Metaheurísticas

Aprendizaje de Pesos en Características

Práctica 3: Enfriamiento simulado, búsqueda local reiterada y evolución diferencial

Antonio Coín Castro
XXXXXXXXXXZ
XXXXXXXXXX@correo.ugr.es
Grupo 1 (M 17:30-19:30)

8 de junio de 2019

Índice

1. Descripción del problema **3**

2. Descripción de la aplicación de los algoritmos **4**
Esquemas de representación P3 4
Operadores comunes P3 6

3. Descripción de los algoritmos considerados **8**
Enfriamiento simulado (ES) 8
Búsqueda local reiterada (ILS) 9
Evolución diferencial (DE) 10

4. Procedimiento considerado para desarrollar la práctica **13**

5. Algoritmos de comparación **14**

6. Resultados **17**
Descripción de los casos del problema 17
Resultados obtenidos 17
Análisis de resultados P3 21

Con el fin de mantener en un único documento toda la información relevante para esta práctica se han incluido en esta memoria las explicaciones y descripciones de los algoritmos que ya se comentaron en la memoria de las prácticas anteriores, así como el análisis de resultados que ya se realizó. Para separar esta información repetida se marcan en rojo las secciones que no aportan nada nuevo a esta práctica.

1. Descripción del problema

En todo el desarrollo de las prácticas consideraremos el marco general de un problema de clasificación. Fijado $n \in \mathbb{N}$, un *clasificador* es cualquier función $c : \mathbb{R}^n \longrightarrow C$, donde C es un conjunto (finito) de clases prefijadas. Consideramos además un *conjunto de entrenamiento* $T \subseteq \mathbb{R}^n$ de elementos ya clasificados: para cada $t \in T$, denotamos $\Gamma(t) \in C$ a su clase, que es conocida.

El problema de clasificación consiste en, dado un *conjunto de prueba* $T' \subseteq \mathbb{R}^n$ no observado previamente, encontrar un clasificador c que maximice el número de clases correctamente clasificadas en T', tras haber sido entrenado sobre los elementos de T.

Uno de los clasificadores más conocidos y más sencillos es el clasificador k–NN, que asigna a cada elemento la clase que más se repite entre sus k vecinos más cercanos. En el caso concreto de esta práctica trabajaremos con el clasificador 1–NN **con pesos**: a cada elemento le asignamos la clase de su vecino más cercano, pero ponderamos la distancia en función de un vector de pesos $w \in [0,1]^n$.

Para el cálculo de la distancia entre dos elementos de $\mathbb{R}^n$ utilizaremos la *distancia euclídea* ponderada por el ya mencionado vector de pesos:

$$d_w(t,s) = \sqrt{\sum_{i=1}^{n} w_i(s_i - t_i)^2}, \quad t,s \in \mathbb{R}^n.$$

La idea tras estos pesos es que midan la importancia de cada una de las características que representan las coordenadas de los elementos n–dimensionales considerados, asignando más peso en el cálculo de la distancia a aquellas que sean más importantes. El problema de **aprendizaje de pesos en características** persigue justamente ''aprender" cuál debe ser el valor de cada peso en una instancia concreta del problema.

Para medir la bondad de un clasificador con pesos utilizamos las siguientes métricas:

- La **precisión** (T). Estudiamos cuántos ejemplos del conjunto de prueba se clasifican correctamente, entrenando previamente el clasificador (que utiliza la distancia d_w) con el conjunto de entrenamiento.
- La **simplicidad** (R). Un clasificador será más simple si tiene en cuenta un menor número de características. Diremos que una característica $i \in \{1, \ldots, n\}$ no se considera en el cálculo de la distancia si su peso asociado w_i es menor que 0.2.

Así, el problema consiste en encontrar un vector de pesos $w \in [0,1]^n$ que maximice la precisión y la simplicidad, es decir, que maximice lo que llamaremos la *función objetivo*:

$$F(w) = \alpha T(w) + (1-\alpha)R(w).$$

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

En primer lugar, los datos de entrada se encuentran en la carpeta DATA. Consisten en 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.

Para representar los datos en el programa se emplea una estructura `Example` 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 Example {
    vector<double> traits;
    string category;
    int n;
};
```

Cada conjunto de datos se representa entonces por un `vector<Example>`, y se emplea la función `read_csv` para rellenar el vector, que 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 `make_partitions` que se encarga de repartir los elementos entre los *K* conjuntos considerados, respetando la proporción original de clases. La forma de hacer esto es simplemente ir poniendo cada clase de forma cíclica en las particiones, primando el reparto equitativo de clases al reparto equitativo de elementos.

Por su parte, la solución es un `vector<double>` del mismo tamaño que el número de categorías consideradas en cada caso. La componente *i*–ésima del vector representa el peso otorgado a la característica *i*–ésima de cada ejemplo del problema.

Esquemas de representación P3

Añadimos una estructura de datos que representa una solución: un vector de pesos junto con su valor de la función objetivo o *fitness*:

```
struct Solution {
    vector<double> w;
    float fitness;
};
```

Además, para representar una población (para el algoritmo de evolución diferencial) definimos una estructura que engloba varias soluciones, junto con un comparador para ellas, similar al que se empleó para la práctica anterior. El orden definido entre soluciones es el que viene determinado por su *fitness*.

```
typedef vector<Solution> Population;

struct SolutionComp {
  bool operator()(const Solution& lhs, const Solution& rhs) {
    return lhs.fitness < rhs.fitness;
  }
} solution_comp;
```

Operadores comunes P1

Todos los algoritmos hacen uso del cálculo de la distancia. Como dijimos, para este cálculo se emplea la distancia euclídea, eventualmente ponderada mediante un vector de pesos. En el caso de que la distancia deseada sea la estándar, se asume que los pesos valen siempre 1 (en la implementación realmente hay dos funciones separadas, una con pesos y otra sin pesos).

---

function DISTANCE_SQ_WEIGHTS(e1, e2, w)
    distance = 0
    for i := 0 to $n - 1$ **do** ▷ n es el número de características
        if w[i] $\geq$ 0.2 **then**
            distance += w[i] $*$ (e2[i] - e1[i]) $*$ (e2[i] - e1[i])
    return distance

Cabe destacar que en realidad estamos calculando al distancia euclídea al cuadrado, pues solo vamos a utilizarla para comparar. Como la función $f(x) = \sqrt{x}$ es creciente para $x \geq 0$ no hay problema en que hagamos esto, pues se mantiene el orden. De esta forma ahorramos tiempo de cálculo, pues esta función va a ser llamada muchas veces a lo largo del programa.

También tenemos la función `classifier_1nn_weights`, que clasifica un ejemplo basándose en la técnica del vecino más cercano. Debemos pasarle también el conjunto de entrenamiento con los ejemplos ya clasificados, y el vector de pesos. De nuevo, si queremos que el clasificador no tenga en cuenta los pesos podemos asumir que son todos 1, aunque en realidad hay dos funciones separadas.

En lo que sigue, para generar números aleatorios reales o enteros utilizamos los tipos predefinidos `uniform_real_distribution` y `uniform_int_distribution`, respectivamente.

También se añade una función para *evaluar* un vector de pesos sobre un conjunto de entrenamiento, que devuelve el valor de la función objetivo o *fitness* que obtiene dicho vector. Esta es

justo la métrica que vamos a usar para comparar dos vectores de pesos y decidir cuál de ellos es mejor. Es necesario emplear la técnica *leave-one-out*, ya implementada en el clasificador con pesos.

```
function CLASSIFIER_1NN_WEIGHTS(e, training, self, w)
    selected = 0
    dmin = ∞
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos de entrenamiento
        if i ≠ self then
            dist = distance_sq_weights(e, training[i], w)
            if dist < dmin then
                dmin = dist
                selected = i
    return training[selected].category
```

```
function EVALUATE(training, w)
    classified = ∅
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos de entrenamiento
        classified.push_back(classifier_1nn_weights(training[i], training, i, w))
    return objective(class_rate(classified, training), red_rate(w))
```

Operadores comunes P3

En primer lugar, disponemos de una función `init_solution` que se encarga de generar una solución inicial aleatoria, siguiendo como se pide una distribución uniforme en $[0, 1]$ para los pesos. Esta solución generada se devuelve ya correctamente evaluada.

```
function INIT_SOLUTION(training, n)
    sol = ∅
    for i := 0 to n − 1 do                    ▷ n es el tamaño del vector de pesos
        sol.w[i] = random_real(0, 1)
    sol.fitness = evaluate(training, sol.w)
    return sol
```

También necesitaremos en los tres algoritmos una forma de mutar una solución. Empleamos el esquema de mutación normal descrito en la primera práctica, que consiste en sumarle a una componente del vector de pesos un valor extraído de una distribución normal de media 0 y desviación típica σ (varía según el algoritmo).

Notamos que al mutar los pesos pueden salirse del intervalo $[0, 1]$, por lo que es necesario truncarlos para obtener soluciones que sigan siendo válidas.

```
function MUTATE(w, comp, sigma)
    w[comp] += normal(0, σ)
    if w[comp] < 0 then
        w[comp] = 0
    if w[comp] > 1 then
        w[comp] = 1
    return w
```

Función objetivo

La función objetivo que queremos maximizar se implementa tal y como se dijo en la descripción del problema, donde el valor α prefijado es de 0.5, dando la misma importancia a la precisión y a la simplicidad.

```
objective(class_rate, red_rate) {
  return alpha * class_rate + (1.0 - alpha) * red_rate;
}
```

Para calcular la tasa de clasificación y de reducción utilizamos otras funciones también muy sencillas. La primera mide el porcentaje de acierto sobre un vector de elementos que el clasificador ha clasificado, y cuya clase real conocemos. La segunda simplemente contabiliza qué porcentaje de los pesos son menores que 0.2.

```
function CLASS_RATE(classified, test)
    correct = 0
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos clasificados
        if classified[i] == test[i].category then
            correct++
    return 100.0 * correct / n
```

```
function RED_RATE(w)
    discarded = 0
    for i := 0 to n − 1 do                    ▷ n es el tamaño del vector de pesos
        if w[i] < 0.2 then
            discarded++
    return 100.0 * discarded / n
```

3. Descripción de los algoritmos considerados

En esta sección se describen los algoritmos implementados en esta práctica para el problema del APC. En todos ellos lo que se pretende es rellenar un vector de pesos para maximizar la función objetivo.

Enfriamiento simulado (ES)

El algoritmo de enfriamiento simulado es un algoritmo de búsqueda por trayectorias que persigue paliar uno de los problemas más acusados de la búsqueda local: la caída en óptimos locales. Esto lo hace permitiendo aceptar soluciones peores que la mejor solución actual, aplicando la filosofía de diversificar al principio e intensificar al final.

Para llevar a cabo este procedimiento nos basamos en un modelo termodinámico en el que hay una "temperatura" figurada que va disminuyendo (enfriando), de forma que a mayor temperatura, mayor probabilidad de aceptación de soluciones peores. Con el aumento de iteraciones disminuye la probabilidad de aceptar soluciones peores que la actual. Así, el algoritmo acepta soluciones mucho peores que la actual al principio de la ejecución (exploración) pero no al final (explotación).

Para controlar la probabilidad de aceptación de soluciones utilizamos el modelo de Metrópolis. En este modelo se genera una perturbación (mutación) aleatoria en el sistema y se calculan los cambios de energía resultantes: si hay una caída energética (mejora la solución), el cambio se acepta automáticamente; por el contrario, si se produce un incremento energético (empeora la solución), el cambio será aceptado con una probabilidad $p = e^{-\Delta f / T}$, donde Δf es la diferencia de la función a maximizar sobre las soluciones y T es la temperatura actual del sistema.

A la hora de implementar el algoritmo, debemos tener en cuenta que estamos **maximizando** la función objetivo, por lo que a la hora de calcular la diferencia entre el *fitness* de la solución actual y la mutada debemos ajustar el orden en consecuencia. También modificamos la generación de la temperatura inicial para tener en cuenta este hecho.

La **mutación** de una solución se realiza siguiendo el esquema de mutación normal ya explicado en la sección de operadores comunes. La solución inicial se construye de forma aleatoria.

Un último detalle a tener en cuenta es que es posible una solución y su mutación tengan el mismo valor de la función objetivo. En este caso, el modelo de Metrópolis aceptaría siempre la mutación, pero nosotros introducimos una pequeña perturbación en la diferencia para evitar que sea exactamente 0, y exista la posibilidad de que la mutación sea rechazada.

El valor de M pretende representar el número de enfriamientos a realizar para alcanzar el número de iteraciones impuesto, aunque en ocasiones el algoritmo enfría más de la cuenta (debido a la condición extra sobre los éxitos).

```
function SIMULATED_ANNEALING(training)
    sol = init_solution(training, n)                    ▷ n es el número de características
    best_sol = sol
    T_0 = (0.3 · (1– best_sol.fitness / 100)) / – log(0.3)
    T = T_0
    iter = 1
    successful = 1
    M = 15000 / 10 · n
    while T_f ≥ T do                                    ▷ Debe ser T_0 > T_f
        T_f = T / 100
    β = (T_0 – T_f) / (M · T_0 · T_f)
    while iter < 15000 and successful ≠ 0  do
        neighbour = 0
        successful = 0
        while iter < 15000 and neighbour < 10 · n and successful < n do
            sol_mut = mutate(sol, random_int(0, 1), 0.3)
            sol_mut.fitness = evaluate(training, sol_mut.w)
            iter++
            neighbour++
            Δf = sol.fitness - sol_mut.fitness
            if Δf == 0 then
                Δf = 0.001
            if diff < 0 or random_real(0, 1) ≤ e^(−Δf/T) then      ▷ Criterio de Metrópolis
                successful++
                sol = sol_mut
                if sol.fitness > best_sol.fitness then
                    best_sol = sol
        T = T / (1 + β · T)                             ▷ Esquema de enfriamiento Cauchy modificado
    return best_sol.w
```

Búsqueda local reiterada (ILS)

La técnica de ILS está basada en la aplicación repetida de un algoritmo de búsqueda local a una solución inicial que se obtiene por mutación brusca de un óptimo local previamente encontrado. En nuestro caso, se utiliza la búsqueda local de la práctica 1, con un máximo de 1000 iteraciones. El pseudocódigo está en la sección de algoritmos de comparación.

Para la mutación de ILS empleamos la misma técnica que venimos usando hasta el momento, pero en vez de sobre una única componente lo hacemos sobre un porcentaje de estas, todas **distintas**. Además, incrementamos el valor de la desviación típica a 0.4 para intentar obtener valores más alejados de los originales.

Para evitar repetir componentes a mutar utilizamos un `set` llamado `mutated` (que no admite repetidos) en el que insertamos las componentes que ya se han mutado.

```
function ILS(training)
    sol = init_solution(training, n)                    ▷ n es el número de características
    s = local_search(training, s)
    for i := 1 to N − 1 do                              ▷ el número de iteraciones es N = 15
        s_mut = s
        mutated = ∅
        for j := 0 to ⌊0.1 · n⌋ − 1 do                  ▷ Mutación "brusca"
            while mutated.size() == j do
                comp = random_int(0, n − 1)
                mutated.insert(comp)
            mutate(s_mut.w, comp, 0.)
        s_mut.fitness = evaluate(training, s_mut.w)
        s_mut = local_search(training, s_mut)           ▷ Búsqueda local reiterada
        if s_mut.fitness > s.fitness then               ▷ Criterio de aceptación
            s = s_mut
    return s.w
```

El criterio de aceptación de soluciones es el de la **mejor solución**. Este criterio favorece la intensificación o explotación del espacio de búsqueda.

Evolución diferencial (DE)

En este caso estamos ante un algoritmo basado en poblaciones que implementa un operador de cruce consistente en generar un vector mutado (según una cierta probabilidad) añadiendo un vector diferencia, escalado y aleatoriamente muestreado, a un vector base aleatoriamente seleccionado de la población actual.

Proporcionamos la implementación de dos variantes de este algoritmo, que difieren únicamente en la estrategia de recombinación de vectores. Para el algoritmo **DE/rand/1** la recombinación sigue la 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.

Necesitaremos además un operador de selección de padres, que escoja aleatoriamente de la población actual un número determinado de elementos para cruzar. Esta selección se hace sin reemplazamiento, y excluyendo de los candidatos el elemento de la población actual sobre el que se ejecuta el algoritmo general.

```
function SELECT_PARENTS(pop, num, self)
    candidates = ∅                                  ▷ es un set sin repetidos
    for i := 0 to num–1 do                          ▷ num es el número de padres a seleccionar
        while candidates.size() == i do
            index = random_int(0, num − 1)
            if index ≠ self then
                candidates.insert(index)
        selected.push_back(pop[index])
    return selected
```

Pasamos ya a describir el algoritmo DE/rand/1. Como en los algoritmos anteriores, la población inicial de soluciones se genera de forma aleatoria. El reemplazamiento es *one-to-one*, es decir, padre e hijo compiten por permanecer en la población.

```
function DE_RAND(training)
    iter = 0
    for i := 0 to N − 1 do                          ▷ El tamaño de la población es N = 50
        pop.push_back(init_solution)
        iter++
    while iter < 15000 do
        for i := 0 to N − 1 do
            parents = select_parents(pop, 3, i)
            chosen = random_int(0, n - 1)           ▷ n es el número de características
            for k := 0 to n − 1 do
                if k == chosen or random_real(0, 1) ≤ CR then          ▷ CR = 0.5
                    offspring.w[k] = parents[0].w[k] + F · (parents[1].w[k] - parents[2].w[k])
                    truncate(offspring.w[k])                           ▷ Truncar a [0, 1].
                else
                    offspring.w[k] = pop[i].w[k]
            offspring.fitness = evaluate(training, offspring.w)
            iter++
            if offspring.fitness >pop[i].fitness then                  ▷ reemplazo
                pop[i] = offspring
    sort(pop, solution_comp)        ▷ ordenamos la población de menor a mayor por su fitness
    return pop[N-1].w
```

Observamos que empleamos el esquema de recombinación binomial, mediante el cual primero generamos un índice aleatorio que siempre cambia, y para el resto consideramos una probabi-

lidad de cruce de 0.5. También debemos tener en cuenta que al aplicar la mutación queremos obtener soluciones válidas (en otro caso truncamos a $[0, 1]$).

El algoritmo DE/current-to-best/1 es muy similar al ya mostrado, cambiando el esquema de recombinación a la segunda alternativa mencionada anteriormente. En este caso hemos de seguirle la pista al mejor elemento de la población actual.

```
function DE_CURRENT_TO_BEST(training)
    iter = 0
    for i := 0 to N − 1 do                          ▷ El tamaño de la población es N = 50
        pop.push_back(init_solution)
        iter++
    sort(pop, solution_comp)        ▷ ordenamos la población de menor a mayor por su fitness
    current_best = pop[N-1]
    while iter < 15000 do
        for i := 0 to N − 1 do
            parents = select_parents(pop, 2, i)
            chosen = random_int(0, n - 1)                ▷ n es el número de características
            for k := 0 to n − 1 do
                if k == chosen or random_real(0, 1) ≤ CR then                        ▷ CR = 0.5
                    offspring.w[k] = pop[i].w[k] + F · (current_best.w[k] - pop[i].w[k])
                                   + F · (parents[0].w[k] - parents[0].w[k])
                    truncate(offspring.w[k])                                ▷ Truncar a [0, 1].
                else
                    offspring.w[k] = pop[i].w[k]
            offspring.fitness = evaluate(training, offspring.w)
            iter++
            if offspring.fitness >pop[i].fitness then                              ▷ reemplazo
                pop[i] = offspring
        sort(pop, solution_comp)
        current_best = pop[N-1]
    return current_best.w
```

4. Procedimiento considerado para desarrollar la práctica

Todo el código de la práctica se ha desarrollado en C++ siguiendo el estándar 2011. 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 aleatoreidad se utiliza un generador de números aleatorios común (llamado `generator`), inicializado con una semilla concreta. La semilla por defecto es 2019, aunque se puede especificar otra mediante línea de comandos. La evaluación de todos los algoritmos considerados se realiza mediante la función `run_p3`, que también se encarga de recoger las estadísticas oportunas.

Se proporciona un makefile para compilar los archivos y generar un ejecutable (con optimización `-O3`), mediante la orden `make p3`. A la hora de ejecutarlo hay dos opciones:

- Pasarle como parámetro una semilla para el generador aleatorio, y a continuación una lista de archivos sobre los que ejecutar los algoritmos (ruta relativa).
- 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`.

El código está disponible en la carpeta `FUENTES`, y consta de los siguientes módulos:

- `p3.cpp` Contiene la implementación de los algoritmos, y las funciones necesarias para ejecutarlos.
- `timer` Módulo para medir tiempos en UNIX.
- `util` Se trata de funciones auxiliares para el preprocesamiento de los archivos de datos, el cálculo de la distancia, etc.

Al compilar se genera un único ejecutable en la carpeta `BIN` de nombre `p3`.

Todas las ejecuciones se han realizado en un máquina con sistema operativo `Linux` y procesador Intel Core i5-7200U @ 2.5GHz.

5. Algoritmos de comparación

Para comparar las metaheurísticas implementadas en esta práctica usaremos los algoritmos implementados en la práctica 1, a saber: el clasificador 1–NN sin pesos, el algoritmo RELIEF y la búsqueda local. También compararemos en ciertos casos con los algoritmos genéticos y meméticos de la práctica 2.

Clasificador 1-NN sin pesos

Ya se ha descrito en la sección de operadores comunes (podemos considerar que es un clasificador con pesos, donde todos los pesos valen 1).

Algoritmo *greedy* RELIEF

Este es un algoritmo voraz muy sencillo, que servirá como caso base para comparar las diferentes metaheurísticas desarrolladas. Se trata de buscar, para cada ejemplo, su amigo (misma clase) y su enemigo (distinta clase) más cercano. Después, componente a componente se suma al vector de pesos la distancia a su enemigo, y se resta la distancia a su amigo.

En este proceso es posible que los pesos se salgan del intervalo $[0, 1]$, por lo que al finalizar es necesario normalizarlos. Además, no debemos olvidar que el algoritmo comienza con el vector de pesos relleno de 0s. Disponemos para ello de una función `init_vector` que se encarga de darle un valor inicial de 0 a todas las componentes del vector de pesos.

Separamos la función que se encarga de buscar los amigos y enemigos más cercanos, teniendo en cuenta que el amigo más cercano no puede ser el propio ejemplo.

```
function NEAREST_EXAMPLE(training, e, self)
    dmin_friend = ∞
    dmin_enemy = ∞
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos de entrenamiento
        if i ≠ self then
            dist = distance_sq(e, training[i])
            if training[i].category ! = e.category and dist <dmin_enemy then
                n_enemy = i
                dmin_enemy = dist
            else if training[i].category == e.category and dist <dmin_friend) then
                n_friend = i
                dmin_friend = dist
    return n_enemy, n_friend
```

El algoritmo RELIEF se detalla ya en la página siguiente.

```
function RELIEF(training)
    w = init_vector()
    for i := 0 to n − 1 do                          ▷ n es el número de ejemplos de entrenamiento
        n_enemy, n_friend = nearest_example(training, training[i], i)
        for j := 0 to m − 1 do                      ▷ m es el tamaño del vector de pesos
            w[j] = w[j] + |training[i].traits[j] - training[n_enemy].traits[j]|
                  − |training[i].traits[j] - training[n_friend].traits[j]|
    max = max(w)
    for j := 0 to m − 1 do                          ▷ normalizamos los pesos
        if w[j] < 0 then
            w[j] = 0
        else
            w[j] = w[j] / max
    return w
```

Búsqueda local

Empleamos la técnica de búsqueda local del **primer mejor** para rellenar el vector de pesos. La idea es mutar en cada iteración una componente aleatoria y **distinta** del vector de pesos, sumándole un valor extraído de una normal de media 0 y desviación típica $\sigma = 0.3$. Si tras esta mutación se mejora la función objetivo, nos quedamos con este nuevo vector, y si no lo desechamos. Si algún peso se sale del intervalo [0, 1] tras la mutación, directamente lo truncamos a 0 ó a 1.

A la hora de escoger qué componente vamos a mutar, tenemos un vector de índices del mismo tamaño que el vector de pesos, que barajamos de forma aleatoria y recorremos secuencialmente. Si llegamos al final, volvemos a barajarlo para seguir generando nuevas soluciones.

Para escoger el valor con el que se muta cada componente utilizamos esta vez el tipo predefinido `normal_distribution<double>`. Para determinar si una mutación mejora, utilizamos como métrica el valor de la función objetivo, tomando la tasa de clasificación sobre el propio conjunto de entrenamiento (*leave-one-out*).

Hemos adaptado ligeramente este algoritmo con respecto al implementado en la práctica 1, para que acepte como parámetro un elemento de tipo `Solution`, que hace las veces de solución inicial.

```
function LOCAL_SEARCH(training, s)
    index = initialize()                                  ▷ vector de índices de 0 a n
    best_fitness = evaluate(training, w)
    while iter < MAX_ITER and neighbour < n * MAX_NEIGHBOUR_PER_TRAIT do
        comp = index[iter  % n]                           ▷ n es el número de características
        s_mut = s
        mutate(s_mut, comp, 0.3)
        s_mut.fitness = evaluate(training, s_mut.w)
        iter++
        if s_mut.fitness > best_fitness then
            neighbour = 0
            s = s_mut
            best_fitness = s_mut.fitness
            improvement = true
        else
            neighbour++
        if iter  % n == 0 or improvement then
            shuffle(index.begin(), index.end(), gen)
            improvement = false
    return w
```

Notamos que detenemos la búsqueda cuando llegamos al máximo de iteraciones, o cuando generamos un número de vecinos (dependiente del número de características) sin mejorar la función objetivo.

Algoritmos genéticos y meméticos

Aunque no se vuelven a describir en esta memoria los algoritmos genéticos y meméticos implementados en la práctica anterior, se mantienen en la tabla global de resultados para compararlos con los algoritmos de evolución diferencial, que también son algoritmos basados en poblaciones.

6. Resultados

Descripción de los casos del problema

Se consideran tres conjuntos de datos sobre los que ejecutar los algoritmos:

- **Colposcopy.** La colposcopia es un procedimiento ginecológico que consiste en la exploración del cuello uterino. Consta de 287 ejemplos, 62 atributos reales y dos clases: positivo o negativo.
- **Ionosphere.** Datos de radar recogidos por un sistema en Goose Bay, Labrador. Consta de 351 ejemplos, 34 atributos y dos clases: retornos buenos (g) y malos (b).
- **Texture.** El objetivo de este conjunto de datos es distinguir entre 11 texturas diferentes. Consta de 550 ejemplos, 40 atributos y 11 clases (tipos de textura).

Resultados obtenidos

A continuación se muestran las tablas de unos resultados obtenidos para cada uno de los algoritmos. El orden de las columnas es siempre el mismo: primero *colposcopy*, después *ionosphere*, y por último *texture*. La semilla utilizada es la semilla por defecto: 2019.

Para cada conjunto de datos se muestra una tabla con cada una de las 5 ejecuciones realizadas, de acuerdo a la técnica 5–*fold cross validation*. En cada una de ellas se muestran los valores de la tasa de clasificación (Clas), tasa de reducción (Red), función objetivo (Agr) y tiempo de ejecución (T) **en segundos**. Además, se muestra finalmente una tabla global con los resultados medios de cada conjunto de datos para todos los algoritmos. Esta información también se recoge en la última fila de las tablas de cada algoritmo.

Clasificador 1–NN sin pesos

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	69.49	0.00	34.75	0.001	88.73	0.00	44.37	0.001	93.64	0.00	46.82	0.002
2	75.44	0.00	37.72	0.001	90.00	0.00	45.00	0.001	96.36	0.00	48.18	0.002
3	70.18	0.00	35.09	0.001	91.43	0.00	45.71	0.001	92.73	0.00	46.36	0.002
4	77.19	0.00	38.60	0.001	81.43	0.00	40.71	0.001	89.09	0.00	44.55	0.002
5	82.46	0.00	41.23	0.001	85.71	0.00	42.86	0.001	97.27	0.00	48.64	0.002
$\bar{x}$	74.95	0.00	37.48	0.001	87.46	0.00	43.73	0.001	93.82	0.00	46.91	0.002

Algoritmo RELIEF

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	69.49	37.10	53.29	0.004	91.55	2.94	47.25	0.003	97.27	7.50	52.39	0.01
2	73.68	33.87	53.78	0.004	91.43	2.94	47.18	0.003	97.27	5.00	51.14	0.01
3	70.18	17.74	43.96	0.004	91.43	2.94	47.18	0.003	92.73	5.00	48.86	0.01
4	78.95	62.90	70.93	0.004	82.86	2.94	42.90	0.003	89.09	2.50	45.80	0.01
5	80.70	40.32	60.51	0.004	85.71	2.94	44.33	0.003	97.27	12.50	54.89	0.01
$\bar{x}$	74.60	38.39	56.49	0.004	88.60	2.94	45.77	0.003	94.73	6.50	50.61	0.01

Algoritmo de búsqueda local

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	77.97	82.26	80.11	19.395	84.51	82.35	83.43	3.285	88.18	77.50	82.84	21.465
2	68.42	77.42	72.92	10.405	82.86	85.29	84.08	5.334	88.18	80.00	84.09	14.294
3	75.44	85.48	80.46	15.763	80.00	91.18	85.59	9.516	92.73	82.50	87.61	25.764
4	71.93	82.26	77.09	16.191	90.00	91.18	90.59	10.026	84.55	85.00	84.77	14.206
5	77.19	75.81	76.50	16.671	85.71	82.35	84.03	4.443	93.64	87.50	90.57	24.355
$\bar{x}$	74.19	80.65	77.42	15.685	84.62	86.47	85.54	6.521	89.45	82.50	85.98	20.017

Algoritmo de enfriamiento simulado

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	71.19	80.65	75.92	49.57	91.55	88.24	89.89	41.80	85.45	85.00	85.23	118.14
2	73.68	90.32	82.00	50.17	90.00	85.29	87.65	41.29	85.45	87.50	86.48	119.92
3	70.18	82.26	76.22	51.43	90.00	91.18	90.59	42.05	84.55	85.00	84.77	119.55
4	71.93	87.10	79.51	51.76	90.00	88.24	89.12	42.26	90.00	82.50	86.25	120.45
5	73.68	82.26	77.97	51.68	84.29	88.24	86.26	41.20	81.82	87.50	84.66	120.37
$\bar{x}$	72.13	84.52	78.32	50.92	89.17	88.24	88.70	41.72	85.45	85.50	85.48	119.69

Algoritmo de búsqueda local reiterada

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	74.58	83.87	79.22	50.89	91.55	91.18	91.36	31.12	91.82	87.50	89.66	99.02
2	68.42	85.48	76.95	51.78	87.14	91.18	89.16	31.44	88.18	85.00	86.59	101.22
3	75.44	85.48	80.46	50.64 84.29	85.29	84.79	30.20	82.73	82.50	82.61	102.65	
4	73.68	85.48	79.58	51.47 84.29	85.29	84.79	31.80	83.64	87.50	85.57	99.24	
5	82.46	82.26	82.36	51.00	95.71	85.29	90.50	30.06	92.73	85.00	88.86	100.29
$\bar{x}$	74.92	84.52	79.72	51.16	88.60	87.65	88.12	30.92	87.82	85.50	86.66	100.48

Algoritmo de evolución diferencial DE/rand/1

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	67.80	90.32	79.06	49.71	94.37	88.24	91.30	41.94	92.73	87.50	90.11	120.50
2	77.19	95.16	86.18	50.65	88.57	94.12	91.34	42.24	90.91	87.50	89.20	119.65
3	68.42	93.55	80.98	50.76	87.14	91.18	89.16	42.23	89.09	87.50	88.30	121.92
4	71.93	95.16	83.55	51.03	77.14	94.12	85.63	42.28	90.91	87.50	89.20	119.79
5	80.70	93.55	87.13	50.28	87.14	91.18	89.16	42.38	94.55	87.50	91.02	120.05
$\bar{x}$	73.21	93.55	83.38	50.49	86.87	91.76	89.32	42.22	91.64	87.50	89.57	120.38

Algoritmo de evolución diferencial DE/current-to-best/1

	Colposcopy				*Ionosphere*				*Texture*			
Nº	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1	71.19	58.06	64.63	51.06	87.32	82.35	84.84	42.34	91.82	82.50	87.16	124.60
2	64.91	77.42	71.17	50.31	92.86	76.47	84.66	42.10	94.55	80.00	87.27	121.55
3	64.91	69.35	67.13	50.98	85.71	85.29	85.50	41.89	91.82	75.00	83.41	123.08
4	78.95	75.81	77.38	50.30	92.86	79.41	86.13	42.38	95.45	77.50	86.48	123.07
5	84.21	62.90	73.56	51.40	91.43	79.41	85.42	42.73	90.00	77.50	83.75	121.13
$\bar{x}$	72.83	68.71	70.77	50.81	90.04	80.59	85.31	42.29	92.73	78.50	85.61	122.69

Resumen global

Se recalca en **negrita** el mejor agregado para cada conjunto de datos, en rojo el mejor agregado entre los algoritmos genéticos y meméticos, y en azul el segundo mejor agregado excluyendo los genéticos y meméticos.

	Colposcopy				*Ionosphere*				*Texture*			
Alg	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**	**Clas**	**Red**	**Agr**	**T**
1-NN	74.95	0.00	37.48	0.001	87.46	0.00	43.73	0.001	93.82	0.00	46.91	0.002
RELIEF	74.60	38.39	56.49	0.004	88.60	2.94	45.77	0.004	94.73	6.50	50.61	0.01
BL	74.19	80.65	77.42	15.685	84.62	86.47	85.54	6.521	89.45	82.50	85.98	20.017
AGG-BLX	75.30	78.71	77.01	41.61	90.02	84.71	87.37	32.32	87.27	84.00	85.64	96.06
AGG-CA	72.87	60.00	66.43	42.77	87.74	71.76	79.75	35.18	92.55	67.50	80.02	104.69
AGE-BLX	73.49	79.35	76.42	39.53	86.02	86.47	86.25	33.11	89.64	83.50	86.57	95.28
AGE-CA	71.10	71.94	71.52	41.04	88.04	81.76	84.90	35.26	90.73	79.50	85.11	99.68
AM1	71.45	84.19	77.82	40.54	86.89	90.59	88.74	33.64	86.73	86.50	86.61	96.11
AM2	70.01	83.23	76.62	38.98	84.61	91.18	87.89	31.67	90.55	84.00	87.27	92.32
AM3	76.00	84.84	80.42	37.20	88.60	90.00	89.30	32.07	88.18	85.50	86.84	90.66
ES	72.13	84.52	78.32	50.92	89.17	88.24	88.70	41.72	85.45	85.50	85.48	119.69
ILS	74.92	84.52	79.72	51.16	88.60	87.65	88.12	30.92	87.82	85.50	86.66	100.48
DE/rand	73.21	93.55	**83.38**	50.49	86.87	91.76	**89.32**	42.22	91.64	87.50	**89.57**	120.38
DE/ctb	72.83	68.71	70.77	50.81	90.04	80.59	85.31	42.29	92.73	78.50	85.61	122.69

Análisis de resultados P1

En primer lugar, observamos que las tasas de clasificación obtenidas mediante el clasificador 1–NN sin pesos, aunque pueden llegar a ser altas según el *dataset*, no contribuyen en exceso al agregado total. Esto era de esperar, pues en este clasificador la simplicidad es siempre 0, la mínima posible.

Tomando como ejemplo el conjunto de datos *texture*, vemos que tiene una tasa de clasificación media de 92 % en este ejemplo, por lo que podemos concluir que las características medidas están bien elegidas. Sin embargo, el conjunto *colposcopy* tiene una tasa de clasificación media de solo 75 %, lo que nos hace pensar que podríamos mejorar seleccionando con más cuidado las características a medir.

En cuanto al algoritmo RELIEF, al tratarse de un algoritmo *greedy* es posible que no proporcione siempre la mejor solución. Al permitir que la tasa de reducción no sea 0 estamos aumentando la simplicidad, y por tanto potencialmente aumentando el valor de la función objetivo. Vemos que efectivamente esto es lo que ocurre, pues consistentemente en todos los conjuntos de datos mejoramos dicho valor con respecto al clasificador sin pesos. Además, el tiempo de ejecución de este algoritmo sigue siendo despreciable, por lo que constituye un buen punto de partida para

la comparación con otras metaheurísticas.

Por último, el algoritmo de búsqueda local es el más costoso en tiempo, llegando a tardar del orden de 1000 veces más por partición que los anteriores. Sin embargo, este algoritmo consigue tasas de reducción muy altas, es decir, aumenta mucho la simplicidad del clasificador. Al ser un algoritmo de búsqueda local típicamente caemos en óptimos locales, por lo que no conseguimos la solución óptima. Sin embargo, en muchas ocasiones llegamos a una solución *suficientemente buena*, como demuestran los resultados obtenidos.

En este último algoritmo conseguimos aumentar considerablemente el valor de la función objetivo en todos los casos, rondando el 80 %, a costa de un tiempo de ejecución mayor (aunque todavía viable). Aún queda ver cómo se comporta este algoritmo relativamente sencillos con otros que implementemos en el futuro.

Por último, cabe destacar que en cuanto a la tasa de clasificación, el algoritmo RELIEF es competitivo con el de la búsqueda local, llegando a ser mejor que este último en varias ocasiones. En cualquier caso, el algoritmo voraz no consigue llegar (ni se acerca) a las tasas de reducción que consigue la búsqueda local.

Análisis de resultados P3