)$
22: num_eval++
23: **if** $porcentaje\_nuevo > porcentaje\_ant$ **then**
24: $pesos = sol\_nueva$
25: $porcentaje\_ant = porcentaje\_nuevo$
26: $mejora = true$
27: $num\_vecinos = 0$
28: **end if**
29: **end for**
30: **end while**
31: **return** $pesos$

En esta búsqueda local utilizamos la búsqueda de primero el mejor. *indices* nos indica en que orden se van a modificar las comopnentes. En cada paso queremos modificar una componenete aleatoria que no hayamos modificado antes, es decir, que esté desde la última que se modificó en *índices, i_auxiliar* y la última que se modificó, *indices.size() - 1*. Así tenemos una forma aleatoria de modificar las componentes.

Algorithm 5 SolInicialAleatoria

Require: $size$

1: **for** $i = 0; i < size; i++$ **do**
2: $\quad solucion\_inicial[i] = random \in [0, 1]$
3: **end for**
4: **return** $solucion\_inicial$

Algorithm 6 ModificarPeso

Require: $cromosoma, gen\_mutar$

1: $cromosoma[gen\_mutar] += random \in distribucion\_normal(0, 0{,}09)$
2: $numero = truncar(cromosoma[gen\_mutar])$
3: $cromosoma[gen\_mutar] = numero$

Con la tasa de agregación se calcula como de buena es una solución en función de su tasa de acierto y reducción.

Algorithm 7 TasaAgregacion

Require: $correctas, calculadas, solucion, alpha$

1: $porcentaje = tasaAcierto$
2: $reduccion = tasaReduccion(solucion$ **return** $alpha * porcentaje + (1 - alpha) * reduccion$

La tasa de acierto calcula cuantas etiquetas calculadas por nuestro clasificador 1-NN en función del vector de pesos son correctas.

Algorithm 8 TasaAcierto

Require: $correctas, calculadas$

1: **for** $i = 0; i < num\_caracteristicas; i++$ **do**
2: $\quad$ **if** $correctas[i] == calculadas$ **then**
3: $\quad\quad correctos++$
4: $\quad$ **end if**
5: **end forreturn** $(correctos * 1{,}0/solucion.size()) * 100{,}0$

La tasa de reducción calcula cuantos pesos son menores que 0.2.

Algorithm 9 TasaReduccion

Require: $solucion$

1: **for** $i = 0; i < solucion.size(); i++$ **do**
2: $\quad$ **if** $solucion[i] < 0{,}2$ **then**
3: $\quad\quad reducidos++$
4: $\quad$ **end if**
5: **end forreturn** $(reducidos * 1{,}0/solucion.size()) * 100{,}0$

3. Pseudocódigo de los algoritmos

Algorithm 10 EnfriamientoSimulado

Require: $train, clases\_train, pesos, num\_eval$

$num\_caracteristicas = train[0].size()$
$max\_vecinos = 10 * num\_caractersticas$
$max\_exitos = 0{,}1 * max\_vecinos$
$num\_enfriamientos = 15000/(max\_vecinos)^2$
$solucion = solInicialAleatoria(num\_caracteristicas)$
$KNN(train, clases\_train, clasificacion, solucion$
$tasa_actual = tasaAcierto(clases\_train, clasificacion)$
num_eval++
$mejor\_solucion = solucion$
$mejor\_tasa = tasa\_actual$
$t\_ini = (0{,}3 * (mejor\_tasa/100{,}0)/(-log(0{,}3))$
$t\_fin = 0{,}001$
while $t\_fin > t\_ini$ **do**
    $t\_fin = t\_ini * 0{,}001$
end while
$beta = (t\_ini - t\_fin)/(num\_enfriamietos * t\_ini * t\_fin)$
$t\_actual = t\_ini$
while $num\_exitos > 0 \&\& num\_eval < 15000 \&\& t\_actual > t\_fin$ **do**
    $num\_exitos = 0$
    $vecino = 0$
    while $num\_exitos < max\_exitos \&\& vecinos < max\_vecinos$ **do**
        $sol\_nueva = solucion$
        $gen = random \in [0, num\_caracteristicas]$
        $mutarSolucion(sol\_nueva, gen)$
        num_eval++
        $vecinos++$
        $KNN(train, clases\_train, train, sol\_nueva)$
        $tasa\_nueva = tasaAgregacion(clases\_train, clasificacion, sol\_nueva, 0{,}5)$
        $mejora = tasa\_nueva - tasa\_actual$
        if $mejora/100{,}0 > 0 || random \in [0, 1] < exp(-(mejora/100{,}0)/t\_actual/100{,}0))$ **then**
            $tasa\_actual = tasa\_nueva$
            $num\_exitos++$
            $solucion = sol\_nueva$
            if $tasa\_nueva > mejor\_tasa$ **then**
                $mejor\_solucion = sol\_nueva$
                $mejor\_tasa = tasa\_actual$
            end if
        end if
    end while
    $t\_actual = t\_actual/(1 + beta * t\_actual)$
end whilereturn $mejor\_solucion$

La temperatura inicial se calcula como:

$$T_0 = \frac{\mu C(S_0)}{-ln(\phi}$$

donde T_0 es la temperatura inicial, $C(S_0)$ es el coste de la solución inicial y $\phi =$ 0,3. La temperatura final, T_f se fija a 10^{-3}. Como esquema de enfriamieto se usará el esquema de Cauchy modificado:

$$T_{k+1} = \frac{T_k}{1+\beta T_k} \; \beta = \frac{T_0 - T_f}{M T_0 T_F}$$

donde M es el número de enfriamientos a realizar.

Algorithm 11 ILS

Require: $train, clases\_train, pesos, num\_eval$

$num\_caracteristicas = train[0].size$
$indices = \{1, 2, 3, ..., num\_caracteristicas\}$
$solucion = solInicialAleatoria(num\_caracteristicas)$
$KNN(train, clases\_train, clasificacion, solucion$
$tasa_actual = tasaAcierto(clases\_train, clasificacion)$
$mejor\_solucion = solucion$
$tasa\_mejor = tasa\_actual$
$num\_mutaciones = 0{,}1 * num\_caracteristicas$
$localSearchILS(train, clases\_train, solucion)$
for $i = 0; i < 14; i++$ **do**
    for $i = 0; i < num\_caracteristicas \&\& !mejora; i++$ **do**
        $elegido = rand()\,\%(indices.size() - i) + i$
        $posicion\_auxiliar = indices[i]$
        $indices[i] = indices[elegido]$
        $indices[elegido] = posicion\_auxiliar$
    end for
    for $i = 0; i < num\_mutaciones \&\& !mejora; i++$ **do**
        $mutarPosicion(solucion, indices[i]$
    end for
    $localSearchILS(train, clases\_train, train, solucion)$
    $KNN(train, clases\_train, train, clasificacion, solucion)$
    $tasa\_actual = tasaAcierto(clases\_train, clasificacion)$
    if $tasa\_actual > tasa\_mejor$ **then**
        $mejor\_solucion = solucion$
        $tasa\_mejor = tasa\_actual$
    end if
    $solucion = mejor\_solucion$
end for// return $mejor\_solucion$

Algorithm 12 LocalSearchILS

Require: train, clases_train, pesos, num_eval

$indices = \{1, 2, 3, ..., num_caracteristicas\}$
$pesos = solIncialAleatoria(num\_caracteristicas)$
$KNN(train, clases\_train, train, clasificacion, pesos)$
$porcentaje_ant = tasaAgregacion(clases\_train, clasificacion, pesos, 0{,}5$
$i\_aux = 0$
while $num\_eval < 1000$ **do**
    $mejora = false$
    for $i = 0; i < num\_caracteristicas \&\& !mejora; i++$ **do**
        $sol\_nueva = pesos$
        $elegido = random \in [i\_auxiliar, indices.size() - 1]$
        $posicion\_auxiliar = indices[i\_auxiliar]$
        $indices[i\_auxiliar] = incides[elegido]$
        $indices[elegido] = posicion\_auxiliar$
        $modificarPesos(sol\_nueva, indices[i\_auxiliar])$
        $num\_vecinos++$
        $i\_auxiliar++$
        if $i\_auxiliar == num\_caracteristicas$ **then**
            $i\_auxiliar = 0$
        end if
        $KNN(train, clases\_train, train, clasificacion, sol\_nueva)$
        $porcentaje\_nuevo = tasaAgregacion(clases\_train, clasificacion, sol\_nueva, 0{,}5)$
        num_eval++
        if $porcentaje\_nuevo > porcentaje\_ant$ **then**
            $pesos = sol\_nueva$
            $porcentaje\_ant = porcentaje\_nuevo$
            $mejora = true$
            $num\_vecinos = 0$
        end if
    end for
end while
return $pesos$

En la búsqueda local que hemos usado en ILS se usa el siguiente operador de mutación, para el cual utilizamos una distribución normal de media 0 y $\sigma^2 = 0{,}16$.

Algorithm 13 MutarPosicion

Require: $solucion, posicion$

$solucion[posicion] += random \in distribucion\_normal(0, 0{,}16)$
$numero = truncar(solucion[posicion])$
$solucion[posicion] = numero$

Algorithm 14 MutarPosicion

Require: $train, clases_train$

$crossover = 0{,}5$
$indices = \{1, 2, 3..., num\_caracteristicas\}$
$poblacion = poblacionInicial(num\_caracteristicas, 50)$
$evalPoblacion(train, clases\_train, poblacion, pcts\_poblacion, num\_eval)$
while $num\_eval < 15000$ **do**
  for $i = 0; i < poblacion.size(); i++$ **do**
    $indices\_generados = 0$
    $i\_aux = 0$
    while $indices\_generados < 4$ **do**
      $elegido = random \in [i\_aux, indices.size() - 1$
      $posicion\_auxiliar = indices[i\_aux]$
      $indices[i\_aux] = incides[elegido]$
      $indices[elegido] = posicion\_auxiliar$
      if $indices[i\_aux]! = i$ **then**
        $indices\_generados++$
        i_aux++
      end if
    end while
    $padre1 = poblacion[indices[0]]$
    $padre2 = poblacion[indices[1]]$
    $padre3 = poblacion[indices[2]]$
    $gen\_elegido = random \in [0, num\_caracteristicas]$
    for $j = 0; j < num\_caracteristicas; j++$ **do**
      $random = random \in [0, 1]$
      if $random < crossover || gen\_elegido == j$ **then**
        $numero = padre1[j] + 0{,}5 * (padre2[j] - padre3[j])$
        $numero = truncar(numero)$
        $hijo[j] = numero$
      else
        $hijo[j] = poblacion[i][j]$
      end if
      $hijos[i] = hijo$
    end for
  end for
  $evalPoblacion(train, clases\_train, hijos, pcts\_hijos, num\_eval$
  for $i = 0; i < poblacion.size(); i++$ **do**
    if $pcts\_poblacion[i] < pcts\_hijos[i]$ **then**
      $poblacion[i] = hijos[i]$
      $pcts\_poblacion[i] = pcts\_hijos[i]$
    end if
  end for
end while
$max = 0$
$pos\_aux = 0$
for $i = 0; i < pcts\_poblacion.size(); i++$ **do**
  if $pcts\_poblacion[i] > max$ **then**
    $max = pcts\_poblacion[i]$
    $pos\_max = i$
  end if
end for// return $poblacion[pos\_max$

Algorithm 15 DE/current-to-best/1

Require: $train, clases_train$

$crossover = 0{,}5$
$indices = \{1, 2, 3..., num\_caracteristicas\}$
$index\_mejores = \{1, 2, 3..., num\_caracteristicas\}$
$poblacion = poblacionInicial(num\_caracteristicas, 50)$
$evalPoblacion(train, clases\_train, poblacion, pcts\_poblacion, num\_eval)$
$sort(index\_mejoresenfunciondepcts\_poblacion$
$best\_padre = poblacion[index\_mejores[0]]$
while $num\_eval < 15000$ **do**
  for $i = 0; i < poblacion.size(); i++$ **do**
    $indices\_generados = 0$
    $i\_aux = 0$
    while $indices\_generados < 3$ **do**
      $elegido = random \in [i\_aux, indices.size() - 1$
      $posicion\_auxiliar = indices[i\_aux]$
      $indices[i\_aux] = incides[elegido]$
      $indices[elegido] = posicion\_auxiliar$
      if $indices[i\_aux]! = i$ **then**
        $indices\_generados++$
        i_aux++
      end if
    end while
    $padre1 = poblacion[indices[0]]$
    $padre2 = poblacion[indices[1]]$
    $gen\_elegido = random \in [0, num\_caracteristicas]$
    for $j = 0; j < num\_caracteristicas; j++$ **do**
      $random = random \in [0, 1]$
      if $random < crossover || gen\_elegido == j$ **then**
        $numero = poblacion[i][j] + 0{,}5 * (best\_padre[j] - poblacion[i][j]) + 0{,}5 * (padre1[j] - padre2[j]$
        $numero = truncar(numero)$
        $hijo[j] = numero$
      else
        $hijo[j] = poblacion[i][j]$
      end if
      $hijos[i] = hijo$
    end for
  end for
  $evalPoblacion(train, clases\_train, hijos, pcts\_hijos, num\_eval$
  for $i = 0; i < poblacion.size(); i++$ **do**
    if $pcts\_poblacion[i] < pcts\_hijos[i]$ **then**
      $poblacion[i] = hijos[i]$
      $pcts\_poblacion[i] = pcts\_hijos[i]$
    end if
  end for
  $sort(index\_mejoresenfunciondepcts\_poblacion)$
  $best\_padre = poblacion[index\_mejores[0]]$
end while
return $poblacion[index\_mejores[0]]$

4. Descripción en pseudocódigo del algoritmo de comparación

Obtenemos los pesos utilizando un algoritmos anteriores, llamamos al clasificador 1-NN con los datos de *train* como conjunto de entrenamiento y *test* como conjunto de prueba y así conseguimos la clasificación para dichos datos. Despues calculamos la tasa de agregación, comparando la clasificación obtenida por nuestros algoritmos con la clasificación real para obtener un porcentaje de acierto y calculando el número de pesos que han sido reducidos.

Algorithm 16 DE/current-to-best/1

Require: $train, clases_train$

1: $pesos = algoritmoacomparar(train, clases\_train)$
2: $KNN(train, clasees\_train, test, clasificacion, pesos)$
3: $porcentaje = tasaAgregacion(clases\_test, clasificacin, pesos, 0{,}5)$

5. Desarrollo

El código empleado en ésta práctica ha sido desarrollado por mí tomando como apoyo el material de teoría y de los seminarios. También he he consultado páginas várias por internet, tanto para ideas de implementación como consultas sobre funciones de diferentes librerias (cplusplus, por ejemplo). La práctica la he desarrollado en C++. Para su compilación aporto un makefile que se encarga de realizar todo el proceso. Para ejecutarlo basta con ejecutar el comando *./bin/main*.

6. Experimentos y análisis de resultados

Los conjuntos de datos que se han utilizado en la práctica son:

- **Parkinsons:** Conjunto de datos orientado a 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. 195 ejemplos con 22 características que deben ser clasificados en 2 clases.
- **Ozone:** Conjunto de datos para la detección del nivel de ozono según las mediciones realizadas a lo largo del tiempo. 320 ejemplos (seleccionados de los 2536 originales, eliminando los ejemplos con valores perdidos y manteniendo la distribución de clases al 50 %) con 73 características que deben ser clasificados en 2 clases.
- **Spectf-heart:** Conjunto de datos de detección de enfermedades cardiacas a partir de imágenes médicas de tomografía computerizada (SPECT) del corazón de pacientes. 267 ejemplos con 44 características que deben ser clasificados en 2 clases.

Todos los datos han sido normalizados en el rango [0,1] y la semilla que he utilizado ha sido un número elegido al azar, *65423174*.

Por algún motivo que no he logrado descubrir no he podido leer el conjunto Spectf-heart, he intentado conseguir leerlo de todos los modos que se me han ocurrido, pero por alguna razón entra en bucle infinito al leer ese archivo, por ello,he decidido utilizar solo Ozone y Parkinsons.

6.1. Resultados

Tabla 1: Resultados obtenidos por el algoritmo K-NN en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	73,44	0,00	36,72	0,00	71,79	0,00	35,90	0,00
Partición 2	42,19	0,00	21,09	0,00	41,03	0,00	20,51	0,00
Partición 3	25,00	0,00	12,00	0,04	72,65	0,00	36,32	0,00
Partición 4	20,31	0,00	10,15	0,00	35,90	0,00	17,95	0,00
Partición 5	87,19	0,00	43,59	0,00	14,36	0,00	7,18	0,00
Media	49,63	0,00	24,71	0,01	47,15	0,00	23,57	0,00

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

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	20,83	23,94	22,39	0,00	15,38	4,76	1,07	0,00
Partición 2	26,11	14,08	20,10	0,00	19,05	4,76	11,07	0,00
Partición 3	28,12	9,85	18,99	0,00	19,87	4,76	12,32	0,00
Partición 4	21,07	19,72	25,40	0,00	39,32	4,76	22,04	0,00
Partición 5	33,60	26,77	30,17	0,00	34,62	14,29	24,45	0,00
Media	25,95	18,87	23,41	0,00	25,65	6,67	14,19	0,00

Tabla 3: Resultados obtenidos por el algoritmo Enfriamiento Simulado en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	33,95	21,43	27,69	1,26	54,01	5,26	29,63	0,06
Partición 2	35,80	17,14	26,48	1,31	74,06	15,79	44,92	0,02
Partición 3	81,50	11,42	46,46	1,21	76,07	21,05	48,56	0,02
Partición 4	37,95	28,57	33,26	1,20	76,11	15,79	45,95	0,02
Partición 5	38,75	12,86	25,80	1,26	76,92	26,32	51,62	0,02
Media	45,59	18,28	31,94	1,25	71,43	16,84	44,14	0,03

Tabla 5: Resultados obtenidos por el algoritmo DE/Rand/1 en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	42,33	100,00	71,17	254,70	74,35	100,00	87,18	25,92
Partición 2	42,97	100,00	71,49	25,98	39,74	100,00	69,87	26,18
Partición 3	42,80	100,00	71,40	253,07	29,05	100,00	64,53	26,95
Partición 4	43,00	100,00	71,71	254,94	52,56	100,00	76,28	25,87
Partición 5	44,06	100,00	72,03	253,47	14,36	100,00	87,18	25,84
Media	43,03	100,00	71,56	208,43	42,01	100,00	77,01	26,15

Tabla 5: Resultados obtenidos por el algoritmo DE/Rand/1 en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	42,33	100,00	71,17	254,70	74,35	100,00	87,18	25,92
Partición 2	42,97	100,00	71,49	25,98	39,74	100,00	69,87	26,18
Partición 3	42,80	100,00	71,40	253,07	29,05	100,00	64,53	26,95
Partición 4	43,00	100,00	71,71	254,94	52,56	100,00	76,28	25,87
Partición 5	44,06	100,00	72,03	253,47	14,36	100,00	87,18	25,84
Media	43,03	100,00	71,56	208,43	42,01	100,00	77,01	26,15

Tabla 6: Resultados obtenidos por el algoritmo DE/current-to-best/1 en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	43,69	80,60	62,14	245,84	25,21	100,00	62,61	24,83
Partición 2	44,33	74,62	59,48	251,23	13,55	95,24	54,40	24,48
Partición 3	44,03	67,16	55,60	251,84	19,23	90,47	54,85	24,48
Partición 4	44,55	71,64	58,10	252,08	22,22	95,24	58,73	24,31
Partición 5	44,90	77,61	61,26	255,05	20,51	90,48	55,49	24,11
Media	44,30	74,33	59,31	251,21	20,14	94,29	57,22	24,44

Tabla 5.2: Resultados globales en el problema del APC

	Ozone				Parkinsons			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	49,63	0,00	24,71	0,01	47,15	0,00	23,57	0
RELIEF	27,95	18,87	23,41	0	34,62	6,66	31,66	0
ES	45,58	18,28	31,93	1,25	71,43	16,84	44,13	0,00
ILS	10,62	25,51	10,06	267,79	81,90	26,55	53,72	22,05
DE/Rand/1	43,03	100,00	71,56	208,43	42,02	100,00	71,00	26,15
DE/current-to-best/1	44,30	74,33	59,31	251,21	20,17	94,31	57,23	24,66

Ya que tenemos todos los datos recogidos, antes de nada quiero comentar que los tiempos obtenidos han sido obtenidos aplicando optimización en tiempo de compilación, de nivel -O2. La ejecución total del programa, incluyendo la ejecución de todos los algoritmos para todos los conjuntos de datos, ha tomado un total de 3739,017 segundos, es decir 1 hora y 2 minutos.

Según los datos recogidos en la tabla de medias podemos ver que los mejores resultados para los conjuntos de datos que tenemos los conseguimos con los algoritmos basados en poblaciones. El algoritmo de *Evolución Differencial* con cruce y mutación aleatoria ha obtenido los mejores resultados.

Comparando los algoritmos, podemos ver como K-NN y Relief obtienen unos peores resultados comparandolos con los algoritmos de Evolución Diferencial, ES, e ILS. A mi parecer, me resultan extraños los resultados obtenidos por los algoritmos K-NN, Relief y ES. Es posible que en algún paso de mi immplementación hay algo que no es correcto o algo así. De todos modos los buenos resultados de Evolución Diferencial si son más coherentes.