Asignatura (2018-2019)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

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

Simón López Vico, 26504148Y
simondelosbros@correo.ugr.es
Grupo de prácticas: Miércoles

8 de junio de 2019

Índice

1. Breve descripción del problema.

En esta práctica nos encargaremos de resolver el problema del Aprendizaje de Pesos en Características, que consistirá en ajustar un vector de pesos $\{\omega_1, \ldots, \omega_n\}$ donde cada una de sus componentes determinará el valor de la característica i-ésima del conjunto de datos. Si $\omega_i = 1$ significará que el atributo i-ésimo es muy importante, mientras que si $\omega_i = 0$ la característica i-ésima no será relevante; concretamente, cuando $\omega_i < 0{,}2$ descartaremos dicho atributo.

Utilizaremos distintos algoritmos para ajustar nuestro vector de pesos. Una vez optimizado el vector, usaremos el clasificador K-NN (con K=1) para clasificar según el vecino más cercano usando una distancia ponderada según los pesos:

```
function Distancia(vector a, vector b, vector pesos)
    Si tamanio(a) != tamanio(b)
        Devuelve -1

    Para i=0 hasta tamanio(a)
        Si pesos[i]>=0.2
            Distancia += ( pesos[i]*(a[i]-b[i]) )^2

    Devuelve Distancia
```

Véase que no estamos calculando la raíz para computar la distancia euclídea; prescindimos de esto porque los resultados serán los mismos calculando y sin calcular la raíz, y además la operación `sqrt(x)` tiene un alto tiempo de computo.

El código del clasificador K-NN será el siguiente:

```
function one_NN(e_prima,pesos, indice) //indice leave-one-out
    c_min=train[0].etiqueta
    dist_min=Distancia(train[0], e_prima, pesos)

    Para i=1 hasta tamanio(train)
        Si i != indice
            d=Distancia(train[i], e_prima, pesos)
            Si d<d_min
                c_min=train[i].etiqueta
                d_min=d

    Devuelve c_min
```

La variable `indice` indicada en la cabecera de la función es utilizada para la técnica *leave-one-out*, la cual es necesaria usar porque si intentásemos clasificar un ejemplo del conjunto de entrenamiento directamente con el clasificador 1-NN, el ejemplo más cercano sería siempre él mismo, con lo que se obtendría un 100 % de acierto. Para solucionar esto, clasificaremos cada elemento del conjunto de entrenamiento buscando el ejemplo más cercano sin considerar a él mismo.

Para ajustar el vector de pesos W sobre el clasificador trataremos de maximizar la función $F(W) = \alpha * tasa_{clas}(W) + (1-\alpha) * tasa_{red}(W)$, donde $tasa_{clas}$ determinará el porcentaje de instancias bien clasificadas sobre el total de instancias, y $tasa_{red}$ el porcentaje de atributos con un peso asociado menor a 0.2 sobre el total de atributos.

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

2.1. Esquema de representación del problema y formateo de los ficheros de datos.

El esquema de representación asociado a nuestro problemas es el vector de pesos w, el cuál tomará valores en el intervalo $[0, 1]$ y tendrá una longitud de N elementos.

En el caso de Evolución Diferencial usaremos una población de 50 cromosomas, cada uno de ellos representado como un `pair<vector<double>, double>`, donde el primer elemento será el vector de pesos y el segundo el valor de la función de evaluación sobre dicho vector, aplicando la técnica *leave-one-out*.

La resolución de este problema APC ha sido implementada en C++. Para el almacenamiento de los datos recogidos en los distintos ficheros `arff` se ha usado un vector llamado `data` en el que cada componente será un `pair <vector<double>, int >`; el primer elemento del `pair` recogerá los atributos de una instancia del conjunto total de datos, mientras que el segundo será un `int` con la etiqueta de la instancia. El vector `data` será separado en 5 particiones disjuntas más adelante para realizar el *5-fold validation*.

Por otra parte, los ficheros `arff` han sido ligeramente modificados para facilitar la lectura de éstos:

- **colposcopy.arff**: ninguna modificación.
- **ionosphere.arff**: cambio de las clases b y g por 0 y 1.
- **texture.arff**: inicialmente cada uno de los atributos de cada instancia se encuentra separado por una coma y un espacio; se modifica para solamente se separen mediante una coma.
  Clases iniciales: {2, 3, 4, 9, 10, 7, 6, 8, 12, 13, 14}.
  Clases modificadas: {0, 1, 2, 6, 7, 4, 3, 5, 8, 9, 10}.

2.2. Lectura y preprocesado.

Comenzaremos leyendo el fichero de datos pasado al programa con la función `read_data`, la cual leerá un archivo en formato `arff` y guardará todos sus valores en el vector `data`;

además, almacenará el total de atributos y de clases del dataset leído.

A continuación normalizaremos los datos para que se encuentren en el intervalo $[0, 1]$. Para ello usaremos la siguiente fórmula:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j},$$

donde Max_j y Min_j serán el máximo y el mínimo de cada uno de los atributos. Hay que tener en cuenta que si un atributo toma el mismo valor para todas las instancias, la diferencia entre el máximo y el mínimo será 0 por lo que no podremos realizar la división; en este caso, $x_j^N = 0$.

Una vez normalizados todos los datos, realizaremos las particiones necesarias para el *K-fold validation*, en nuestro caso K=5. Para ello usaremos el siguiente código:

```
function k folds()
    Mezclar(data)

    Para i=0 hasta n_clases
        Para j=0 hasta tamanio(data)
            Si data.clase == i
                particiones[n_introd%n_part].push_back(data[j]);
```

De esta manera, el número de clases estará balanceada respecto a cada partición, es decir, en cada partición habrá el mismo porcentaje (aproximadamente) de elementos con una clase determinada.

Notemos que la normalización se debe de aplicar siempre antes de realizar las particiones de elementos, pues si se hiciera al contrario cada partición estaría normalizada por un máximo y mínimo distinto.

Finalmente generaremos los conjuntos *train* y *test* con los que vamos a entrenar y validar nuestro clasificador.

2.3. Generación de soluciones aleatorias.

En el caso de Enfriamiento Simulado y Búsqueda Local Reiterada, hemos de generar un vector de pesos inicial con valores aleatorios dentro de una distribución $U(0, 1)$. Para ello, hemos definido la siguiente función que nos devolverá dicha solución inicial:

```
function generar_solucion_inicial()
   distribucion_uniforme distribution(0,1)

   Para i=0 hasta n_atrib
      pesos.aniade(distribution(generator))
```

```
   Devuelve pesos
```

Por otra parte, generaremos la población inicial aleatoria para los dos algoritmos de Evolución Diferencial con la función `generate_cromosomas(tam)`, a la cuál le pasaremos el tamaño de la población e inicializará el vector de cromosomas utilizando la función `generar_solucion_inicial()` definida anteriormente.

```
function generate_cromosomas(tam)
   cromosomas.clear()

   Para i=0 hasta tam
      cromosoma tmp
      tmp.first=generar_solucion_inicial()
      tmp.second=f_evaluacion(tmp)
      cromosomas.aniade(tmp)
```

2.4. Algoritmo de Búsqueda Local.

El algoritmo de búsqueda local utilizado será el mismo del que hablamos en la práctica 1, aunque tendrá ligeras modificaciones para controlar el número máximo de vecinos evaluados, el número de llamadas a la función de evaluación y el vector de pesos sobre el que se ejecutará la búsqueda. El código es el siguiente:

```
function busqueda_local(alpha, max_eval, &evals, pesos)
   n_generados=0, n_eval=0
   max_generados=20*n_atrib

   Si pesos.esta_vacio()   pesos=generar_solucion_inicial()

   indices=generar_indices(n_atrib)
   int i=0;
   valor_eval=0;

   Mientras n_generados < max_generados && n_eval < max_eval
      Si i==0     baraja(indices)

      tmp=pesos
      mutacion(tmp, indices[i])
      n_generados++

      new_valor=f_evaluacion(tmp)
      n_eval++

      Si new_valor>valor_eval
         pesos=tmp
         valor_eval=new_valor
```

```
        n_generados=0

    i++
    Si i==n_atrib  i=0

evals+=n_eval  //Actualizamos el numero de evaluaciones
Devuelve pesos
```

El operador de generación de vecino consistirá en realizar un "movimiento" basado en la mutación de una componente del vector de pesos sumándole un valor generado mediante una distribución $\mathcal{N}(0,0{,}4)$ y truncándolo a 0 si el resultado es menor que 0 o a 1 si el resultado es mayor que 1. El código es el siguiente:

```
function mutacion(&pesos, indice)
   distribucion_normal distribution(0.0,0.4)

   pesos[i]+=distribution(generator)

   Si pesos[indice]>1   pesos[indice]=1
   Si pesos[indice]<0   pesos[indice]=0
```

3. Enfriamiento Simulado (ES).

El algoritmo de enfriamiento simulado consistirá en generar una solución aleatoria inicial e ir mutando en cada iteración una de sus componentes mediante el operador de movimiento mencionado anteriormente. Esta solución mutada reemplazará a la actual si la diferencia entre su función de evaluación y la del vector de pesos actual es mayor que 0, o si, tras generar un valor aleatorio en una distribución $U(0,1)$, su valor es menor a $e^{\Delta f/(T*K)}$, donde Δf es la diferencia entre la solución mutada y la solución actual (si $\Delta f = 0$, pondremos $\Delta f = 0{,}005$), T es la temperatura actual y $K = 1$. Además, si es mejor que la mejor solución encontrada hasta ese momento, la definiremos como la mejor solución.

Esta temperatura T irá disminuyendo cada vez que generemos un máximo de vecinos o que obtengamos un número máximo de éxitos (donde el éxito consiste en reemplazar el vector de pesos actual por el mutado). La temperatura disminuirá mediante la fórmula:

$$T_{k+1} = \frac{T_k}{1 + \beta * T_k} \quad ; \quad \beta = \frac{T_0 - T_f}{M * T_0 * T_f}, \tag{3.1}$$

donde M es el número de enfriamientos a realizar. En nuestro caso, siguiendo este esquema de enfriamiento, la temperatura disminuye muy rápido (en la primera iteración de `colposcopy` pasa de 11.0691 a 0.02395), por lo que usaremos un esquema de enfriamiento más sencillo el cuál estará definido por $T_{k+1} = \alpha * T_k$, donde $\alpha = 0{,}9$.

Esto se repetirá hasta que se llame más de 15000 veces a la función de evaluación, se obtengan 0 éxitos con una temperatura determinada o se llegue a una temperatura final definida. La temperatura inicial la calcularemos como $T_0 = \frac{\mu * C(S_0)}{-ln(\phi)}$, donde $C(S_0)$ es el coste de la solución inicial y $\mu = \phi = 0{,}3$; la temperatura final valdrá $T_f = 0{,}001$

El código será el siguiente:

```
function ES()
   solucion_actual=generar_solucion_inicial()
   valor_actual=f_evaluacion(solucion_actual)
   mejor_solucion=solucion_actual, mejor_valor=valor_actual
   T0=0.3*valor_actual/1.20397280433, TF=0.001, Tactual=T0
   Si TF>T0 Devuelve mejor_solucion
   K=1
   max_vecinos=10*n_atrib, max_exitos=0.1*max_vecinos
   n_eval=0, max_eval=15000
   M=max_eval/max_vecinos  // Numero de enfriamientos

   Hacer
      n_vecinos=0, n_exitos=0

      Mientras n_vecinos<max_vecinos y
               n_exitos<max_exitos y
               n_eval<max_eval

         tmp=solucion_actual
         mutacion(tmp, Randint(0,n_atrib-1))
         n_vecinos++
         new_valor=f_evaluacion(tmp)
         n_eval++
         increment=new_valor-valor_actual

         Si increment==0   increment=0.005

         value=exp(increment/(Tactual*K))

         Si increment>0 o distribution(generator) <= value
            solucion_actual=tmp
            valor_actual=new_valor
            n_exitos++
            Si valor_actual>mejor_valor
               mejor_solucion=solucion_actual
               mejor_valor=valor_actual

      Tactual*=0.9 // Enfriamiento

   Mientras Tactual>TF y n_exitos>0 y n_eval<max_eval

   Devuelve mejor_solucion
```

4. Búsqueda Local Reiterada (ILS).

El algoritmo ILS consistirá en generar una solución inicial aleatoria y aplicar el algoritmo de Búsqueda Local sobre ella. Una vez obtenida la solución optimizada, se estudiará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de BL sobre esta solución mutada. Este proceso se repite un determinado número de veces, devolviéndose la mejor solución encontrada en todo el proceso. Por tanto, se sigue el criterio del mejor como criterio de aceptación de la ILS.

La mutación a realizar sobre el vector al que aplicaremos BL consistirá en mutar $t = 0{,}1 * nAtributos$ características escogidas aleatoriamente mediante el operador de mutación definido en la sección de búsqueda local.

Ejecutaremos el algoritmo con una condición de parada de llamar a la Búsqueda Local 15 veces, donde el número máximo de evaluaciones de la BL será de 1000 evaluaciones. El pseudocódigo es el siguiente:

```
function ILS()
   solucion_actual=generar_solucion_inicial()
   mejor_solucion=busqueda_local(solucion_actual)
   mejor_valor=f_evaluacion(mejor_solucion)
   indices=generar_indices(n_atrib) // Para los t atributos
   t=0.1*n_atrib

   Para i=1 hasta 15
      baraja(indices)
      tmp=mejor_solucion

      Para j=0 hasta t
         mutacion(tmp, indices[j])

      tmp=busqueda_local(tmp)
      new_valor=f_evaluacion(tmp)

      Si new_valor>mejor_valor
         mejor_valor=new_valor
         mejor_solucion=tmp

   Devuelve mejor_solucion
```

5. Evolución Diferencial (DE).

La DE es un modelo evolutivo propuesto para optimización con parámetros reales que enfatiza la mutación y utiliza un operador de cruce/recombinación a posteriori. Básica-

mente, el algoritmo se encargará de crear una nueva generación que la conformarán todos los elementos de la población actual con un porcentaje de sus atributos mutados mediante una fórmula dada, en la que intervendrán n cromosomas de la población escogidos aleatoriamente (tres para *Rand/1* y dos para *Current-to-best/1*). Tras esto, comparará el cromosoma mutado y sin mutar y se quedará en la población el que mejor resultado obtenga en su función de evaluación. En nuestro caso, usaremos una población de tamaño 50 y mutaremos un 50 % (escogidos aleatoriamente) de los atributos de cada cromosoma.

Para no tener que generar $nAtrib$ valores aleatorios por cada hijo que generemos, fijaremos el número de atributos que van a ser modificados, es decir, $0{,}5 * nAtrib$. Para cada hijo, barajaremos un vector de índices (entre 0 y $nAtrib - 1$) y realizaremos la mutación sobre los $0{,}5 * nAtrib$ primeros índices.

5.1. DE/Rand/1.

En este caso, la fórmula para obtener el vector con mutación es la siguiente:

$$V_{i,G} = X_{r1,G} + F * (X_{r2,G} - X_{r3,G}), \tag{5.1}$$

donde $r1$, $r2$ y $r3$ será el índice de los individuos escogidos aleatoriamente de forma excluyente incluyendo el valor de i, G la generación en la que nos encontramos y $F = 0{,}5$. El código es el siguiente:

```
function mutacion_rand(&pesos,p1,p2,p3,i)
   F=0.5
   pesos[i]=cromosomas[p1][i]+
            F*(cromosomas[p2][i]-cromosomas[p3][i])

   Si pesos[i]>1  pesos[i]=1
   Si pesos[i]<0  pesos[i]=0

function DE_rand_1()
   tam_pob=50
   prob_cruce=0.5
   eval=0, max_eval=15000

   generate_cromosomas(tam_pob)
   eval+=tam_pob

   Mientras eval<15000
      Para i=0 hasta tam_pob
         indices=generar_indices(n_atrib)
         barajar(indices)

         p1, p2, p3
         Hacer p1=Rand(0,tam_pob-1) Mientras p1==i
         Hacer p2=Rand(0,tam_pob-1) Mientras p2==i o p2==p1
         Hacer p3=Rand(0,tam_pob-1) Mientras p3==i o p3==p2 o p3==p1
```

```
            hijo_generado=cromosomas[i]

            Para j=0 hasta prob_cruce*n_atrib
               mutacion_rand(hijo_generado,p1,p2,p3,indices[j])

            hijo_generado.second=f_evaluacion(hijo_generado)
            eval++
            hijos.aniade(hijo_generado)

         // Reemplazamiento
         Para i=0 hasta tam_pob
            Si hijos[i].second>cromosomas[i].second
               cromosomas[i]=hijos[i]

      reordenar()

      Devuelve cromosomas[0]
```

5.2. DE/current-to-best/1.

Por otra parte, el esquema de mutación para esta variante del algoritmo DE es el siguiente:

$$V_{i,G} = X_{i,G} + F * (X_{best,G} - X_{i,G}) + F * (X_{r1,G} - X_{r2,G}), \tag{5.2}$$

donde $X_{best,G}$ denotará el mejor individuo de la generación G y $r1$ y $r2$ serán escogidos aleatoriamente.

El código en este caso es muy similar al de *DE/rand/1*, pero ahora deberemos de calcular el mejor individuo de la población en cada generación, pues es necesario para calcular el valor de la mutación.

```
function mutacion_ctb(&pesos,p1,p2,current,i)
   F=0.5
   pesos[i]=cromosomas[current][i]+
            F*(cromosomas[0][i]-cromosomas[current][i])+
            F*(cromosomas[p1][i]-cromosomas[p2][i])
   Si pesos[i]>1   pesos[i]=1
   Si pesos[i]<0   pesos[i]=0

function DE_current_to_best()
   tam_pob=50
   prob_cruce=0.5
   eval=0, max_eval=15000

   generate_cromosomas(tam_pob)
   eval+=tam_pob
   reordenar()
```

```
Mientras eval<15000
   Para i=0 hasta tam_pob
      indices=generar_indices(n_atrib)
      barajar(indices)

      p1, p2
      Hacer p1=Rand(0,tam_pob-1) Mientras p1==i
      Hacer p2=Rand(0,tam_pob-1) Mientras p2==i o p2==p1

      hijo_generado=cromosomas[i]

      Para j=0 hasta prob_cruce*n_atrib
         mutacion_ctb(hijo_generado,p1,p2,i,indices[j])

      hijo_generado.second=f_evaluacion(hijo_generado)
      eval++
      hijos.aniade(hijo_generado)

   // Reemplazamiento
   Para i=0 hasta tam_pob
      Si hijos[i].second>cromosomas[i].second
         cromosomas[i]=hijos[i]

   reordenar()

Devuelve cromosomas[0]
```

6. Manual de usuario.

Para replicar los resultados de mis algoritmos y el funcionamiento de mi código, habría que empezar por formatear los archivos de datos tal y como se han comentado en la subsección **Esquema de representación del problema y formateo de los ficheros de datos.** Tras ello, bastaría con compilar el código con la instrucción `make` y ejecutar el archivo `ejecutar.sh`, el cual generará un fichero `solucion.txt` con los resultados de la ejecución de los algoritmos sobre los 3 ficheros de datos. Es posible que hubiera que dar permisos de ejecución al archivo `ejecutar.sh`.

El archivo `solucion.txt` será del siguiente estilo:

```
---------------------- Datos usados: datos -------------------------

=====================================================================
=====================================================================
                              PARTICION N
```

```
=====================================================================
=====================================================================

Algoritmo:          Tasa clas: VALOR              Tasa red: VALOR
*************** F evaluacion: VALOR (tiempo: VALOR) ******************
```

7. Experimentos y análisis de resultados.

La semilla empleada para la realización de los experimentos está definida al comienzo del archivo `main.cpp`, tras los `#include`, con el código `#define SEED 10000`.

Las tablas con los resultados de la primera práctica (1-NN, BL y RELIEF) son las siguientes:

Tabla 5.1: Resultados obtenidos por el algoritmo KNN en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	67,7966	0	33,8983	0,003576	85,9155	0	42,95775	0,006898	93,6364	0	46,8182	0,011131
Partición 2	78,9474	0	39,4737	0,003404	82,8571	0	41,42855	0,004034	94,5455	0	47,27275	0,011008
Partición 3	82,4561	0	41,22805	0,003391	85,7143	0	42,85715	0,004527	90,9091	0	45,45455	0,010584
Partición 4	70,1754	0	35,0877	0,003435	90	0	45	0,004011	90,9091	0	45,45455	0,010899
Partición 5	70,1754	0	35,0877	0,003516	87,1429	0	43,57145	0,004022	91,8182	0	45,9091	0,010889
Media	73,91018	0	36,95509	0,0034644	86,32596	0	43,16298	0,0046984	92,36366	0	46,18183	0,0109022

Tabla 5.2: Resultados obtenidos por el algoritmo BL en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	83,0508	77,4194	80,2351	10,7524	94,3662	67,6471	81,00665	5,21633	90,9091	77,5	84,20455	22,9095
Partición 2	87,7193	70,9677	79,3435	9,3236	90	82,3529	86,17645	6,77597	95,4545	87,5	91,47725	46,213
Partición 3	87,7193	53,2258	70,47255	11,6677	94,2857	85,2941	89,7899	8,11535	92,7273	85	88,86365	48,1748
Partición 4	78,9474	59,6774	69,3124	6,64878	98,5714	85,2941	91,93275	7,76152	96,3636	72,5	84,4318	20,6004
Partición 5	80,7018	54,8387	67,77025	5,92223	92,8571	85,2941	89,0756	6,55822	90	67,5	78,75	17,1701
Media	83,62772	63,2258	73,42676	8,862942	94,01608	81,17646	87,59627	6,885478	93,0909	78	85,54545	31,01356

Tabla 5.3: Resultados obtenidos por el algoritmo RELIEF en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	69,4915	27,4194	48,45545	0,012593	84,507	2,94118	43,72409	0,015008	94,5455	2,5	48,52275	0,03811
Partición 2	78,9474	20,9677	49,95755	0,012947	82,8571	2,94118	42,89914	0,015028	93,6364	15	54,3182	0,049454
Partición 3	84,2105	16,129	50,16975	0,012867	85,7143	2,94118	44,32774	0,015125	90	2,5	46,25	0,066729
Partición 4	66,6667	19,3548	43,01075	0,012814	91,4286	2,94118	47,18489	0,015878	95,4545	15	55,22725	0,044007
Partición 5	80,7018	56,4516	68,5767	0,012803	88,5714	2,94118	45,75629	0,014961	93,6364	15	54,3182	0,039821
Media	76,00358	28,0645	52,03404	0,0128048	86,61568	2,94118	44,77843	0,0152	93,45456	10	51,72728	0,0476242

Los resultados obtenido en la ejecución de los algoritmos referentes a esta práctica (ES, ILS, DE/rand/1, DE/current-to-best/1) son los siguientes:

Tabla 5.4: Resultados obtenidos por el algoritmo ES en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	88,1356	85,4839	86,80975	22,6183	94,3662	88,2353	91,30075	15,9475	96,3636	85	90,6818	48,2902
Partición 2	92,9825	80,6452	86,81385	22,3967	97,1429	82,3529	89,7479	15,9473	97,2727	85	91,13635	47,8056
Partición 3	91,2281	79,0323	85,1302	22,3299	97,1429	88,2353	92,6891	15,3149	95,4545	82,5	88,97725	47,5567
Partición 4	85,9649	83,871	84,91795	21,3771	95,7143	91,1765	93,4454	15,5013	94,5455	85	89,77275	47,9453
Partición 5	94,7368	82,2581	88,49745	22,6236	97,1429	88,2353	92,6891	15,922	90,9091	85	87,95455	47,8512
Media	90,60958	82,2581	86,43384	22,26912	96,30184	87,64706	91,97445	15,7266	94,90908	84,5	89,70454	47,8898

Tabla 5.5: Resultados obtenidos por el algoritmo ILS en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	89,8305	82,2581	86,0443	46,2122	92,9577	91,1765	92,0671	50,6143	94,5455	87,5	91,02275	137,076
Partición 2	89,4737	74,1935	81,8336	43,9841	98,5714	88,2353	93,40335	49,3932	95,4545	87,5	91,47725	141,27
Partición 3	91,2281	74,1935	82,7108	43,9857	98,5714	85,2941	91,93275	47,6292	95,4545	82,5	88,97725	148,652
Partición 4	85,9649	74,1935	80,0792	44,4701	95,7143	88,2353	91,9748	49,3599	97,2727	82,5	89,88635	143,131
Partición 5	92,9825	69,3548	81,16865	45,3213	95,7143	94,1176	94,91595	51,0641	95,4545	82,5	88,97725	142,936
Media	89,89594	74,83868	82,36731	44,79468	96,30582	89,41176	92,85879	49,61214	95,63634	84,5	90,06817	142,613

Tabla 5.6: Resultados obtenidos por el algoritmo DE/rand/1 en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	91,5254	90,3226	90,924	47,9166	98,5915	91,1765	94,884	59,1786	98,1818	87,5	92,8409	151,914
Partición 2	92,9825	87,0968	90,03965	47,0919	97,1429	91,1765	94,1597	60,0063	99,0909	87,5	93,29545	153,117
Partición 3	96,4912	88,7097	92,60045	46,2612	97,1429	91,1765	94,1597	59,3227	96,3636	87,5	91,9318	150,46
Partición 4	92,9825	88,7097	90,8461	47,172	97,1429	88,2353	92,6891	59,1227	95,4545	90	92,72725	153,591
Partición 5	94,7368	87,0968	90,9168	46,3821	97,1429	91,1765	94,1597	59,0239	96,3636	87,5	91,9318	169,418
Media	93,74368	88,38712	91,0654	46,96476	97,43262	90,58826	94,01044	59,33084	97,09088	88	92,54544	155,7

Tabla 5.7: Resultados obtenidos por el algoritmo DE/current_to_best/1 en el problema del APC

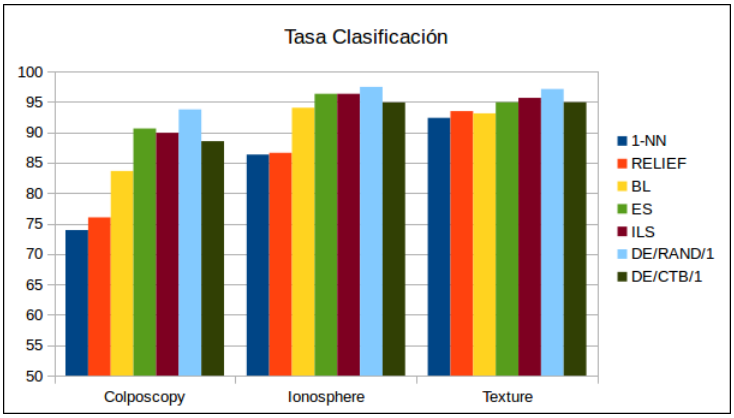
	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
Partición 1	84,7458	70,9677	77,85675	46,6198	97,1831	85,2941	91,2386	59,5124	96,3636	72,5	84,4318	148,848
Partición 2	89,4737	66,129	77,80135	46,1291	92,8571	85,2941	89,0756	58,4973	96,3636	75	85,6818	153,456
Partición 3	94,7368	59,6774	77,2071	46,8826	92,8571	82,3529	87,605	62,5479	92,7273	80	86,36365	159,967
Partición 4	82,4561	64,5161	73,4861	45,9504	95,7143	82,3529	89,0336	58,7911	95,4545	72,5	83,97725	167,328
Partición 5	91,2281	70,9677	81,0979	45,4196	95,7143	91,1765	93,4454	58,1283	93,6364	80	86,8182	159,909
Media	88,5281	66,45158	77,48984	46,2003	94,86518	85,2941	90,07964	59,4954	94,90908	76	85,45454	157,9016

Finalmente, la tabla con todos los resultados será la siguiente:

Tabla 5.8: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	73,91018	0	36,95509	0,0034644	86,32596	0	43,16298	0,0046984	92,36366	0	46,18183	0,0109022
RELIEF	76,00358	28,0645	52,03404	0,0128048	86,61568	2,94118	44,77843	0,0152	93,45456	10	51,72728	0,0476242
BL	83,62772	63,2258	73,42676	8,862942	94,01608	81,17646	87,59627	6,885478	93,0909	78	85,54545	31,01356
ES	90,60958	82,2581	86,43384	22,26912	96,30184	87,64706	91,97445	15,7266	94,90908	84,5	89,70454	47,8898
ILS	89,89594	74,83868	82,36731	44,79468	96,30582	89,41176	92,85879	49,61214	95,63634	84,5	90,06817	142,613
DE/RAND/1	93,74368	88,38712	91,0654	46,96476	97,43262	90,58826	94,01044	59,33084	97,09088	88	92,54544	155,7
DE/CTB/1	88,5281	66,45158	77,48984	46,2003	94,86518	85,2941	90,07964	59,4954	94,90908	76	85,45454	157,9016

Empecemos analizando la tasa de clasificación de todos los algoritmos sobre los distintos dataset, que se mostrará en la siguiente gráfica:

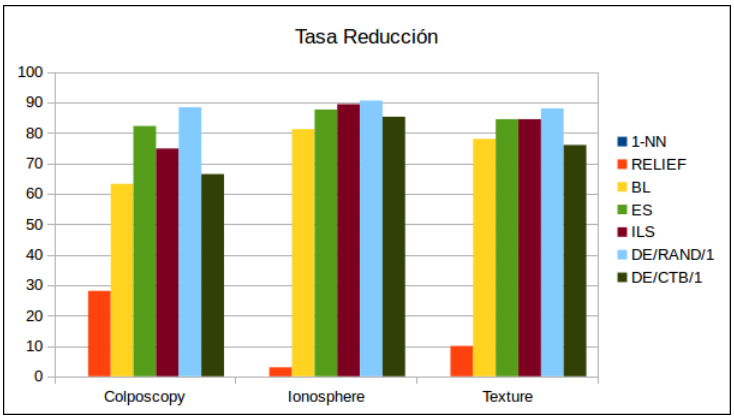


(Notar que si la diferencia entre dos algoritmos parece muy grande es porque los resultados se muestran entre 50 y 100, no entre 0 y 100.)

Podemos ver que la tasa de clasificación tiende a ser similar entre los algoritmos ES, ILS, y los dos tipos de DE, siendo mejor en todos los casos el algoritmo de evolución diferencial con el esquema de mutación *rand/1*. Por otra parte, los algoritmos de la primera práctica obtienen peores resultados que los de ésta, aunque la búsqueda local se acerca a dichos valores para los dataset `ionosphere` y `texture`, e incluso en `texture`, la tasa de clasificación de RELIEF es mayor que la de la BL.

En el caso de `texture`, la diferencia entre el menor y el mayor valor de tasa de clasificación entre todos los algoritmos es de sólo un 5 % (92.36366 % en 1-NN, 97.09088 % en DE/rand/1), por lo que en dicho dataset se obtienen muy buenos valores de tasa de clasificación, y solo con ésto no podríamos discernir entre qué algoritmo elegir para nuestro ajuste de datos. Con la inclusión de la tasa de reducción, estos valores pasarán a estar mas separados entre sí.

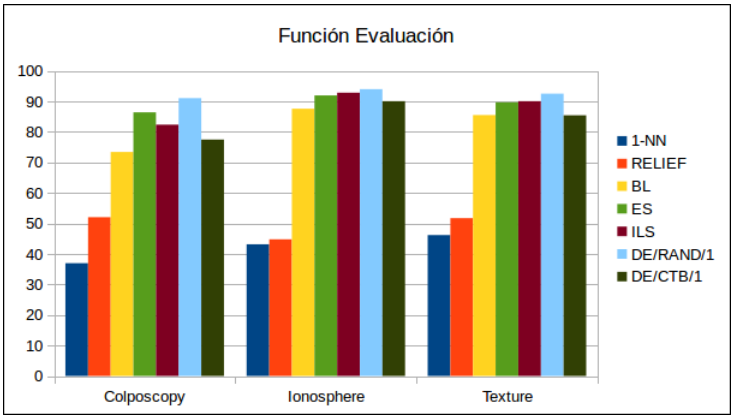
Veamos ahora la tasa de reducción:



La tasa de reducción en 1-NN y RELIEF es muy baja; en 1-NN vale 0, ya que el vector de pesos esta formado solo por unos, y en RELIEF obtiene valores muy pequeños, pues este algoritmo no busca aumentar el valor de la tasa de reducción. Por tanto, no tiene sentido que los comparemos con el resto.

Al igual que en la tasa de clasificación, el algoritmo que obtiene el mayor valor es DE/-rand/1, seguido por *Simulated Annealing* e ILS, el cuál obtiene peores resultados cuando el algoritmo de BL obtiene malos resultados, como es esperable. Respecto a DE/current-to-best/1, podemos ver que el esquema de mutación definido para este algoritmo no optimiza tan bien la tasa de reducción como el de DE/rand/1, pues el valor de la tasa de reducción de DE/current-to-best/1 es muy bajo en comparación con el resto de algoritmos (llegando a ser hasta más bajo que el de la BL). Además, la mutación de DE/current-to-best/1 va siempre moviéndose en entornos de la mejor solución, por lo que está aumentando la intensificación y dejando más de lado la exploración.

Con lo visto hasta ahora, podemos deducir que el algoritmo con el que mejores resultados hemos obtenido es DE/rand/1; veamos y comentemos los datos de la función de evaluación:

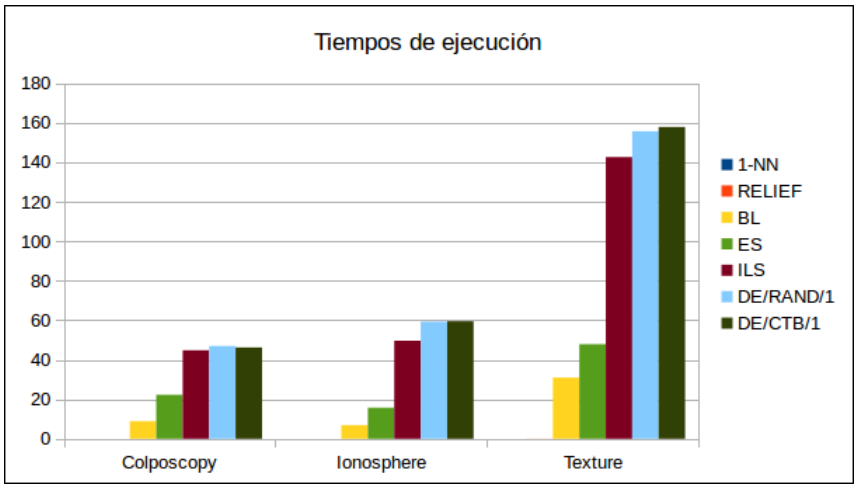


Los valores de RELIEF y 1-NN son tan bajos como esperábamos, pues su tasa de reducción era casi nula. Por parte la BL, no llega a obtener muy buenos resultados, pero resulta realmente útil en ILS, que es el segundo mejor clasificado en 2 de los 3 datasets. Respecto a ES, supera a ILS en `colposcopy` y casi que lo alcanza para los otros dos datasets (tal y como ha pasado en los valores de la tasa de clasificación y de reducción). Esto puede ser debido al enfriamiento de ES, pues en el momento de tener una temperatura muy baja se realiza una exploración casi nula (no como en ILS) y puede que nos quedemos en un máximo local.

El valor del agregado ha disminuido mucho en DE/current-to-best/1 debido a sus malos resultados en la tasa de reducción, y por tanto el claro ganador es DE/rand/1, que realiza una buena exploración por realizar la mutación escogiendo elementos aleatorios de la

población, y una buena intesificación por el método de reemplazamiento que sigue.

Aún así, ya que en dos casos ILS le pisa los talones a DE/rand/1, comparemos los tiempos de cada algoritmo para tomar una decisión final:



Entre los dos algoritmos de evolución diferencial los tiempos se parecen mucho, siendo algo mayores en DE/current-to-best/1 porque en cada generación hemos de buscar el individuo con mayor agregado.

Respecto a la decisión final entre si utilizar ILS o DE/rand/1, los tiempos para `colcoscopy` son casi iguales, aunque recordemos que en dicho dataset ILS no obtenía tan buenos resultados como en el resto. Para los otros dos dataset, la diferencia de tiempo entre estos dos algoritmos es de unos 10-20 segundos, valor que irá incrementándose cuanto mayor sea el conjunto de datos. Por tanto, si tenemos un tiempo limitado, podríamos conformarnos con utilizar ILS, el cuál nos arrojará unos buenos resultados de aproximadamente el 90 % valor de agregado, pero si nuestro tiempo no es tan limitado, será preferible usar DE/rand/1 que nos devolverá el resultado con un porcentaje mayor de acierto.