Metaheurísticas

Práctica 3.b
APC

Johanna Capote Robayna
4 del Doble Grado en Informática y Matemáticas
johannacapote@correo.ugr.es
42199994p
Grupo del Martes

8 de junio de 2019

UNIVERSIDAD
DE GRANADA

Índice

1 Descripción del problema **3**

2 Descripción de la aplicación de los algoritmos **4**

3 Descripción de los algoritmos considerados **9**
3.1 Algoritmos P1 9
3.2 Algoritmos P3 13
3.2.1 Enfriamiento simulado 13
3.2.2 ILS 15
3.2.3 Evolución diferencial (DE) 17

4 Procedimiento considerado para desarrollar la práctica **21**

5 Experimentos y análisis de resultados **22**
5.1 Descripción de los casos del problema 22
5.1.1 Ionosphere 22
5.1.2 Colposcopy 22
5.1.3 Texture 22
5.2 Resultados obtenidos 22
5.3 Análisis de los resultados 26
5.3.1 Casos base 26
5.3.2 Algoritmos considerados en la práctica 26

1. Descripción del problema

El problema que se plantea es un problema de clasificación, el cual consiste en dado un conjunto de datos ya clasificados, obtener un clasificador que tras entrenarlo permita clasificar otros ejemplos. En nuestro problema utilizaremos el clasificador 1-NN, que asigna a cada nuevo elemento la categoría del vecino más cercano. La distancia considerada es la euclídea ponderada por un vector de pesos $w_i \in [0, 1]$:

$$d_x(u, v) = \sqrt{\sum_i w_i(u_i - v_i)^2 + \sum_j w_j d_h(u_j, v_j)} \quad u, v \in \mathbb{R}^n$$

Para medir la presición de nuestro vector de pesos podemos medir dos características:

- La presición (*Tclas*): consideramos el número de elementos que se clasifican correctamente para medir la calidad del clasificador, como de bueno es.
- La simplicidad (*Tred*): medimos esta característica teniendo en cuenta cuantos pesos valen menos de 0.2, ya que estos pesos tendrán menos importancia en el clasificador.

El objetivo de esta práctica es encontrar un vector de pesos que maximice estás dos características, es decir nuestra función objetivo es:

$$F_{obj}(w) = \alpha Tclass(w) + (1 - \alpha)Tred(w)$$

2. Descripción de la aplicación de los algoritmos

En esta sección se describen los elementos comunes a todos los algoritmos desarrollados, así como los esquemas de representación de datos de entrada y soluciones. Todo el código se ha desarrollado en `C++11`.

Esquemas de representación P1

Para representar los datos en el programa se emplea una estructura `Ejemplo` que recoge toda la información necesaria: un `vector<double>` con los valores de cada una de las `n` características del ejemplo concreto, así como un `string` que representa su clase o categoría.

```
struct Ejemplo {
    vector<double> caract;
    string clase;
    int n;
}
```

El conjunto de datos de entrada se encuentran en la carpeta `data`. Se trata de tres conjuntos de datos **ya normalizados** en formato `csv`, donde cada fila representa un ejemplo con los valores de sus características separados por `';'` y el último elemento de la fila es su clase.

Cada conjunto de datos es almacenado por un `vector<Ejemplo>`, y se emplea la función `read_csv` para rellenar el vector, este va leyendo los archivos línea a línea.

Además, como será necesario hacer particiones de cada conjunto de datos para implementar la técnica de *K-fold cross validation*, se proporciona la función `hacerParticiones` que se encarga de repartir los elementos entre los K conjuntos considerados, respetando la proporción original de clases.

Por su parte, la solución es vector de pesos que representaremos en un `vector<double>` del mismo tamaño que el número de categorías consideradas en cada caso.

Esquema de representación P3

Para desarrollar está ultima practica se ha implementado una nueva estructura `Solucion`, que consta de un vector de pesos `w` y un flotante `objetivo` en el que se guardará el valor de la función objetivo que consigue ese vector de pesos en el problema.

```
struct Solucion{
  vector<double> w;
  float objetivo;
};
```

Además se ha implementado un comparador de soluciones, para poder ordenar fácilmente los vectores de *Soluciones* según su valor *objetivo* en los últimos algoritmos. Este comparador devuelve si el valor objetivo del primer elemento es menor que el del segundo.

```
bool comparador_solucion(const Solucion& sol1, const Solucion& sol2){
  return sol1.objetivo < sol2.objetivo;
}
```

Operadores comunes P1

En esta sección describimos los operadores que se utilizan en todos los algoritmos o en su evaluación. La función clasificar1NN toma un vector de elementos ya clasificados, un vector de pesos y un elemento a clasificar; y calcula la clase del nuevo elemento utilizando el algoritmo de vecino más cercano. Hay que tener en cuenta que si usamos el mismo conjunto para entrenamiento y validación hay que usar la técnica *leave one out* para que el vecino más cercano no sea el propio elemento.

La función distancia con pesos calcula la distancia al cuadrado. Como la raíz cuadrada es una función creciente para calcular mínimos o máximos el resultado es equivalente.

Data: Vector se entreniamiento(ya_clasificados); Elemento a clasificar (elemento); Vector de pesos (pesos);
Result: Clase del vecino más cercano

```
1 vecino_cercano = 0;
2 distancia_minima = ∞ ;
3 for i ∈ ya_clasificados[] do
4     if ( i != indice(elemento) ) then
5         distancia = distanciaConPesos(elemento, i, pesos);
6         if distancia <distancia_minima then
7             distancia_minima = distancia;
8             vecino_cercano = i;
9 return ya_clasificados[vecino_cercano].clase;
```

Algoritmo 1: Clasificador 1-NN

Data: Elemento 1 (e1); Elemento 2 (e2);
Result: Distancia euclidia ponderada con pesos entre los dos elementos.

```
1 distancia = 0 ;
2 for i ∈ e1.n do
3     if ( w[i] >= 0.2 ) then
4         distancia += w[i] * (e2.caracteristica[i] − e1.caracteristica[i])^2;
5 return distancia;
```

Algoritmo 2: Distancia con pesos

Función objetivo

Como se dijo antes la función objetivo es la suma ponderada de precisión y simplicidad. En nuestro caso la constante alpha tiene un valor de 0,5.

La tasa de clasificación es el número de elementos bien clasificados.

La tasa de reducción mide el número de pesos que son menos significantes (con un valor menor de 0.2 en nuestro caso).

Data: Tasa de clasificación (tasa_class); Tasa de reducción (tasa_red);
Result: Valor de la función objetivo.

```
1 return alpha * tasa_clas + (1,0 − alpha) * tasa_red ;
```

Algoritmo 3: Función objetivo

Data: Vector clasificados; Vector test
Result: Tasa de clasificación.

```
1 aciertos = 0 ;
2 for i ∈ calsificados.size() do
3     if ( clasificados[i] == test[i].clase ) then
4         aciertos + +;
5 return 100.0 * aciertos / clasificados.size();
```

Algoritmo 4: Tasa de clasificación.

Data: Vector de pesos;
Result: Tasa de reducción.

```
1 descartados = 0 ;
2 for i ∈ pesos.size() do
3     if ( w[i] <0.2 ) then
4         descartados + +;
5 return 100.0 * descatados / pesos.size();
```

Algoritmo 5: Tasa de reducción

Operadores Comunes P3

En primer lugar se implementa la función **recalcularObjetivo**, una función para recalcular el valor de cada `Solucion` medido sobre el conjunto de entrenamiento, a esta función se le pasa una solución y un vector de `Ejemplo` y actualiza el valor de la función objetivo que consigue esta solución. Para actualizar el valor se utiliza la técnica *leave-one-out*, ya implementada en el clasificador con pesos. Esta es justo la métrica que se empleará para comparar dos vectores de pesos y decidir cuál de ellos es mejor.

En lo que sigue, cada vez que aparezca el símbolo "&"significará que este objeto se pasa por referencia y se cambia su valor, de esta forma no se devuelve ningún resultado sino que se modifica la entrada al pasar por la función.

```
Data: Solucion & (sol); vector de Ejemplo (entrenamiento);
clasificados = ∅;
for k ∈ entrenamiento.size() do
    clasificados.push_back( clasificar1NN(entrenamiento[k], entrenamiento,
     sol.w,k));
sol.objetivo = objetivo(tasaClas(clasificados, entrenamiento), tasaRed(sol.w));
```

Algoritmo 6: recalcularObjetivo

En segundo lugar, se implementa la función **inicializar_solucion**, la cual devuelve una solución correctamente inicilizada. Para ello recibe como entrada un vector de `Ejemplo` y un entero `n` que indica el tamaño del vector de pesos que debe tener la solución a inicializar. Los valores iniciales de los pesos de la nueva solución se obtienen de forma aleatoria de una distribución uniforme(0.0, 1.0), posteriormente se actualiza el valor objetivo de la solución, llamando a la función explicada anteriormente.

```
Data: vector de Ejemplo (entrenamiento); entero (n);
Result: Solucion
sol = ∅;
for i < n do
    sol.w[i] = random_real(0.0, 0.1);
recalcularObjetivo(sol, entrenamiento);
return sol;
```

Algoritmo 7: inicializar_solucion

Por último, se programa el operador de mutación `mutar`, al que se le pasa por referencia un vector de pesos, la componente a mutar y el parámetro *sigma* de la distribución normal que se va a utilizar (el valor de mu de la distribución normal si es el mismo para todos los algoritmos). El último parámetro es necesario para poder reutilizar la función en todos los algoritmos, ya que no todos utilizan la misma distribución normal para generar el valor aleatorio de la mutación. Este operador genera un valor aleatorio de la distribución normal y muta la componente `i` con el valor aleatorio, posteriormente trunca el resultado si este es negativo o mayor que 1 (resultados inválidos para un peso).

Data: vector de pesos & (w); entero (i); flotante (sigma);

```
w[i] += normal(0.0, sigma);
if w[i] < 0,0 then
    w[i] = 0,0 ;
if w[i] > 1,0 then
    w[i] = 1,0 ;
```

Algoritmo 8: mutar

3. Descripción de los algoritmos considerados

3.1. Algoritmos P1

En esta sección se comentan los algoritmos que se desarrollaron en la práctica 1 y que se utilizan en los análisis de resultados.

Algoritmo greedy RELIEF

Este algoritmo recorre todos los elementos del vector de ejemplos que se le pasa, y para cada elemento se modifica su peso asignado en función del vecino amigo y enemigo más cercano.

Data: Vector de elementos (ejemplos); Vector de pesos (pesos);

```
1  peso_max = 0 ;
2  for i ∈ ejemplos.size() do
3      buscarAmigoEnemigo(i, ejemplos, amigo, enemigo);
4      for j ∈ ejemplos[i].caract.size() do
5          pesos[j]+ = |ejemplos[i].caract[j] − enemigo.caract[j]| −
             |ejemplos[i].caract[j] − amigo.caract[j]|;
6  for p ∈ pesos do
7      if p >peso_max then
8          peso_max = p ;
9  for j ∈ pesos do
10     if pesos[j] <0 then
11         pesos[j] = 0;
12     else
13         pesos[j] = pesos[j] / peso_max;
```

Algoritmo 9: RELIEF

Para calcular el vecino amigo y enemigo más cercano, recorremos todo el vector de ejemplos sin pasar por el elemento que estamos clasificando, y calculamos el amigo (elemento de su misma clase) o enemigo (elemento de distinta clase) más cercano.

```
Data: Indice del elemento(posicion); Vector de elementos(ejemplos); Elemento
      amigo(amigo); Elemento enemigo(enemigo);
1  distanciaAmigoMin = distanciaEnemigoMin = ∞ ;
2  posicionAmigoMin = posicionEnemigoMin = 0;
3  for i ∈ ejemplos.size() do
4      if ( i != posicion ) then
5          distancia = distanciaEuclidea(ejemplos[i], ejemplos[posicion]);
6          if ejemplos[posicion].clase == ejemplos[i].clase then
7              if distancia <distanciaAmigoMin then
8                  distanciaAmigoMin = distancia;
9                  posicionAmigoMin = i;
10         else
11             if distancia <distanciaEnemigoMin then
12                 distanciaEnemigoMin = distancia;
13                 posicionEnemigoMin = i;
14 amigo = ejemplos[posicionAmigoMin];
15 enemigo = ejemplos[posicionEnemigoMin];
```

Algoritmo 10: distanciaAmigoEnemigo

Busqueda local

En primer lugar inicializamos el vector de pesos con un valor aleatorio extraído de una distribución uniforme de media 0 y desviación típica 0.3. Aprovechamos el bucle de incialización para rellenar un vector de índices de 1 a n (tamaño del vector de pesos). A continuación entramos en un bucle en el que en cada iteración mutamos uno de los pesos según una normal(0,0.3) (el que señale el vector de índices previamente mezclado), si se observa mejora nos quedamos con el vector de pesos nuevo, sino descartamos el cambio y volvemos a mutar. El bucle terminará cuando se hallan alcanzado 20 (MAX_VECINOS) * n o el número de iteraciones alcance el límite (en este caso 15000).

```
Data: Vector de elementos(entrenamiento); Vector de pesos (pesos);
numero_vecinos = iteraciones = 0 ;
mejora = false;
n = pesos.size();
for i ∈ pesos.size() do
    pesos[i] = distribución_uniforme;
    indices.push_back[i];
for k ∈ entrenamiento do
    clasificados.push_back(clasificar1NN(entrenamiento[k], entrenamiento, pesos,
     k));
mejor_obejtivo = objetivo(tasaClas(clasificados, entrenamiento), tasaRed(pesos));
clasificados.clear();
shuffle(indices);
while iteraciones <MAX_ITERACIONES and numero_vecinos <n * MAX_VECINOS
 do
    j = indices[iteracion % n];
    pesos_mutados = pesos;
    pesos_mutados[j] += normal(gen);
    if pesos_mutados[j] <0 then
        pesos_mutados[j] = 0;
    if pesos_mutados[j] >1 then
        pesos_mutados[j] = 1;
    for k ∈ entrenamiento do
        clasificados.push_back(clasificar1NN(entrenamiento[k], entrenamiento,
         pesos_mutados, k));
    obejtivo_actual = objetivo(tasaClas(clasificados, entrenamiento),
     tasaRed(pesos_mutados));
    clasificador.clear();
    if objetivo_actual >mejor_objetivo then
        mejor_obejtivo = obejtivo_actual;
        pesos = pesos_mutados;
        numero_vecino = 0;
        mejora = true;
    else
        numero_vecino ++;
    iteracion ++;
    if mejora or (iteracion % n == 0) then
        mejora = false;
        shuffle(indices);
```

Algoritmo 11: Busqueda local

3.2. Algoritmos P3

En esta sección se describen los algoritmos implementados para esta práctica.

3.2.1. Enfriamiento simulado

Con este algoritmo de búsqueda por trayectorias se busca evitar que la solución se quede atrapada en algún óptimo local, uno de los mayores problemas de algoritmos como la búsqueda local. Para solucionarlo, al principio se permiten aceptar soluciones peores que la mejor solución actual y con el paso de iteraciones afinar estas buscando la mejor solución, es decir cambiar la estrategia de diversificar a intensificar con el transcurso de iteraciones.

Para conseguir esta filosofía, el algoritmo de enfriamiento simulado trabaja con una *temperatura* que vas disminuyendo. Al principio de este algoritmo cuando la temperatura es mayor hay mayor posibilidad de aceptar soluciones peores que la solución actual (exploración), a medida que esta *temperatura* va bajando se reduce esta probabilidad (explotación). Con esto se consigue que el algoritmo explore soluciones alejándose de la mejor solución actual, evitando extremos locales. El número de enfriamientos los representamos con M, sin embargo, en ocasiones el algoritmo "se enfría"más de lo que debería debido a la condición extra sobre los éxitos).

En este algoritmo la solución inicial se construye como ya se comentó anteriormente, también quedó comentado el operador de mutación empleado. Como estamos maximizando la función objetivo a la hora de calcular la diferencia entre los valores objetivos debemos restarle el valor de la solución actual al valor de la solución mutada.

Además tenemos en cuenta de que la solución y la solución mutada tengan el mismo valor de la función objetivo, este hecho implica que el valor de la diferencia es 0, por lo tanto siempre se aceptaría la solución. Para solucionarlo le damos el valor $0{,}01$ a la diferencia, para que pueda haber posibilidad de que se rechace la mutación.

```
Data: vector de Ejemplo (entrenamiento), vector de doubles & (pesos)
sigma = 0.3;
it = 0;
sol = inicializar_solucion(entrenamiento, pesos.size());
mejor_sol = sol;
it ++ ;
temp_inicial = (0.3 * mejor_sol.objetivo) / (-1,0 * log(0,3) );
temp = temp_inicial;
M = 15000 / 10 * n;
while temp_final >= temp do
    temp_final = temp/100,0 ;
beta = (temp_inicial - temp_final) / (M * temp_inicial * temp_final);
exito = nS;
while it < 15000 and exito! = 0 do
    exito = 0;
    num_vec = 0;
    while it < 15000 and num_vec < 10 * n and exito < n do
        i = uniforme(0, pesos.size() -1);
        sol_mutada = sol;
        mutar(sol_mutada.w, i, sigma);
        recalcularObjetivo(sol_mutada, entrenamiento);
        it ++;
        num_vec ++;
        dif = sol.objeto - sol_mutada.objetivo;
        if dif == 0 then
            dif = 0.001;
        valor_aleatorio = uniforme(0.0,0.1);
        if dif < 0 or valor_aleatorio <= exp(-1,0 * dif/temp) then
            exito++;
            sol = sol_mutada;
            if sol.objetivo > mejor_sol.objetivo then
                mejor_sol = sol;
    temp = temp / (1,0+ beta * temp);
pesos = mejor_sol.w;
```

Algoritmo 12: Enfriamiento simulado

3.2.2. ILS

Este algoritmo se basa en aplicar de forma reiterada un algoritmo de búsqueda local a una solución previa obtenida por mutación de un óptimo local encontrado anteriormente. El algoritmo de búsqueda local utilizado es el de la P1, con pequeñas modificaciones y con un máximo de 1000 iteraciones.

A la hora de mutar utilizamos el operador de mutación comentado anteriormente, pero esta vez en vez de mutar solo una única componente, mutamos un porcentaje de ellas y todas distintas. Para conseguir esto último utilizamos un |set| de indices llamado `candidatos` en el que insertamos los indices ya mutados para no repetirlos. Además para mutar se utiliza una distribución uniforme con $simga = 0{,}4$ para conseguir valores más alejados de los obtenidos.

Por último, utilizamos como criterio de aceptación quedarnos con la mejor solución, buscando la intensificación o explotación del espacio de búsqueda.

```
Data: vector de Ejemplo (ejemplo); Solucion  sol;
sigma = 0.3;
n = sol.w.size();
mejor_objetivo = sol.objetivo;
iteracion = 0;
num_vecinos = 0;
mejora = false ;
indices = ∅;
for i < n do
    indices.push_back(i);
shuffle(indices);
while iteracion < 1000 and num_vecinos < n * 20 do
    j = indices[iteracion % n];
    sol_mutada = sol;
    mutar(sol_mutada.w, j, sigma);
    recalcularObjetivo(sol_mutada, entrenamiento);
    if sol_mutada.objetivo > mejor_objetivo then
        mejor_objetivo = sol_mutada.objetivo;
        sol = sol_mutada;
        num_vecinos = 0;
        mejor = true;
    else
        num_vecinos ++;
    iteracion ++ ;
    if mejora or (iteracion %n == 0) then
        mejora = false;
        shuffle(indices);
```

Algoritmo 13: búsqueda local reiterada

```
Data: vector de Ejemplo (entrenamiento); vector de double & (pesos)
sigma = 0.2;
sol = inicializar_solucion(entrenamiento, pesos.size());
busquedalocalreiterada(entrenamiento, sol);
for i < 15 do
    sol_mutada = sol ;
    indices_mutados = ∅;
    for j < 0,1 * n do
        while indices_mutados.size() == j do
            indice = uniforme(0, pesos.size() -1 );
            indices_mutados.insert(indice);
        mutar(sol_mutada.w, indice, sigma);
    recalcularObjetivo(sol_mutada, entrenamiento);
    busquedalocalreiterada(entrenamiento, sol_mutada);
    if sol_mutada.objetivo >sol.objetivo then
        sol = sol_mutada;
pesos = sol.w;
```

ALGORITMO 14: ILS

3.2.3. Evolución diferencial (DE)

Para este algoritmo, implementamos dos variantes que se diferencian a la hora de recombinar los valores del vector de pesos. Para el primer algoritmo **DE/rand/1** utilizamos la siguiente fórmula:

$$X_i^{t+1} = X_{r_1}^t + F(X_{r_2}^t - X_{r_3}^t),$$

donde X_i^t es el elemento $i-$ésimo de la población en la generación t, r_1, r_2, r_3 son índices aleatorios de la población y F es el factor de escala, que en nuestro caso vale $0{,}5$. Para el algoritmo **DE/current-to-best/1** la recombinación se puede expresar como

$$X_i^{t+1} = X_i^t + F(X_{\text{best}}^t - X_i^t) + F(X_{r_1}^t - X_{r_2}^t),$$

donde se ha introducido el término X_{best}^t, que es el mejor elemento de la población actual.

Para elegir los padres, utilizamos un operador de selección `seleccion_padres` que escoge 2 o 3 elementos (depende de la variante) de la población para posteriormente

cruzarlos (con probabilidad $0{,}5$) con la fórmula ya dicha. En esta selección se excluye de posibles candidatos el elemento de la población actual sobre el que se esta ejecutando el algoritmo principal.

```
Data: vector de Soluciones (poblacion); vector de Soluciones & seleccionados;
      entero (num_padres); entero (padre)
1 candidatos = emptyset;
2 for i < num_padres do
3     while candidatos.size() == i do
4         indice = uniforme(0, poblacion.size() - 1);
5         if indice ≠ padre then
6             candidatos.insert(indice);
7     seleccionados.push_back(poblacion[indice]);
```

Algoritmo 15: seleccion_padres

```
Data: vector de Ejemplo (entrenamiento); vector de double & (pesos)
poblacion = [];
n = pesos.size();
it = 0;
for i < 50 do
    poblacion.push_back(inicializar_solucion(entrenamiento, pesos.size()));
    it ++;
while it < 15000 do
    for i < 50 do
        selecion_padres(poblacion, padres, 3, i);
        elegido = uniforme(0, n - 1);
        for k < n do
            aleatorio = uniforme(0.0,0.1);
            if k == elegido or aleatorio < 0,5 then
                hijo.w[k] = padres[0].w[k] + F * (padres[1].w[k] - padres[2].w[k]);
                if hijo.w[k] < 0,0 then
                    hijo.w[k] = 0,0;
                if hijo.w[k] > 1,0 then
                    hijo.w[k] = 1,0;
            else
                hijo.w[k] = poblacion[i].w[k];
        recalcularObjetivo(hijo, entrenamiento);
        it ++ ;
        if hijo.objetivo >poblacion[i].objetivo then
            poblacion[i] = hijo ;
sort(poblacion);
pesos = poblacion[50 - 1].w;
```

Algoritmo 16: DE aleatoria

```
Data: vector de Ejemplo (entrenamiento); vector de double & (pesos)
poblacion = [];
n = pesos.size();
it = 0;
for i < 50 do
    poblacion.push_back(inicializar_solucion(entrenamiento, pesos.size()));
    it ++;
sort(poblacion);
mejor_sol = poblacion[50 - 1];
while it < 15000 do
    for i < 50 do
        selecion_padres(poblacion, padres, 2, i);
        elegido = uniforme(0, n - 1);
        for k < n do
            aleatorio = uniforme(0.0,0.1);
            if k == elegido or aleatorio < 0,5 then
                hijo.w[k] = poblacion[i].w[k] + F * (mejor_sol.w[k] -
                 poblacion[i].w[k]) +F * (padres[0].w[k] - padres[1].w[k]);
                if hijo.w[k] < 0,0 then
                    hijo.w[k] = 0,0;
                if hijo.w[k] > 1,0 then
                    hijo.w[k] = 1,0;
            else
                hijo.w[k] = poblacion[i].w[k];
        recalcularObjetivo(hijo, entrenamiento);
        it ++ ;
        if hijo.objetivo >poblacion[i].objetivo then
            poblacion[i] = hija ;
    sort(poblacion);
    mejor_sol = poblacion[50 - 1];
pesos = poblacion[50 - 1].w;
```

Algoritmo 17: DE ctb

4. Procedimiento considerado para desarrollar la práctica

La práctica se ha implementado en C(++11) y el ordenador sobre el que se han realizado las ejecuciones tiene sistema operativo Ubuntu 18, con procesador Intel Core i7-8750H @2.20GHz. Se utiliza la biblioteca `std` y otras bibliotecas auxiliares, pero no se ha hecho uso de ningún *framework* de metaheurísticas.

Para todos los procedimientos que implican aleatoriedad se utiliza un generador de números aleatorios común (llamado `gen`), inicializado con una semilla concreta. La semilla por defecto es 327, aunque se puede especificar otra mediante línea de comandos. Por otro lado se proporciona un makefile para compilar los archivos y generar un ejecutable para cada práctica, mediante la orden `make`. Si queremos compilar las tres prácticas utilizamos la orden `make` sin argumentos, si solo queremos compilar una de las tres prácticas, por ejemplo solo la práctica 1 ejecutaremos `make p1` (o p2,p3 en el otro caso). A la hora de ejecutar cualquiera de las dos prácticas hay dos opciones:

- Ejecutarlo sin argumentos. En este caso, utiliza la semilla por defecto y ejecuta los algoritmos sobre los tres conjuntos de datos de la carpeta `DATA`.
- Pasarle como parámetro una semilla para el generador aleatorio y ejecuta los algoritmos sobre los tres conjuntos de datos de la carpeta `DATA`.

Al compilar se generarán tres ejecutables en la carpeta `BIN` de nombre `p1`, `p2` y `p3`.

5. Experimentos y análisis de resultados

5.1. Descripción de los casos del problema

Se ha considerado los 3 casos del problema en los que se nos han proporcionado datos.

5.1.1. Ionosphere

Es una base de datos de radar recogidos por un sistema en Goose Bay, Labrador. Consta de 352 ejemplos, 34 atributos y 2 clases (retornos buenos y malos).

5.1.2. Colposcopy

La colposcopia es un procedimiento ginecológico que consiste en la exploración del cuello uterino. El conjunto de datos fue adquirido y anotado por médicos profesionales del Hospital Universitario de Caracas. Consta de 287 ejemplos, 62 atributos y dos clases (positivo o negativo).

5.1.3. Texture

El objetivo de este conjunto de datos es distinguir entre 11 texturas diferentes (césped, piel de becerro prensada, papel hecho a mano, rafia en bucle a una pila alta, lienzo de algodón,...). Consta de 550 ejemplos, 40 atributos y 11 clases (tipos de textura).

5.2. Resultados obtenidos

Todos los resultados se han redondeado a 2 cifras significativas. Cada fila se corresponde con una partición (se consideran las mismas particiones para todos los algoritmos). Las columnas de la tabla son:
N.º: el número de la partición.
Clas: la tasa de clasificación (precisión) en porcentaje.

Red: la tasa de reducción (simplicidad) en porcentaje.
Agr: La función agregada con $\alpha = 0{,}5$.
T: El tiempo en segundos.

Se ha abreviado los nombres de los últimos algoritmos, búsqueda local **(BL)**, enfriamiento simulado **(ES)**, búsqueda local iterativa **(ILS)** y por último las dos versiones de evolución diferencial, la versión aleatoria **(DE - rand)** y la versión current to best **(DE - ctb)**.

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	90.14	0	45.07	0.003	74.58	0	37.29	0.003	93.64	0	46.82	0.008
2	80	0	40	0.003	70.18	0	35.09	0.003	89.09	0	44.55	0.008
3	82.86	0	41.43	0.003	73.68	0	36.84	0.003	94.55	0	47.27	0.008
4	92.86	0	46.43	0.003	75.44	0	37.72	0.003	92.73	0	46.36	0.007
5	87.14	0	43.57	0.003	82.46	0	41.23	0.003	92.73	0	46.36	0.007
$\bar{x}$	86.60	0	43.30	0.003	75.27	0	37.63	0.003	92.55	0	46.27	0.007

Tabla 1: Algoritmo 1-NN

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	90.14	2.94	46.54	0.012	72.88	40.32	56.6	0.012	91.82	15	53.41	0.033
2	81.43	2.94	42.18	0.012	75.44	27.42	51.43	0.011	91.82	2.5	47.16	0.035
3	82.86	2.94	42.90	0.012	77.20	32.26	54.73	0.011	95.45	2.5	48.98	0.033
4	92.86	2.94	47.90	0.012	71.93	51.61	61.77	0.011	92.73	2.5	47.61	0.032
5	90	2.94	46.47	0.012	82.46	30.65	56.55	0.011	93.64	5	49.32	0.031
$\bar{x}$	87.46	2.94	45.20	0.012	75.98	36.45	56.22	0.011	93.09	5.5	49.30	0.033

Tabla 2: Algoritmo RELIEF

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	85.92	88.24	87.08	14.45	69.49	85.48	77.49	23.24	93.64	82.5	88.07	27.69
2	84.29	79.41	81.85	6.88	68.42	85.48	76.95	24.18	86.36	87.5	86.93	35.54
3	87.14	88.24	87.69	8.46	77.19	85.48	81.34	24.53	85.45	85	85.23	28.87
4	87.14	85.29	86.22	14.77	70.18	74.19	72.48	15.88	85.45	85	85.23	50.47
5	82.86	82.25	82.61	9.26	73.68	82.26	77.97	25.09	89.09	87.5	88.30	51.51
$\bar{x}$	85.47	84.71	85.09	11.36	71.79	82.58	77.19	22.58	88	85.5	86.75	38.82

Tabla 3: Algoritmo de búsqueda local

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	88.73	55.24	88.48	126.99	69.49	79.03	74.26	119.93	86.36	82.5	84.43	383.45
2	88.57	88.24	88.40	126.57	66.67	79.03	72.85	123.10	91.82	87.5	89.66	388.87
3	87.14	88.24	87.69	129.17	73.68	82.26	77.97	119.18	88.18	87.5	87.84	383.12
4	95.71	91.18	93.45	128.79	75.44	83.87	79.65	119.17	86.36	82.5	84.43	375.28
5	91.43	88.24	89.83	128.71	84.21	82.26	83.23	125.63	83.64	87.5	85.57	381.30
$\bar{x}$	90.32	88.82	89.57	128.05	73.90	81.29	77.59	121.40	87.27	85.5	86.39	382.40

Tabla 4: Algoritmo enfriamiento simulado

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	85.92	88.24	87.08	95.58	71.19	83.87	77.53	117.63	90	85	87.5	324.17
2	78.57	85.29	81.93	91.78	78.95	77.42	78.18	125.93	90	85	87.5	320.18
3	87.14	88.24	87.69	104.91	77.19	72.58	74.89	125.80	90.91	85	87.95	342.93
4	84.29	88.24	86.26	95.92	71.93	79.03	75.48	124.45	88.18	87.5	87.84	325.31
5	87.14	88.24	87.69	90.27	68.42	90.32	79.37	120.46	92.73	85	88.86	313.35
$\bar{x}$	84.61	87.65	86.13	95.63	73.54	80.65	77.09	122.85	90.36	85.5	87.93	325.19

Tabla 5: Algoritmo ILS

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	88.73	91.18	89.95	133.62	74.58	93.55	84.06	123.57	90	87.5	88.75	395.35
2	90	91.18	90.59	134.10	70.18	90.322	80.25	125.10	86.36	87.5	86.93	396.96
3	74.29	94.12	84.20	132.66	75.44	93.55	84.49	127.48	90.91	87.5	89.20	392.47
4	94.29	94.12	94.20	131.39	71.93	93.55	82.74	125.28	83.64	87.5	85.57	385.53
5	82.86	91.18	87.02	132.63	66.67	93.55	80.12	127.51	93.64	87.5	90.57	398.83
$\bar{x}$	86.03	92.35	89.19	132.88	71.76	92.90	82.33	125.79	88.91	87.5	88.20	393.83

Tabla 6: Algoritmo DE - rand

	Ionosphere				Colposcopy				Texture			
N.º	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1	83.10	85.29	84.20	134.71	77.97	75.81	76.89	125.88	84.55	77.5	81.02	408.16
2	84.29	79.41	81.85	136.84	77.19	72.58	74.89	127.21	88.18	85	86.59	395.70
3	88.57	88.24	88.40	130.07	70.18	70.97	70.57	130.04	88.18	80	84.09	391.44
4	94.29	79.41	86.85	135.30	77.19	80.65	78.92	128.70	83.64	77.5	80.57	419.44
5	91.43	79.41	85.42	138.90	71.93	70.97	71.45	132.57	91.82	77.5	84.66	399.87
$\bar{x}$	88.33	82.35	85.34	135.16	74.89	74.19	74.54	128.88	87.27	79.5	83.39	402.92

Tabla 7: Algoritmo DE - ctb

	Ionosphere				Colposcopy				Texture			
Algoritmo	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**	**Clas**	**Red**	**Agr**	**T(s)**
1-NN	86.60	0	43.30	0.00	75.27	0	37.63	0.00	92.55	0	46.27	0.01
RELIEF	87.46	2.94	45.20	0.01	75.98	36.45	56.22	0.01	93.09	5.5	49.30	0.03
BL	85.47	84.71	85.09	11.4	71.79	82.58	77.19	22.6	88	85.5	86.75	38.8
AGG-BLX	89.17	84.12	86.64	81.2	73.16	70	71.58	87.2	90.36	83	86.68	222
AGG-CA	88.04	71.76	79.90	82.2	74.58	53.33	64.06	87.2	90.18	69.5	79.84	225
AGE-BLX	89.90	84.12	85.51	82.2	74.58	76.06	76.32	88.2	88.91	83	85.95	225
AGE-CE	84.62	85.29	84.95	82.1	75.96	71.94	73.95	89.8	88.55	80	84.27	227
AM1	86.58	90	88.29	82.2	71.78	85.48	78.63	92.6	88.73	86	87.36	227
AM2	84.6	90	87.3	80.2	72.81	83.23	78.02	88.7	90.18	85	87.56	216
AM3	87.45	88.82	88.14	80.5	73.55	83.23	78.39	88.3	90	85	87.5	216
ES	90.32	88.82	89.57	128	73.90	81.29	77.59	121	87.27	85.5	86.39	382
ILS	84.61	87.65	86.13	95.6	73.54	80.65	77.09	123	90.36	85.5	87.93	325
DE-rand	86.03	92.35	89.19	133	71.76	92.90	82.33	126	88.91	87.5	88.20	394
DE-ctb	88.33	82.35	85.34	135	74.89	74.19	74.54	129	87.27	79.5	83.39	403

Tabla 8: Algoritmos

5.3. Análisis de los resultados

5.3.1. Casos base

En primer lugar los resultados del algoritmo **1-NN**, como utilizan un vector de pesos con todo 1 su *simplicidad* es mínima. Por lo tanto la tasa *agregada* es bastante baja ya que como mucho puede llegar al $50\,\%$. Por otro lado la *tasa de clasificación* es bastante alta, sobre todo en el conjunto de datos *texture*, llegando al $92,55\,\%$. Por último destacar que el *tiempo de ejecución* es bastante bajo, llegando como máximo a $7,54$ milésimas de segundo.

5.3.2. Algoritmos considerados en la práctica

Análisis P1

En el algoritmo *greedy* **RELIEF** podemos apreciar también una *tasa de clasificación* alta, llegando al $93{,}10\,\%$ en el mejor de los casos (texture), superando además los resultados del 1-NN en todos los casos. Sin embargo la *tasa de reducción* es baja, a excepción de el conjunto de datos Colposcopy que llega a un $36{,}45\,\%$, esto hace que la tasa *agregada* alcance como mucho el $56{,}22\,\%$. Cabía de esperar que este algoritmo no diera los mejores resultados tratándose de un algoritmo *greedy*, sin embargo su bajo tiempo comparado con el último algoritmo hace que sea un algoritmo rápido que podemos probar como estrategia inicial.

Por último el algoritmo de **búsqueda local**, es el que mejores resultados a cambio de un gran coste de tiempo. Este algoritmo consigue una *tasa de clasificación* un poco peor que la del algoritmo RELIEF, sin embargo consigue una mejora significante en la *tasa de reducción*, alcanzando hasta un $85{,}5\,\%$ de mejora respecto de el anterior algoritmo. Estos dos fenómenos, hacen que este último algoritmo consiga la mejor tasa de *agregación* de los tres algoritmos, llegando hasta un $87{,}13\,\%$ en el mejor de los casos. Esto nos muestra que el algoritmo de búsqueda local es capaz de buscar con mucha más precisión en el espacio de soluciones, obteniendo una convergencia rápida hacia un máximo local.

Como conclusión y observando los resultados globales, podemos observar:

- El algoritmo **RELIEF** alcanza casi la misma precisión con un tiempo de ejecución mucho menor. Sin embargo en los casos de datos probados no consigue buena simplicidad.

- La **búsqueda local** consigue la mejor *tasa agregada*, ya que consigue mantener la *tasa de clasificación* del algoritmo *greedy* y aumentar la *tasa de reducción*. Su tiempo es bastante más elevado que el algoritmo RELIEF, aunque este varíe mucho al cambiar las condiciones iniciales, las cuales son aleatorias.

- La **función objetivo** da mucho valor a la simplicidad del vector de pesos, por eso algoritmos como el aleatorio consiguen un valor de la *tasa de agregación* bastante elevado, cuando el número de clases del conjunto de datos es pequeño. Esto nos indica que posiblemente la función objetivo no mida correctamente la calidad de la solución y sea necesario modificarla cambiando por ejemplo el parámetro α.

Análisis P3

Comenzamos analizando los resultados del primer algoritmo, **enfriamiento simulado**. Este algoritmo consigue muy buenos resultados en los tres conjuntos de datos, consiguiendo el mejor resultado en *Ionosphere* de todos los algoritmos programados en las tres prácticas. Si lo comparamos con la búsqueda local de la práctica 1 los resultados son parecidos, siendo en alguno conjuntos mejor el algoritmo de enfriamiento simulado y en otros casos la búsqueda local.

Por otro lado el algoritmo de **búsqueda local reiterada (ILS)** también consigue resultados bastante buenos, superando incluso al algoritmo de enfriamiento simulado en el conjunto *Texture*. Si lo comparamos con la búsqueda local simple que implementamos en la práctica 1, vemos que como cabía esperar la supera en la mayoría de los casos en cuanto a tasa agregada y tasa de reducción. La explicación es sencilla: al realizar una técnica multiarranque podemos escapar de óptimos locales (exploración), y el hecho de realizar después una búsqueda local nos permite intensificar la explotación del espacio de soluciones.

Por último analizamos los dos algoritmos de **evolución diferencial**. Nos damos cuenta de que la primera de las dos versiones, la versión aleatoria, consigue con diferencia los mejores resultados de todos los algoritmos programados. Esto se puede deber a la capacidad de escapar de óptimos locales, gracias a la gran componente aleatoria a la hora de recombinar. Sin embargo la segunda versión empeora bastante, llegando

a ser superada por la búsqueda local en los conjuntos de *Colposcopy* y *Texture*. Esto puede deberse a que introducimos una gran presión selectiva en el esquema de reemplazamiento, obteniendo una convergencia prematura hacia un óptimo local. Este hecho lo podemos ver mejor en el siguiente gráfico.

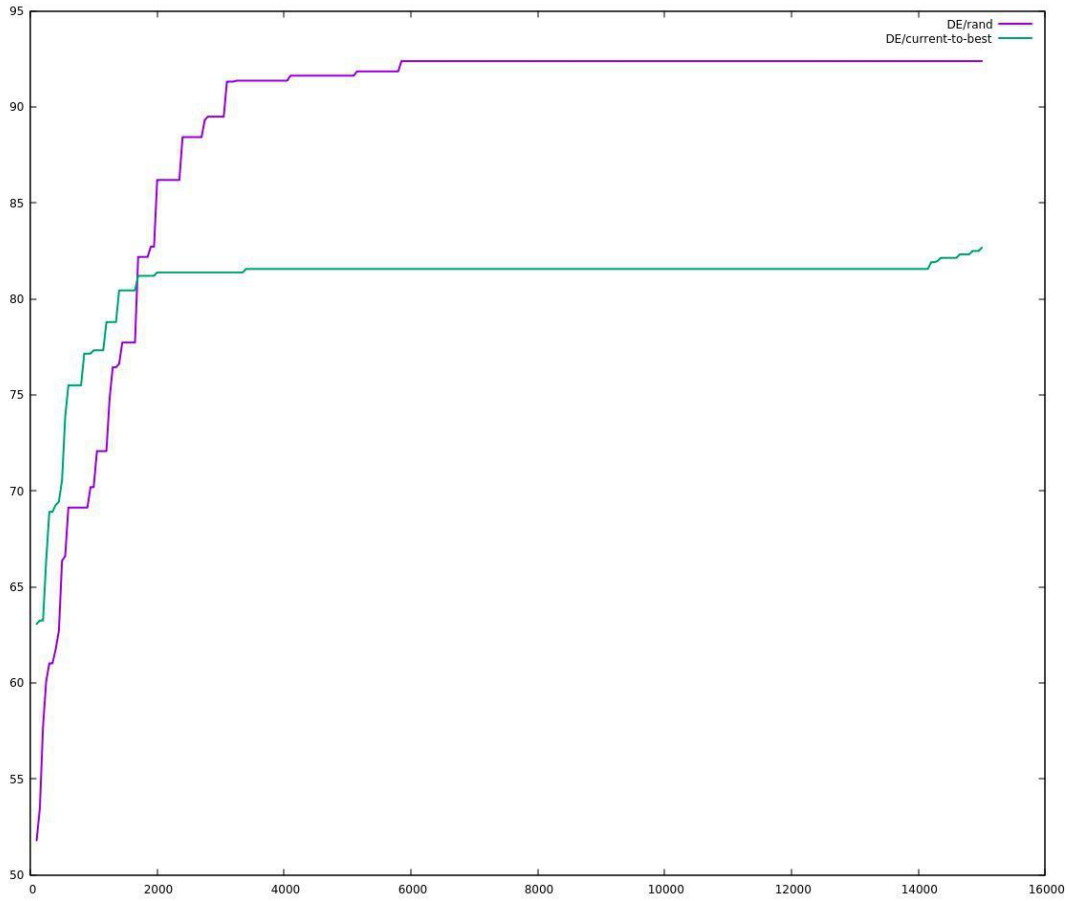

FIGURA 1: Comparación en una ejecución concreta de los algoritmos de DE en *ionosphere*.

Por último, podemos observar que en general los algoritmos implementados en esta práctica son mejores que los implementados en la práctica 1, y bastante competitivos (y en muchos casos mejores) con los algoritmos genéticos y meméticos de la práctica 2.

Para terminar si nos fijamos en el tiempo de ejecución, los algoritmos de la práctica 3 tardan más o menos el mismo tiempo. Aunque si nos fijamos en los resultados del agregado, puede q1ue esta inversión de tiempo merezca la pena.