Universidad de Granada

PRÁCTICA 3:
Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial para el Problema del Aprendizaje de Pesos en Características

Metaheurísticas.
Grupo 2. Martes 17:30-19:30

Realizado por:
Francisco Solano López Rodríguez
DNI: 20100444P
Email: fransol0728@correo.ugr.es

DOBLE GRADO EN INGENIERÍA INFORMÁTICA Y MATEMÁTICAS.
CUARTO CURSO

Índice

1. Descripción del problema.

El problema del APC o Aprendizaje de Pesos en Características consiste en optimizar el rendimiento de un clasificador mediante la obtención de un vector de pesos con el que ponderar las carácteristicas de un objeto. Tendremos un conjunto de objetos $\{O_1, ...O_m\}$ donde cada O_i tiene unas características asociadas $\{a_1, ...a_n, C\}$, en las cuales las primeras n características son atributos del objeto y la última característica C es la clase a la que pertenece el objeto. El vector de pesos con el que pretendemos ponderar dichos atributos vendrá dado por $\{w_1, ..., w_n\}$, donde cada w_i pertenece al intervalo $[0, 1]$.

Vamos a usar el clasificador 1-NN, es decir la clase que vamos a asignar al objeto a clasificar es la del vecino más cercano. Para determinar la distancia de un objeto a otro usaremos la siguiente distancia:

$$d_e(e_1, e_2) = \sqrt{\sum_i w_i(e_1^i - e_2^i)^2 + \sum_j w_j d_h(e_1^j, e_2^j)}$$

Donde d_h corresponde a la distancia de Hamming para el caso de variables nominales, aunque en los conjuntos de datos que vamos a utilizar nosotros todas las variables son numéricas.

El objetivo será obtener un sistema que nos permita clasificar nuevos objetos de manera automática. Usaremos la técnica de validación cruzada 5-fold cross validation. Para ello dividiremos en 5 particiones disjuntas al 20 %, con la distribución de clases equilibrada. Aprenderemos el clasificador utilizando 4 de las particiones y validaremos con la partición restante. Este proceso se puede realizar de 5 formas diferentes con lo que obtendremos un total de 5 valores de porcentaje de clasificación en el conjunto de prueba.

Buscaremos optimizar tanto a precisión como la complejidad del clasificador. La función que queremos maximizar es la siguiente:

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

En donde *tasa_clas* y *tasa_red* corresponden a las siguientes funciones:

$$tasa\_clas = 100 \cdot \frac{n^o\ instancias\ bien\ clasificadas\ en\ T}{n^o\ instancias\ en\ T}$$

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0{,}2}{n^o\ caracteristicas}$$

T es el conjunto de objetos a clasificar, $\alpha \in [0, 1]$ pondera la importancia entre el acierto y la reducción de la solución encontrada y W es el vector de pesos.

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

Una solución en el problema del Aprendizaje de Pesos en Características es un vector de pesos $W = \{w_1, w_2, \cdots w_n\}$ con los que ponderar la distancia entre 2 objetos. cada valor $w_i \in [0, 1]$, en donde si el valor es menor de 0,2 la característica no se tiene en cuenta para el cálculo de la distancia, si es igual a 1 se tiene totalmente en cuenta y un valor intermedio ponderara la importancia de dicha característica.

La distancia entre dos objetos $e1$ y $e2$ ponderada por el vector W viene dada por la siguiente función:

```
1  Funcion distancia(e^1, e^2, w)
2      sum = 0
3
4      Para i = 0 hasta w.size
5          Si w_i >= 0,2
6              sum = sum + w_i * (e_i^1 - e_i^2)^2
7
8      Devolver sqrt(sum)
9  Fin
```

Antes de utilizar los datos para la ejecución de los algoritmos, estos han sido normalizados. La función es la siguiente:

```
1  Funcion Normalizar(T)
2      Para i = 0 hasta num_atrib-1
3          max = min = T[0][i]
4
5          Para j = 1 hasta T.size
6              Si max < T[j][i]
7                  max = T[j][i]
8              Si min > T[j][i]
9                  min = T[j][i]
10
11         Para j = 0 hasta T.size
12             T[j][i] = (T[j][i]-min)/(max-min)
13 Fin
```

Al inicio del programa después de haber leído los datos y haberlos normalizado, pasamos a crear las particiones con las que realizaremos la técnica de validación cruzada 5-fold cross validation. Las particiones se crearan de forma que se mantengan en cada una la misma proporción de cada clase que había el conjunto original, para ello se insertaran primero los elementos de la clase 1 de forma rotatoria, y después los elementos de la clase 2.

```
1  Funcion crearParticiones()
2      j = 0
3      Para i = 0 hasta N
4          Si e[i][num_atrib-1] == 1
5              particiones[j%5].insertar(e[i])
6              j = j + 1
7      Para i = 0 hasta N
8          Si e[i][num_atrib-1] == 2
9              particiones[j%5].insertar(e[i])
```

```
10            j = j + 1
11    Fin
```

Para clasificar un nuevo dato e_{new} usaremos el algoritmo 1-NN con distancia ponderada por un vector de pesos. Viene dada por el siguiente pseudocódigo, en donde el parámetro out indica el indice del elemento que vamos a dejar fuera en el 'leave one out'.

```
1   Funcion KNN(T, new_e, w, out = -1)
2       d_min = 9999999
3
4       Para i = 0 hasta T.size
5           Si out != i
6               d = distancia(T[i], new_e, w)
7               Si d < d_min
8                   c_min = T[i][T[i].size-1]
9                   d_min = d
10
11      Devolver c_min
12  Fin
```

La función objetivo es la combinación con pesos de la tasa de acierto y la complejidad del clasificador donde el valor de α considerado vale 0,5. El objetivo es maximizar dicha función. El parámetro Data corresponde al conjunto de datos sobre el que clasificaremos y T el conjunto de datos que pretendemos clasificar, el parámetro leave_one_out es un booleano que indica si se realizará dicha técnica, la cual será necesaria para clasificar al conjunto de entrenamiento.

```
1   Funcion F(Data, T, w, leave_one_out)
2       alpha = 0.5
3       Devolver alpha*tasaClas(Data, T, w, leave_one_out) + (1-alpha)*tasaRed(w)
4   Fin
```

La función tasaClas calcula la tasa de acierto del clasificador contando el número de aciertos y devolviendo el porcentaje de acierto que ha tenido.

```
1   Funcion tasaClas(Data, T, w, leave_one_out)
2       clasify_ok = 0
3
4       Para i = 0 hasta T.size
5           Si leave_one_out = true
6               out = i
7           Si No
8               out = -1
9
10          Si clasify(Data, T[i], w, out) = T[i][T[i].size-1]
11              clasify_ok = clasify_ok + 1
12
13      Devolver 100.0*clasify_ok/T.size
14  Fin
```

La función tasaRed calcula la tasa de reducción de características con respecto al conjunto original, para ello cuenta el número de elementos del vector de pesos cuyo valor esta por debajo de 0,2, los cuales no serán tomados en cuenta en el cálculo de la distancia.

```
Funcion tasaRed(w)
    num = 0

    Para i = 0 hasta w.size
        Si w_i < 0,2
            num = num + 1

    Devolver 100.0*num/w.size
Fin
```

La generación de un vecino se realizará mediante la alteración de una componente del vector de pesos W.

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

Donde $z_i \sim N(0; 0{,}4)$, es decir es un valor aleatorio que sigue una distribución normal de media 0 y varianza 0,4. Si el valor de w_i queda fuera de su dominio lo trucamos a $[0, 1]$.

```
Funcion nuevoVecino(w, i)
    z = aleatorio ~ Normal(0, 0.4)
    w_i = w_i + z

    Si w_i > 1
        w_i = 1
    Si w_i < 0
        w_i = 0

    Devolver w
Fin
```

Para las poblaciones de soluciones en los algoritmos genéticos se ha usado una estructura llamada cromosoma que almacena un vector solución w, junto con el valor de la función objetivo obtenido con dicho vector w. La población estará formada por un vector de cromosomas.

```
    cromosoma{w, valor}
```

A continuación se muestra la función utilizada para generar una población de soluciones aleatorias.

```
Funcion generarPoblacion(poblacion, T, tam_p, num_atrib)

    Para i = 0 hasta tam_p
        w = {}

        Para j = 0 hasta num_atrib-1
            w.insertar(aleatorio ∈ [0,1])

        poblacion.insertar(cromosoma(w, F(T,T,w)))

Fin
```

3. Enfriamiento simulado

A continuación se muestra el pseudocódigo del algoritmo SimulatedAnnealing. La temperatura inicial considerada es $T_0 = \frac{0{,}3 \cdot valor}{-\log 0{,}3}$, la temperatura final $T_f = 0{,}001$, y posteriormente comprobando que esta sea inferior a a inicial. El valor de $\beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$. La temperatura se actualiza mediante la siguiente fórmula $T = \frac{T}{1 + \beta \cdot T}$.

```
1  Funcion SimulatedAnnealing(T)
2      evaluacioness = 0
3      max_eval = 15000
4      max_vecinos = 10 * num_atrib
5      max_exitos = 0.1 * max_vecino
6      M = 15000/max_vecinos
7      w = {}
8
9      Para j = 0 hasta num_atrib-1
10         w.insertar(aleatorio ∈ [0,1])
11
12     T0 = (0.3*valor)/(-log(0.3))
13     Tf = 0.001
14     Tk = T0
15
16     Mientras Tk > T0
17         Tk = Tk*0.001
18
19     beta = (T0 - Tf) / (M*T0*Tf)
20     exitos = 1
21
22     Mientras Tk > Tf y exitos > 0 y evaluaciones < max_eval
23         exitos = 0
24         vecinos = 0
25
26         Mientras exitos < max_exitos y vecinos < max_vecinos
27             i = entero aleatorio ∈ {0,1,···,numatrib-1}
28             copia = w[i]
29             nuevoVecino(w,i)
30             vecinos = vecinos + 1
31             evaluaciones = evaluaciones + 1
32
33             valor_nuevo = F(T,T,w)
34             dif = valor_nuevo-valor_actual
35
36             Si dif > 0  or (aleatorio ∈ [0,1] <= exp(dif/Tk))
37                 valor_actual = valor_nuevo
38                 exitos = exitos + 1
39                 Si valor_actual > valor_mejor
40                     mejor_solucion = w
41                     valor_mejor = valor_actual
42             Si No
43                 w[i] = copia
44
45         Tk = Tk/(1+beta*Tk)
46
47     Devolver mejor_solucion
48 Fin
```

4. Búsqueda local reiterada

```
Funcion BL(T, w)
    iteraciones = 0
    nn = 0
    nn_top = 20*num_atrib

    indices = {}

    Para i = 0 hasta num_atrib-1
        indices.insertar(i)

    valor_mejor = F(T,T,w)

    Mientras iteraciones < 1000 y nn < nn_top
        Si iteraciones % indices.size == 0
            shuffle(indices)

        k = indices[iteraciones % indices.size]
        copia = w_k

        nuevoVecino(w, k)

        nuevo_valor = F(T, w)
        iteraciones = iteraciones + 1
        nn = nn + 1
        evaluaciones = evaluaciones + 1

        Si nuevo_valor > valor
            nn = 0
            valor = nuevo_valor
        Si No
            w[k] = copia

    Devolver w

Fin
```

```
Funcion ILS(T)
    num_mutaciones = 0.1*num_atrib
    w = {}
    indices = {}

    Para j = 0 hasta num_atrib-1
        w.insertar(aleatorio ∈ [0,1])
        indice.insertar(j)

    w = BL(T,w)
    valor_mejor = F(T,T,w)

    Para i = 0 hasta 14
        w_ = w

        Permutar(indices)

        Para j = 0 hasta num_mutaciones
            nuevoVecino(w, ind[j])

        w_ = BL(T,w_)

        valor_nuevo = F(T,T,w_)

        Si valor_nuevo > valor_mejor
            valor_mejor = valor_nuevo
            w = w_

    Devolver w
Fin
```

5. Evolución diferencial

Se han implementado dos algoritmos de búsqueda local. El que se muestra a continuación obtiene el vector con mutación de la siguiente forma: $V_{i,G} = X_{r1,G} + F \cdot (X_{r2,G} - X_{r3,G})$.

```
1  Funcion EvolucionDiferencialRand(T)
2      generarPoblacion()
3
4      indices = {}
5
6      Para i = 0 hasta tam_poblacion
7          indices.insertar(i)
8
9      Mientras evaluaciones < 15000
10         hijos = {}
11         Para i = 0 hasta tam_poblacion
12             shuffle(indices)
13             padre1 = poblacion[indices[0]].w
14             padre2 = poblacion[indices[1]].w
15             padre3 = poblacion[indices[2]].w
16
17             hijo = {}
18
19             Para j = 0 hasta j < num_atrib-1
20                 Si (aleatorio ∈ [0,1]) < 0.5
21                     v = padre1[j] + 0.5*(padre2[j]-padre3[j])
22                     Si v > 1
23                         v = 1
24                     Si v < 0
25                         v = 0
26                     hijo.insertar(v)
27                 Si No
28                     hijo.insertar(poblacion[i].w[j])
29
30             hijos.insertar(cromosoma(hijo, F(T,T,hijo)))
31             evaluaciones = evaluaciones + 1
32
33         Para i = 0 hasta tam_poblacion
34             Si hijos[i].valor > poblacion[i].valor
35                 poblacion[i] = hijos[i]
36
37     mejor = 0
38
39     Para i = 1 hasta tam_poblacion
40         Si poblacion[i].valor > poblacion[mejor].valor
41             mejor = i
42
43     Devolver poblacion[mejor].w
44
45 Fin
```

El siguiente algoritmo de evolución diferencial obtiene el vector con mutación mediante la siguiente fórmula: $V_{i,G} = X_{i,G} + F \cdot (X_{best,G} - X_{i,G}) + F \cdot (X_{r1,G} - X_{r2,G})$

```
1  Funcion EvolucionDiferencialBest(T)
2      generarPoblacion(poblacion)
3
4      mejor = 0
5
6      Para i = 1 hasta tam_poblacion
7          Si poblacion[i].valor > poblacion[mejor].valor
8              mejor = i
9
10     indices = {}
```

```
11      Para i = 0 hasta tam_poblacion
12          indices.insertar(i)
13
14      Mientras evaluaciones < 15000
15          hijos = {}
16          Para i = 0 hasta tam_poblacion
17              shuffle(indices)
18              padre1 = poblacion[indices[0]].w
19              padre2 = poblacion[indices[1]].w
20
21              hijo = {}
22
23              Para j = 0 hasta j < num_atrib-1
24                  Si (aleatorio ∈ [0,1]) < 0.5
25                      v = poblacion[i].w[j] + 0.5*(poblacion[mejor].w[j]-poblacion[i].w[j])
26                          + 0.5*(padre1[j]-padre2[j])
27                      Si v > 1
28                          v = 1
29                      Si v < 0
30                          v = 0
31                      hijo.insertar(v)
32                  Si No
33                      hijo.insertar(poblacion[i].w[j])
34
35              hijos.insertar(cromosoma(hijo, F(T,T,hijo)))
36              evaluaciones = evaluaciones + 1
37
38              Para i = 0 hasta tam_poblacion
39                  Si hijos[i].valor > poblacion[mejor].valor
40                      mejor = i
41                  Si hijos[i].valor > poblacion[i].valor
42                      poblacion[i] = hijos[i]
43
44      Devolver poblacion[mejor].w
45  Fin
```

Como puede verse en los dos casos, la recombinación se hace de la misma forma, se hace uso del vector de mutación, si el número aleatorio $\in [0, 1]$ obtenido, está por debajo de 0.5.

El remplazamiento se lleva a cabo si el nuevo hijo i-ésimo tiene mejor valor en la función objetivo, que el elemento i-ésimo de la población.

6. Procedimiento considerado para el desarrollo de la práctica y manual de usuario.

La práctica ha sido realizada en C++, el código en su mayoría ha sido desarrollado por mi, incluyendo la lectura de datos. Para la generación de números pseudoaleatorios he utilizado la implemetación aportada en la web de la asignatura, a la cual le he añadido una función para obtener números aleatorios que sigan una distribución normal y un método para permutar de forma aleatoria un vector. Para medir los tiempos de ejecución de los algoritmos he utilizado la biblioteca ctime.

Para no obtener resultados diferentes cada vez que se ejecute el programa, he fijado la semilla para los número pseudoaleatorios con el valor 17.

Para poder ejecutar el programa se ha incluido un makefile en la carpeta FUENTES, por lo que para generar el ejecutable tan solo se deberá de escribir `make` en la terminal. La compilación se ha realizado utilizando clang++ por lo que debería poder compilarse en un Mac. Yo en mi caso he realizado la práctica en Ubuntu.

Podemos ejecutar la práctica escribiendo `./practica3`, tras lo cual se mostrará el mensaje siguiente por pantalla:

```
Pulse el número que desee ejecutar:
1: ozone-320.arff
2: parkinsons.arff
3: spectf-heart.arff
```

Después de elegir el fichero de datos, que vamos a utilizar para la ejecución, tendremos que elegir el algoritmo que deseamos ejecutar:

```
Elija el algoritmo que desee ejecutar:
1: Simulated Annealing
2: Búsqueda Local Reiterada
3: DiferentialEvoulutionRand
4: DiferentialEvoulutionBest
5: Simulated Annealing (T = T*0.95)
```

Tras pulsar alguno de los números se ejecutarán los algoritmos indicados utilizando los datos del fichero elegido, y se mostrarán los resultados obtenidos.

7. Experimentos y análisis de resultados.

Bases de datos utilizadas:

- **Ozone:** base de datos para la detección del nivel de ozono, consta de 320 ejemplos, cada uno con 73 atributos y consta de 2 clases.
- **Parkinsons:** base de datos utilizada para distinguir entre la presencia y la ausencia de la enfermedad. Consta de 195 ejemplos, con 23 atributos y 2 clases.
- **Spectf-heart:** base de datos utilizada para determinar si la fisiología del corazón analizado es correcta o no. Consta de 267 ejemplos con 45 atributos y 2 clases.

Comentar que los ficheros de datos proporcionados contenían más ejemplos de los comentados, el motivo era que había varias líneas repetidas, con lo cual muchos ejemplos aparecían varias veces. Para evitar esto he filtrado los datos para eliminar repetidos y con ello ya se cumplen las cifras comentadas. Además los datos han sido también normalizados utilizando la fórmula

$$x_j^N = (x_j - Min_j)/(Max_j - Min_j)$$

Las prácticas han sido implementadas en C++, y ejecutadas en un ordenador con procesador Intel Core i3, 12 GB de RAM y disco duro SSD en el sistema operativo Ubuntu 16.04 LTS.

Resultados obtenidos

Primero mostramos los resultados que obtuvimos en la primera práctica para 1-NN, relief y BL.

1-NN	**Ozone**				**Parkinsons**				**Spectf-heart**			
	%clas	**%red**	**Agr.**	**T**	**%clas**	**%red**	**Agr.**	**T**	**%clas**	**%red**	**Agr.**	**T**
P1	79.68	0	39.84	0.0021	97.43	0	48.71	0.0006	75.92	0	37.96	0.0015
P2	82.81	0	41.40	0.0021	94.87	0	47.43	0.0006	64.81	0	32.40	0.0015
P3	81.25	0	40.62	0.0021	94.87	0	47.43	0.0006	67.92	0	33.96	0.0019
P4	77.77	0	38.88	0.0022	97.43	0	48.71	0.0005	71.69	0	35.84	0.0015
P5	80.95	0	40.47	0.0024	97.43	0	48.71	0.0002	73.58	0	36.79	0.0015
Media	80.49	0	40.24	0.0022	96.41	0	48.20	0.0005	70.78	0	35.39	0.0016

Relief	**Ozone**				**Parkinsons**				**Spectf-heart**			
	%clas	**%red**	**Agr.**	**T**	**%clas**	**%red**	**Agr.**	**T**	**%clas**	**%red**	**Agr.**	**T**
P1	82.81	13.88	48.35	0.0225	94.87	4.545	49.70	0.0034	83.33	38.63	60.98	0.0108
P2	78.12	18.05	48.09	0.0202	94.87	4.545	49.70	0.0031	70.37	38.63	54.50	0.0107
P3	79.68	19.44	49.56	0.0198	97.43	4.545	50.99	0.0036	69.81	36.36	53.08	0.0100
P4	80.95	13.88	47.42	0.0200	97.43	4.545	50.99	0.0032	75.47	43.18	59.32	0.0096
P5	79.36	26.38	52.87	0.0206	97.43	0	48.71	0.0040	67.92	40.90	54.41	0.0116
Media	80.18	18.33	49.26	0.0206	96.41	3.636	50.02	0.0034	73.38	39.54	56.46	0.0106

BL	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	79.68	79.16	79.42	16.652	89.74	81.81	85.78	1.1339	72.22	63.63	67.92	5.5271
P2	76.56	81.94	79.25	26.105	89.74	81.81	85.78	0.8149	72.22	68.18	70.20	10.118
P3	75	70.83	72.91	21.232	100	72.72	86.36	0.6046	71.69	75	73.34	8.0806
P4	71.42	77.77	74.60	16.267	92.30	90.90	91.60	0.9606	71.69	81.81	76.75	10.382
P5	77.77	84.72	81.25	22.167	94.87	72.72	83.79	0.5796	81.13	79.54	80.33	7.2339
Media	76.09	78.88	77.49	20.484	93.33	80	86.66	0.8187	73.79	73.63	73.71	8.2686

ES	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	71.87	83.33	77.60	12.603	87.17	90.90	89.04	1.4989	70.37	79.54	74.95	6.4520
P2	71.87	81.94	76.90	12.156	89.74	90.90	90.32	1.5276	74.07	81.81	77.94	5.6872
P3	70.31	80.55	75.43	12.629	94.87	81.81	88.34	1.8009	81.13	75	78.06	6.7383
P4	80.95	76.38	78.67	13.493	82.05	86.36	84.20	1.6434	73.58	84.09	78.83	5.7827
P5	74.60	83.33	78.96	12.517	87.17	86.36	86.77	1.6267	73.58	84.09	78.83	6.4888
Media	73.92	81.11	77.51	12.679	88.20	87.27	87.73	1.6195	74.54	80.90	77.72	6.2298

ILS	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	67.18	75	71.09	78.340	89.74	81.81	85.78	5.0235	74.07	75	74.53	31.550
P2	81.25	73.61	77.43	76.995	87.17	81.81	84.49	4.8623	68.51	68.18	68.35	31.744
P3	84.37	75	79.68	77.498	94.87	77.27	86.07	4.8317	73.58	75	74.29	32.787
P4	73.01	87.5	80.25	76.796	94.87	90.90	92.89	4.8228	73.58	79.54	76.56	32.123
P5	74.60	86.11	80.35	77.661	89.74	86.36	88.05	5.5258	71.69	81.81	76.75	35.071
Media	76.08	79.44	77.76	77.458	91.28	83.63	87.45	5.0132	72.29	75.90	74.10	32.655

ED Rand	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	73.43	88.88	81.16	80.976	97.43	90.90	94.17	10.438	70.37	93.18	81.77	36.472
P2	73.43	90.27	81.85	82.231	89.74	90.90	90.32	10.256	70.37	90.90	80.63	35.332
P3	81.25	91.66	86.45	81.647	97.43	90.90	94.17	10.155	75.47	90.90	83.19	35.425
P4	66.66	94.44	80.55	94.119	92.30	90.90	91.60	10.049	75.47	90.90	83.19	34.796
P5	73.01	91.66	82.34	95.719	92.30	90.90	91.60	9.9398	67.92	93.18	80.55	39.918
Media	73.56	91.38	82.47	86.938	93.84	90.90	92.37	10.168	71.92	91.81	81.87	36.388

ED Best	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	78.12	66.66	72.39	94.835	87.17	86.36	86.77	10.309	81.48	79.54	80.51	36.489
P2	78.12	68.05	73.09	94.130	84.61	86.36	85.48	10.261	83.33	77.27	80.30	36.428
P3	79.68	77.77	78.73	86.762	92.30	81.81	87.06	10.638	71.69	72.72	72.21	37.764
P4	76.19	70.83	73.51	86.653	84.61	86.36	85.48	10.540	67.92	79.54	73.73	36.997
P5	85.71	66.66	76.19	89.36	92.30	90.90	91.60	10.093	66.03	63.63	64.83	39.507
Media	79.56	70	74.78	90.348	88.20	86.36	87.28	10.368	74.09	74.54	74.32	37.437

	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
1-NN	80.49	0	40.24	0.0022	96.41	0	48.20	0.0005	70.78	0	35.39	0.0016
Relief	80.18	18.33	49.26	0.0206	96.41	3.636	50.02	0.0034	73.38	39.54	56.46	0.0106
BL	76.09	78.88	77.49	20.484	93.33	80	86.66	0.8187	73.79	73.63	73.71	8.2686
ES	73.92	81.11	77.51	12.679	88.20	87.27	87.73	1.6195	74.54	80.90	77.72	6.2298
ILS	76.08	79.44	77.76	77.458	91.28	83.63	87.45	5.0132	72.29	75.90	74.10	32.655
EDRand	73.56	91.38	82.47	86.938	93.84	90.90	92.37	10.168	71.92	91.81	81.87	36.388
EDBest	79.56	70	74.78	90.348	88.20	86.36	87.28	10.368	74.09	74.54	74.32	37.437

Análisis de la tasa de clasificación

En primer lugar vamos a analizar la tasa de clasificación de los algoritmos a comparar. Para ello se muestra el siguiente gráfico, con el objetivo de facilitar la visualización de los resultados.

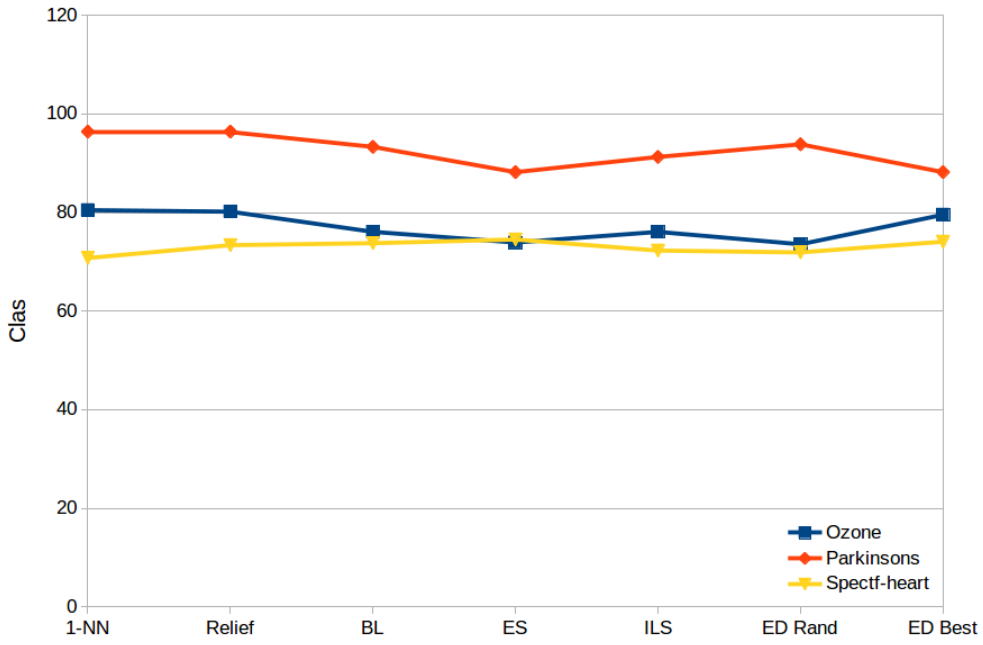

Figura 1: tasaClas

Respecto a la tasa de clasificación no se aprecian grandes diferencias y no se ve que ningún algoritmo destaque por encima de los demás de forma significativa. Donde se van a apreciar diferencias mayores va a ser en la tasa de reducción, la cual será analizada en la siguiente sección.

Si es cierto que puede verse unos resultados algo peores en enfriamiento simulado, en el caso de parkinsons y ozone, aunque como veremos después se compensa con una mejora en la tasa de reducción.

Análisis de la tasa de reducción

Ahora pasamos a analizar la tasa de reducción.

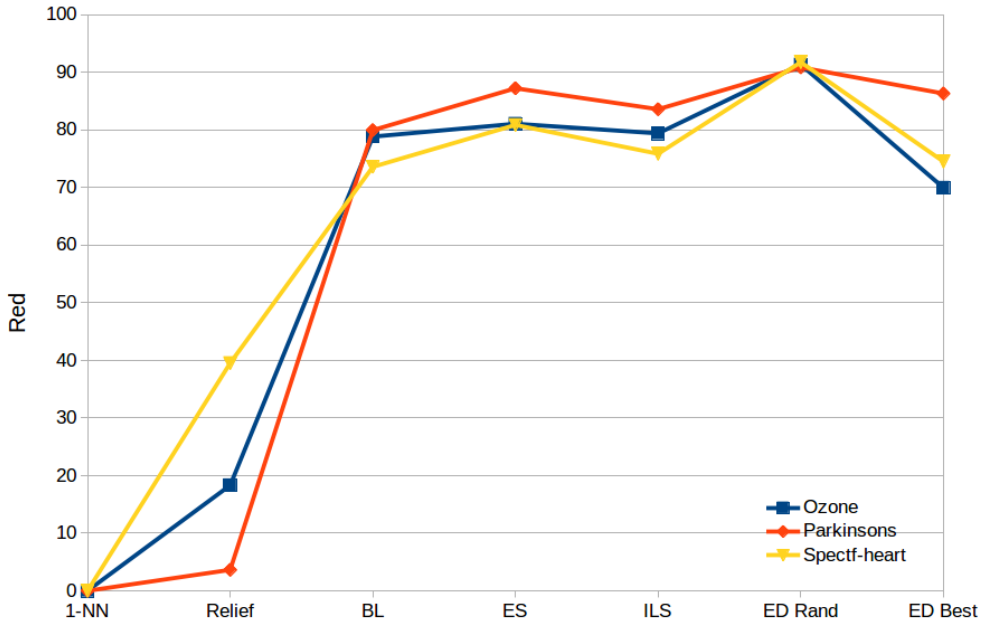

Figura 2: tasaRed

Poco que decir de 1-NN y relief, ya que no tiene mucho sentido compararlo con respecto a los demás ya que estos no tenían como objetivo mejorar la tasa de reducción y de ahí viene su baja puntuación en este aspecto.

Se puede apreciar que el enfriamiento simulado llega a mejorar algo la tasa de reducción obtenida por BL. La capacidad de salir de óptimos locales, permitiendo empeorar la solución con más probabilidad al inicio y disminuyendola durante la ejecución, puede ser la responsable de esta mejora en tasaRed, aunque se ha visto un poco perjudicado en la tasa de clasificación.

La búsqueda local reiterada llega a dar mejores resultados que la simple búsqueda local, cosa que puede deberse a que tiene lo bueno de la búsqueda local que es la intensificación y además le añade algo de exploración al hacer búsquedas locales desde diferentes puntos de arranque mediante mutaciones bruscas de las soluciones que van obteniendose en cada búsqueda local. Como contra veremos que hay un aumento en el tiempo de ejecución del ILS en comparación con BL.

Si comparamos las dos versiones de evolución diferencial vemos que la versión de Rand consigue mejores resultados. Esto puede deberse a que ED best va moviendose en torno a la mejor solución encontrada, de forma que aumenta la intensificación, pero puede verse reducida la exploración, con la consecuencia de obtener resultados peores que ED Rand e incluso por debajo de BL.

Análisis del agregado

A continuación vamos a analizar el valor de agregado obtenido por los algoritmos.

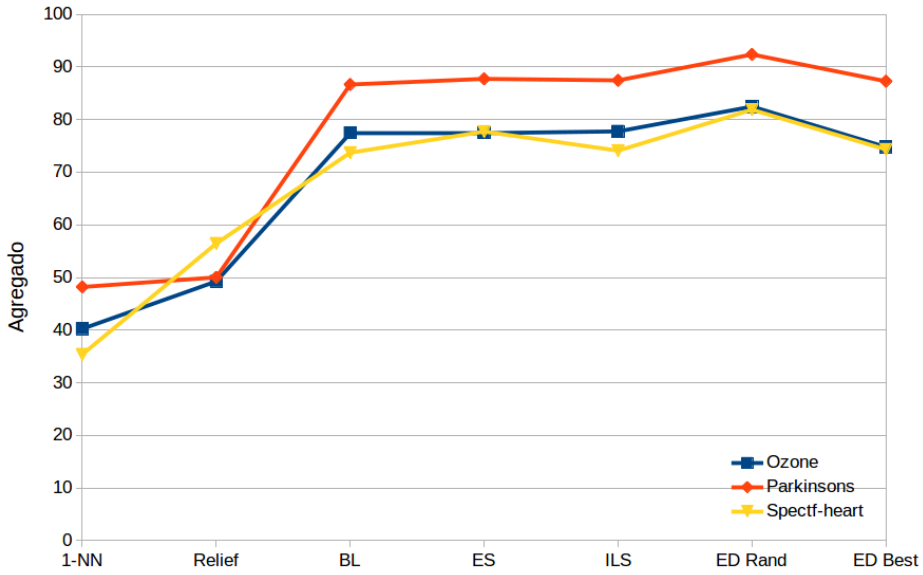

Figura 3: Agregado

El algoritmo que mejores resultados ha dado ha sido evolución diferencial Rand. ES, ILS y ED Best han obtenido resultados muy similares a la búsqueda local. El motivo de que ES no haya obtenido resultados notablemente mejores que el BL, creo que es debido a un enfriamiento demasiado rápido. Que ILS no haya superado al BL puede deberse a búsquedas locales demasiado pequeñas de solo 1000 iteraciones.

ED Rand ha sido el claro ganador. Creo que es debido a que este algoritmo tiene una gran capacidad de exploración y además de explotación debido a la forma de reemplazamiento que tiene.

Análisis del tiempo de ejecución

Viendo las gráficas anteriores podemos ver, ILS y ED Best, obtienen resultamos muy parecidos a los del BL en agregado, pero con unos tiempos superiores de ejecución en el caso de ILS y ED Best, por lo que puede ser preferible la BL. En cambio, aunque ES ha obtenido valores similares en el agregado, se puede apreciar como a medida que aumenta la dimensión del problema, la diferencia entre los tiempos de ejecución de BL y ES se va haciendo más notable, siendo menores los de ES. Por lo que puede ser preferible ES, debido a la calidad de sus soluciones para los tiempos tan tan buenos que ofrece. Si el tiempo no es tan importante se podría hacer uso de ED Rand que mejora algo los resultados obtenidos por BL, a pesar de que aumenta bastante el tiempo de ejecución.

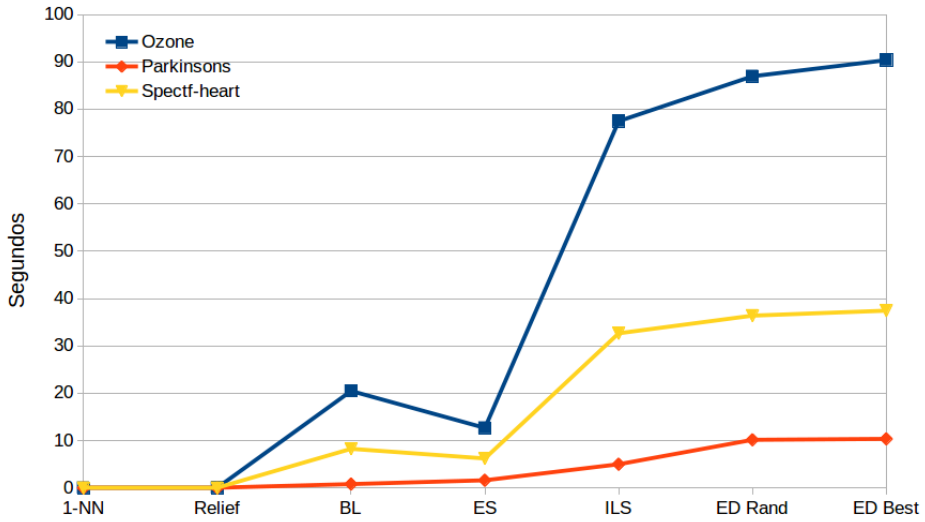

Figura 4: Tiempos

Enfriamiento simulado T = T*0.95

Por último se ha vuelto a ejecutar el algoritmo de enfriamiento simulado, pero cambiando la fórmula de actualización de la temperatura por $T = T * 0{,}95$. Esto se a realizado con el objetivo de obtener un enfriamiento más lento. Los resultados han sido los siguientes.

ES 2	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	75	84.72	79.86	83.441	87.17	90.90	89.04	4.4163	66.66	93.18	79.92	26.562
P2	62.5	88.88	75.69	84.080	89.74	90.90	90.32	4.3850	85.18	79.54	82.36	31.668
P3	78.12	91.66	84.89	81.433	71.79	90.90	81.35	4.3575	69.81	86.36	78.08	29.58
P4	74.60	83.33	78.96	86.256	92.30	90.90	91.60	4.0460	77.35	90.90	84.13	27.744
P5	80.95	87.5	84.22	84.000	89.74	90.90	90.32	4.1532	77.35	90.90	84.13	28.371
Media	74.23	87.22	80.72	83.842	86.15	90.90	88.53	4.2716	75.27	88.18	81.72	28.785

A continuación se muestra la tabla de comparación entre el enfriamiento simulado

	Ozone				Parkinsons				Spectf-heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
ES	73.92	81.11	77.51	12.679	88.20	87.27	87.73	1.6195	74.54	80.90	77.72	6.2298
ES 2	74.23	87.22	80.72	83.842	86.15	90.90	88.53	4.2716	75.27	88.18	81.72	28.785

Se aprecian notables mejoras en la tasa de reducción, cosa que se ha visto reflejada en el valor del agregado que ha mejorado con respecto al primer enfriamiento simulado. Las mejoras son debidas al enfriamiento más lento del segundo ES, en el cual la probabilidad de aceptar peores soluciones va disminuyendo más lentamente y con lo cual puede tener mayor capacidad para salir de óptimos locales, al hacer más lenta la convergencia. Lo

que se ha visto repercutido ha sido el tiempo de ejecución, el cual ha aumentado como consecuencia del enfriamiento más lento, que habrá provocado que el algoritmo termine después de un número de iteraciones superior al del primer ES.

8. Referencias

Principalmente los seminarios y los temas de teoría de la asignatura. Además del material para las bases de datos e implementación en c++ de número aleatorios disponibles en la web de la asignatura.