Metaheurísticas

Aprendizaje de Pesos en Características

Práctica 2: Búsqueda basada en poblaciones

Antonio Coín Castro
77191012E
antoniocoin@correo.ugr.es
Grupo 1 (M 17:30-19:30)

5 de mayo de 2019

Índice

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 la práctica anterior, 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). Estudiámos 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 el 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 P2

Para el desarrollo de los diferentes algoritmos genéticos consideramos una estructura común que representa un cromosoma. En este contexto, un cromosoma será un vector de pesos (una posible solución) ya evaluado, es decir, junto con el valor que la función objetivo le otorga (*fitness*).

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

El valor por defecto del *fitness* de un cromosoma es -1. Esto será útil para saber que un cromosoma aún no está evaluado.

Para representar una población de soluciones (un conjunto de cromosomas) utilizamos el tipo de dato `multiset`: un contenedor asociativo y ordenado que admite repetidos. Esto último es importante, pues en un momento dado podemos concebir una población con varios cromosomas iguales. El orden que definimos entre cromosomas se puede adivinar fácilmente: si c_1 y c_2 son cromosomas, diremos que $c_1 < c_2$ si c_1.fitness $<$ c_2.fitness.

```
// Custom comparator for chromosomes
struct ChromosomeComp {
  bool operator()(const Chromosome& lhs, const Chromosome& rhs) {
    return lhs.fitness < rhs.fitness;
  }
};

// Population
typedef multiset<Chromosome, ChromosomeComp> Population;
```

Así, cuando queramos obtener el mejor cromosoma de la población miraremos el último elemento del contenedor, y cuando queramos el peor miraremos el primer elemento.

Un último detalle en cuanto a la representación de la población es que en ciertas fases del algoritmo no es necesario mantener los cromosomas ordenados. Aprovechando esto podemos introducir un nuevo tipo de dato llamado `IntermediatePopulation`, que no es más que un `vector<Chromosome>`, y beneficiarnos de la posibilidad de acceso aleatorio que ofrece este contenedor.

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] ≥ 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.

```
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
```

Operadores comunes P2

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

En primer lugar, 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 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))
```

Disponemos de una función `init_population` que se encarga de generar la población inicial, siguiendo como se pide una distribución uniforme en $[0, 1]$ para los pesos.

Pasamos ya a describir los operadores genéticos comunes a todos los algoritmos desarrollados. El operador de **selección** realiza un torneo binario entre dos padres escogidos al azar, y devuelve el

```
function INIT_POPULATION(training, n, m)
    pop = ∅
    for i := 0 to n − 1 do                          ▷ n es el número cromosomas a crear
        for j := 0 to m − 1 do                      ▷ m es el número de genes por cromosoma
            c.w[j] = random_real(0, 1)
        c.fitness = evaluate(training, c.w)
        pop.insert(p)
    return pop
```

ganador (aquel que tenga mayor valor de *fitness*). Las particularidades del tipo de dato escogido para representar una población nos obligan a manipular iteradores para seleccionar un elemento aleatorio de la población.

```
function SELECTION(population)
    p1 = advance(population.begin(), random_int(0, n − 1))    ▷ n es el tamaño de la población
    p2 = advance(population.begin(), random_int(0, n − 1))
    return p1⟶fitness < p2⟶fitness ? ∗p2 : ∗p1
```

El operador de **cruce blx** recibe dos cromosomas y devuelve dos descendientes, cuyos pesos se obtienen de forma aleatoria en un intervalo concreto que depende de los pesos de los padres. La exploración que realiza está determinada por el valor del parámetro α_{blx}, que en nuestro caso es 0,3. Notamos que eventualmente podríamos generar pesos fuera de $[0, 1]$, y sería necesario truncarlos a 0 ó a 1.

```
function BLX_CROSS(c1, c2)
    for i := 0 to n − 1 do                          ▷ n es el número de genes
        cmin = min(c1.w[i], c2.w[i])
        cmax = max(c1.w[i], c2.w[i])
        diff = cmax - cmin
        h1.w[i] = random_real(cmin - diff ∗ α_blx, cmax + diff ∗ α_blx)
        h1.w[i] = random_real(cmin - diff ∗ α_blx, cmax + diff ∗ α_blx)
        truncate(h1)
        truncate(h2)
    return (h1, h2)
```

Por su parte, el operador de **cruce aritmético** recibe dos cromosomas y devuelve un único cromosoma, cuyos pesos resultan de realizar la media aritmética de los pesos de los padres, componente a componente.

Por último, el operador de **mutación** simplemente altera una componente elegida de un cromosoma por un valor distribuido según una normal de media 0 y desviación típica $\sigma = 0{,}3$. En este

```
function ARITHMETIC_CROSS(c1, c2)
    for i := 0 to n − 1 do                          ▷ n es el número de genes
        h.w[i] = (c1.w[i] + c2.w[i]) / 2
    return h
```

caso también es posible que tengamos que truncar los pesos antes de devolver el cromosoma mutado.

```
function MUTATE(c, gene)
    c.w[gene] += normal(0, 0,3)
    c.fitness = −1                                  ▷ es necesario volver a evaluarlo
    truncate(c)
    return c
```

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.

Consideraciones comunes

Todos los algoritmos desarrollados comparten una serie de comportamientos, que detallamos a continuación con el fin de evitar repeticiones.

Tenemos un contador `iter` para llevar la cuenta del número de evaluaciones de la función objetivo que realizamos, y salimos del bucle principal de los algoritmos cuando superamos las 15,000 evaluaciones.

En general, la evolución de la población inicial se lleva a cabo mediante una población intermedia `pop_temp`, de tipo `IntermediatePopulation`, a la que podemos acceder por índices. El proceso de **selección** de un número concreto (n) de padres se realiza siempre de la misma manera:

```
SELECT(n):
    for (i := 0 to n - 1)
      pop_temp[i] = selection(pop)
```

El proceso de **cruce** de $2n$ padres sigue también un esquema similar, con pequeñas diferencias (de implementación) entre el cruce aritmético y el blx.

```
BLX(n):
    for (i := 0 to 2 * n - 1 step 2)
      pop_temp[i], pop_temp[i+1] = blx_cross(pop_temp[i], pop_temp[i+1])
```

Llamando p al tamaño de la población intermedia, en el cruce aritmético nos aseguraremos de que $4n \leq p$, para que no disminuya el tamaño de la población (este cruce nos devuelve un solo hijo por cada dos padres).

```
ARITHMETIC(n, p):
    for (i := 0 to 2 * n - 1)
      pop_temp[i] = arithmetic_cross(pop_temp[i], pop_temp[p-i-1])
```

Para el esquema de **mutación** tenemos dos alternativas. La primera es calcular el número esperado de mutaciones de genes, es decir, `num_mut` = $N \cdot M \cdot p_m$, donde N es el número de cromosomas de la población intermedia, M es el número de genes de cada cromosoma, y p_m es la probabilidad de que un gen mute. Ahora, en vez de truncar el valor obtenido, consideramos su parte entera (PE) y su parte decimal (PD), y en cada iteración del algoritmo se realizan, al

menos, tantas mutaciones como indique PE. Establecemos la posibilidad de que se realice una mutación extra, generando **en cada iteración** un número aleatorio u entre 0 y 1, y sumando 1 al número de mutaciones si $u \leq PD$.

Este es el esquema que seguiremos en los algoritmos genéticos generacionales y en los algoritmos meméticos. Además, como queremos introducir diversidad en la población, imponemos que **al menos se realice una mutación en cada generación**.

```
function NUM_MUT(n)
    expected = p_m * n                          ▷ n es el número total de genes
    if expected ≤ 1.0 then
        return 1
    remainder = modf(expected, &expected)       ▷ expected = PE, remainder = PD
    if random_real(0, 1) ≤ remainder then
        expected++
    return expected
```

También es importante tener en cuenta que al sustituir la comprobación en cada gen por realizar un número fijo de mutaciones, debemos evitar mutar dos veces el mismo gen. Usamos para esto un `set` llamado `mutated`, que no admite repetidos. Para elegir el gen a mutar, miramos la población como una matriz y generamos un entero aleatoriamente entre 0 y $N \cdot M - 1$.

```
MUTATE1(n, m):
    mutated = ∅
    num_mut = expected_mutations(n * m)
    for (i := 0 to num_mut - 1)
      while(mutated.size() == i)
        comp = random_int(0, n * m - 1)
        mutated.insert(comp)
      mutate(pop_temp[comp / m], comp % m)
```

La otra alternativa es realizar las mutaciones a nivel de cromosoma. Si p_m es la probabilidad de que mute un solo gen, definimos `pmut` = $p_m \cdot M$ como la probabilidad de que mute un cromosoma (por un solo gen). Esta alternativa es mejor cuando el tamaño de la población intermedia es pequeño, pues en esos casos el número esperado de mutaciones según la primera alternativa será casi siempre 0. La que usaremos en los AGE. Si n es el tamaño de la población intermedia:

```
MUTATE2(n, m):
    for (i := 0 to n - 1)
      if (random_real(0, 1) <= pmut)
        gene = random_int(0, m - 1)
        mutate(pop_temp[i], gene)
```

Algoritmo de búsqueda local de baja intensidad

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.

La solución inicial sobre la que iterar es la que recibimos como parámetro. 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.

```
INDEX(n):
    for (i := 0 to n - 1)
      index.push_back(i)
    shuffle(index)
```

---

function LOW_INTENSITY_LOCAL_SEARCH(training, c)
    iter = 0
    index = INDEX(n) ▷ n es el tamaño de c.w
    best_objective = c.fitness
    while iter < 2*n **do**
        comp = index[iter % n]
        c_mut = c
        c_mut.w[comp] += normal(gen)
        if c_mut.w[comp] > 1 **then**
            c_mut.w[comp] = 1
        else if c_mut.w[comp] < 0 **then**
            c_mut.w[comp] = 0
        c_mut.fitness = evaluate(training, c_mut.w)
        iter++
        if c_mut.fitness > best_objective **then**
            c = c_mut
            best_objective = c_mut.fitness
        if iter % *n* == 0 **then**
            shuffle(index)
    return c

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*).

Al tratarse de una búsqueda local de **baja intensidad**, realizamos un número fijo (pequeño) de iteraciones, proporcionales al número de características del conjunto de datos que empleamos. Notamos que hemos adaptado el código para funcionar recibiendo un dato del tipo `Chromosome`.

Algoritmos genéticos generacionales

El esquema general de los algoritmos genéticos generacionales se muestra a continuación. La única diferencia entre el AGG-BLX y el AGG-CA, aparte del uso de distintos operadores de cruce, es que en el AGG-CA selecciona el doble de padres (60) en el torneo binario, para obtener después 30 hijos tras cruzarlos (recordemos que el operador de cruce aritmético produce un hijo por cada dos padres).

Hay que tener en cuenta que como la probabilidad de cruce es $p_c = 0{,}7$, realizamos el número *esperado* de cruces en cada iteración, que en este caso es $\lfloor 15 \cdot 0{,}7 \rfloor = 10$.

```
function AGG(training, m)
    pop = init_population(training, 30, m)          ▷ m es el número de genes por cromosoma
    iter = 30;
    total_genes = 30 * m
    num_cross = pc * (30 / 2);                      ▷ número esperado de cruces
    while iter < 15000 do
        pop_temp = ∅                                                    ▷ sin ordenar
        new_pop = ∅                                                     ▷ ordenado
        best_parent = pop.last()
        SELECT(30)                                                  ▷ SELECT(60) en AGG-CA
        BLX(num_cross)                          ▷ ARITHMETIC(num_cross, 60) en AGG-CA
        MUTATE1(30, total_genes)
        for i := 0 to 30 do                                             ▷ reemplazo
            if pop_temp[i].fitness == −1 then
                pop_temp[i].fitness = evaluate(training, pop_temp[i].w)
                iter++
            new_pop.insert(pop_temp[i])
        current_best = new_pop.last()
        if current_best->fitness < best_parent->fitness then            ▷ elitismo
            new_pop.erase(new_pop.first())
            new_pop.insert(*best_parent)
        pop = new_pop                                                   ▷ nueva generación
    return pop.last()->w
```

En el algoritmo AGG-CA seleccionamos 60 padres para la población intermedia (aunque realmente solo necesitamos 40), pero tras el cruce trabajamos únicamente con los 30 primeros, respetando el esquema generacional en el que la población intermedia tiene el mismo tamaño que la población original.

Observamos que al reemplazar la nueva población nos aseguramos de que todos los cromosomas están correctamente evaluados (y ordenados según su evaluación). Antes de pasar a la siguiente generación aplicamos un comportamiento elitista: si hemos perdido la mejor solución que teníamos en la generación anterior, la introducimos en la nueva eliminando la peor solución de esta.

Algoritmos genéticos estacionarios

En este caso, cambiamos el esquema de reemplazamiento con respecto a los algoritmos generacionales. Como ahora la población intermedia es únicamente de dos hijos (que siempre cruzan), también modificamos el esquema de mutación, utilizando la segunda alternativa que comentábamos antes.

Los dos cromosomas que componen la población intermedia compiten para entrar en la nueva población con los dos peores de la población anterior, quedándonos al final con los dos mejores de entre los cuatro.

```
function AGE(training, m)
    pop = init_population(training, 30, m)          ▷ m es el número de genes por cromosoma
    iter = 30;
    while iter < 15000 do
        pop_temp = ∅                                                          ▷ sin ordenar
        new_pop = ∅                                                           ▷ ordenado
        SELECT(2)                                                  ▷ SELECT(4) en AGE-CA
        BLX(1)                                                ▷ ARITHMETIC(1, 4) en AGE-CA
        MUTATE2(30, total_genes)
        for i := 0 to 2 do                                                    ▷ reemplazo
            pop_temp[i].fitness = evaluate(training, pop_temp[i].w)
            iter++
            new_pop.insert(pop_temp[i])
        worst = pop.first()
        second_worst = pop.first() + 1
        new_best = new_pop.last()
        new_second_best = new_pop.last() - 1
        if new_second_best->fitness > second_worst->fitness then    ▷ sobreviven los dos hijos
            pop.erase(second_worst);
            pop.erase(worst);
            pop.insert(*new_second_best);
            pop.insert(*new_best);
        else if new_best->fitness > worst->fitness then          ▷ sobrevive solo el mejor hijo
            pop.erase(worst);
            pop.insert(*new_best);
        pop = new_pop                                                         ▷ nueva generación
    return pop.last()->w
```

Algoritmos meméticos

Para implementar los algoritmos meméticos nos basamos en el algoritmo AGG-BLX, que como veremos es el algoritmo generacional que mejores resultados ha proporcionado. En este caso, el tamaño de la población es de 10 cromosomas.

Simplemente vamos contando el número de generaciones mediante una variable `age`, y cada diez generaciones, es decir, si `age % 10 == 0`, aplicamos la búsqueda local de baja intensidad como se pide. Esta aplicación se realiza justo antes de pasar a la siguiente iteración del bucle principal.

En la primera versión, aplicamos la búsqueda local a **todos los cromosomas** de la población. Si *m* es el número de genes por cromosoma, recordamos que la búsqueda local de baja intensidad

realiza siempre $2m$ iteraciones.

```
AM-(10 - 1.0):
    for (c in pop)
        c = low_intensity_local_search(training, c)
        iter += 2 * m
```

En la implementación real borramos el cromosoma y lo volvemos a insertar, para así conseguir que se ordene de nuevo dentro de la población. En la segunda versión, aplicamos la búsqueda local a cada cromosoma con una probabilidad de 0,1, lo que se traduce en $10 \cdot 0{,}1 = 1$ aplicación esperada.

```
AM-(10 - 0.1):
    c = pop.first() + random_int(0, 9)
    *c = low_intensity_local_search(training, *c)
    iter += 2 * m
```

La tercera versión es muy similar a la segunda, solo que en vez de aplicar la búsqueda local a un cromosoma aleatorio de la población, lo hacemos al mejor de ellos.

```
AM-(10 - 0.1 mej):
    c = pop.last()
    *c = low_intensity_local_search(training, *c)
    iter += 2 * m
```

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_p2`, 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 p2`. 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:

- `p2.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 `p2`.

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

5. 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

Resumen global

	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

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 P2

cambiando parámetros, hay más libertad, etc