UNIVERSIDAD
DE GRANADA

METAHEURÍSTICAS

GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

TÉCNICAS DE BÚSQUEDA BASADAS EN POBLACIONES PARA EL PROBLEMA DEL APRENDIZAJE DE PESOS EN CARACTERÍSTICAS

Autor
Vladislav Nikolov Vasilev

NIE
X8743846M

E-Mail
vladis890@gmail.com

Grupo de prácticas
MH3 Jueves 17:30-19:30

Rama
Computación y Sistemas Inteligentes

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE TELECOMUNICACIÓN

CURSO 2018-2019

Índice

1. Descripción del problema

El problema que se aborda en esta práctica es el Aprendizaje de Pesos en Características (APC). Es un problema típico de *machine learning* en el cuál se pretende optimizar el rendimiento de un clasificador basado en vecinos más cercanos. Esto se consigue mediante la ponderación de las características de entrada con un vector de pesos W, el cuál utiliza codificación real (cada $w_i \in W$ es un número real), con el objetivo de modificar sus valores a la hora de calcular la distancia. Cada vector W se expresa como $W = \{w_1, w_2, \ldots, w_n\}$, siendo n el número de dimensiones del vector de características, y cumpliéndose además que $\forall w_i \in W,\ w_i \in [0, 1]$.

El clasificador considerado para este problema es el 1-NN (genéricamente, un clasificador k-NN, con k vecinos, siendo en este caso $k = 1$), es decir, aquél que clasifica un elemento según su primer vecino más cercano utilizando alguna medida de distancia (en este caso, utilizando la distancia Euclídea). Cabe destacar que no en todos los casos se usará el clasificador 1-NN ya que se pueden dar casos en los que el vecino más cercano de un elemento sea él mismo. Por ese motivo, en algunas técnicas/algoritmos se usará un 1-NN con el criterio de *leave-one-out*, es decir, que se busca el vecino más cercano pero excluyéndose a él mismo.

El objetivo propuesto es aprender el vector de pesos W mediante una serie de algoritmos, de tal forma que al optimizar el clasificador se mejore tanto la precisión de éste como su complejidad, es decir, que se considere un menor número de características. Estos dos parámetros, a los que llamaremos $tasa\_clas$ y $tasa\_red$, respectivamente, se pueden expresar de la siguiente forma:

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

$$tasa\_red = 100 \cdot \frac{n^o\ valores\ w_i < 0.2}{n^o\ caracteristicas}$$

siendo T el tamaño del conjunto de datos sobre que el que se evalúa el clasificador.

Por tanto, al combinarlos en una única función a la que llamaremos $F(W)$, la cuál será nuestra función objetivo a optimizar (maximizar), tenemos que:

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

siendo α la importancia que se le asigna a la tasa de clasificación y a la de reducción, cumpliendo que $\alpha \in [0, 1]$. En este caso, se utiliza un $\alpha = 0.5$ para dar la misma importancia a ambos, con lo cuál se pretende que se reduzcan al máximo el número de características conservando una $tasa\_clas$ alta.

2. Descripción de los algoritmos

2.1. Consideraciones previas

Antes de empezar con la descripción formal de los algoritmos implementados, vamos a describir algunos aspectos comunes, como por ejemplo cómo se representan e inicializan las soluciones; cómo se representan la población, los padres y las elecciones para mutar los cromosomas; cómo se realiza el torneo binario, la mutación y los distintos cruces, y algunas funciones utilizadas en muchas partes del código, como por ejemplo la función objetivo o la forma de evaluar a la población. Cabe destacar que muchos de los pseudocódigos que aparecen a continuación no se han implementado exactamente igual o no aparecen en el código, ya que o bien son operaciones que se han vectorizado o bien ya hay funciones que hacen eso.

Primero vamos a ver como se representa la población, los padres y una elección de mutación. Una población no es más que un vector de vectores, o lo que es lo mismo, una matriz de tamaño $N \times M$, donde N es el número de cromosomas (soluciones) y M es el número de genes (número de características). Por tanto, ahora tenemos un conjunto de vectores de pesos, lo que viene a significar que tenemos múltiples W. Esta población está ordenada en todo momento por el valor *fitness* de cada cromosoma para facilitar las operaciones posteriores. Un padre p no es más que un índice de un cromosoma de la población, con la restricción de que $p \in [0, N)$. Una mutación viene dada por dos valores c, g, donde c es el cromosoma a mutar y g el gen a mutar. Estos dos valores están sujetos a que $c \in [0, N)$ y a que $g \in [0, M)$.

Como cada fila de la matriz es un vector de pesos W, se tiene que cumplir que $\forall w_i \in W,\ w_i \in [0, 1]$. Por tanto, para evitar que las soluciones se salgan de este intervalo, se ha implementado una función que se encarga de normalizar los valores de W en el rango. La función se ha usado tanto en los algoritmos meméticos como en los genéticos a la hora de realizar un cruce o una mutación, para que los valores de los cromosomas siguiesen siendo válidos. La implementación de esta función es la siguiente:

Algorithm 1 Función que normaliza un vector de pesos W

```
1: function NormalizarW(W)
2:     for each w_i ∈ W do
3:         if w_i < 0 then
4:             w_i ← 0
5:         else if w_1 > 1 then
6:             w_i ← 1
7:     return W
```

Para generar las soluciones iniciales se ha utilizado una función que recibe como parámetros el número de genes y el número de cromosomas, y crea una nueva matriz que representa la población, inicializándola con valores aleatorios generados mediante una distribución uniforme en el rango $[0, 1]$. Su pseudocódigo se puede ver a continuación:

Algorithm 2 Función que genera una población inicial

```
1: function GenerarPoblacionInicial(numCrom, numGenes)
2:     poblacion ← NuevaMatrizVacia(numCrom, numGenes)
3:     for i ← 0 to numCrom − 1 do
4:         for j ← 0 to numGenes − 1 do
5:             poblacion[i][j] ← ValorAleatorioUniformeRango0-1()
6:     return poblacion
```

Para seleccionar los padres o cromosomas que pasarán su material genético se ha utilizado una función que recibe los índices de 2 cromosomas y la lista de valores *fitness*, y devuelve el índice del cromosoma con mejor valor *fitness*. Se muestra su implementación a continuación:

Algorithm 3 Función que realiza un torneo binario y elige el mejor padre

```
1: function TorneoBinario(listaFitness, indxCrom1, indxCrom2)
2:     fitCrom1 ← listaFitness[indxCrom1]
3:     fitCrom2 ← listaFitness[indxCrom2]
4:     mejorPadre ← indxCrom1
5:     if fitCrom1 < fitCrom2 then
6:         mejorPadre ← indxCrom2
7:     return mejorPadre
```

En cuanto a los cruces, para el cruce BLX-α se ha creado una función que recibe la población, los índices de los padres y aplica el cruce, generando dos descendientes. En este caso se ha especificado que $\alpha = 0.3$. Aquí se puede ver como funciona:

Algorithm 4 Cruce BLX-α con $\alpha = 0.3$ (I)

```
1: function CruceBLXAlfa(poblacion, indPadre1, indPadre2, numGenes, α)
2:     padre1, padre2 ← poblacion[indPadre1], poblacion[indPadre2]
3:     hijo1, hijo2, C_min, C_max, I ← NuevoVectorVacio(numGenes)
4:     for i ← 0 to numGenes − 1 do
5:         C_min[i] ← Minimo(padre1[i], padre2[i])
6:         C_max[i] ← Maximo(padre1[i], padre2[i])
7:         I[i] ← C_max[i] − C_min[i])
```

Algorithm 5 Cruce BLX-α con $\alpha = 0.3$ (II)

```
8:    for i ← 0 to numGenes − 1 do
9:        hijo1[i] ← ValorAleatorioUnifInter(C_min[i] − I[i] · α, C_max[i] + I[i] · α)
10:       hijo2[i] ← ValorAleatorioUnifInter(C_min[i] − I[i] · α, C_max[i] + I[i] · α)
11:   NormalizarW(hijo1), NormalizarW(hijo2)
12:   return hijo1, hijo2
```

Para el cruce aritmético (AC) se ha implementado una función que recibe la población y los índices de los padres, y genera los descendientes haciendo la media aritmética, como se puede ver aquí:

Algorithm 6 Función del cruce aritmético

```
1: function CruceAritmetico(poblacion, indPad1, indPad2, numGenes)
2:     padre1, padre2 ← poblacion[indPad1], poblacion[indPad2]
3:     hijo ← NuevoVectorVacio(numGenes)
4:     for i ← 0 to numGenes − 1 do
5:         hijo[i] ← (padre1[i] + padre2[i]) / 2
6:     return hijo
```

Para la mutación se ha creado una función que recibe la población y los índices del cromosoma y gen a mutar, y añade a dicho gen un valor generado por una distribución normal con $\mu = 0$ y $\sigma = 0.3$. Se puede ver a continuación:

Algorithm 7 Función de mutación

```
1: procedure Mutacion(pob, indCrom, indGen)
2:     pob[indCrom][indGen] ← pob[indCrom][indGen]+ ValorDistribNorm(μ, σ)
3:     NormalizarW(pob[indCrom])
```

Vamos a comentar ahora algunos detalles extra. Es importante saber como se calcula la distancia a un vecino, ya que esto juega un factor muy importante a la hora de encontrar cuál es el vecino más cercano a un elemento (o el vecino más cercano por el criterio *leave-one-out*). En la implementación de la práctica se ha utilizado un KDTree, que es una estructura de datos parecida a un árbol binario, solo que de K dimensiones. Por dentro, esta estructura utiliza la distancia Euclídea (distancia en línea recta entre dos elementos) para determinar cuál es el elemento más próximo a otro. No hace falta conocer como se implementa esta estructura de datos, pero sí es importante conocer cómo se realiza el cálculo de la distancia Euclídea. En el siguiente pseudocódigo se puede ver el cálculo:

Algorithm 8 Cálculo de la distancia Euclídea entre dos puntos

function DISTANCIAEUCLIDEA(e_1, e_2)
  $distancia \leftarrow \sqrt{\sum_{i=1}^{N}(e_1^i - e_2^i)^2}$
  return *distancia*

También es importante ver cómo se mantiene la población ordenada. Para hacer esto, se ha creado una función que recibe la lista de valores *fitness* y la poblacion, obtiene los índices que dan el orden de forma ascendente de la lista y con estos índices ordena la población y la lista de *fitness*. Aquí se puede ver como funciona:

Algorithm 9 Función para ordenar la población según su valor *fitness*

```
1: function OrdenarPoblacion(fitness , poblacion)
2:     indicesOrden ← ObtenerIndicesOrdenados(fitness)
3:     fitnessOrdenado ← NuevoVectorVacioMismaCapacidad(fitness)
4:     poblacionOrdenada ← NuevaMatrizVaciaMismaCapacidad(poblacion)
5:     for each indice ∈ indicesOrden do
6:         fitnessOrdenado ← fitness[indice]
7:         poblacionOrdenada ← poblacion[indice]
8:     return fitnessOrdenado, poblacionOrdenada
```

Pasemos a ver ahora la función objetivo, $F(W)$, que es lo que se pretende optimizar. Para evaluar la función objetivo, necesitamos calcular $tasa\_clas$ y $tasa\_red$. Para calcular lo primero, podemos seguir la idea detrás del siguiente pseudocódigo:

Algorithm 10 Cálculo de la tasa de clasificación

```
1: function CalculoTasaClas(etiq, etiqPred, N)
2:     bienClasificados ← 0
3:     for i ← 1 to N do
4:         if etiq_i = etiqPred_i then
5:             bienClasificados ← bienCasificados + 1
6:     tasa_clas ← bienClasificados / N
7:     return tasa_clas
```

Para calcular $tasa\_red$, suponiendo que queremos saber el número de características por debajo de 0.2 podemos seguir un esquema como el siguiente:

Algorithm 11 Cálculo de la tasa de reducción (I)

```
1: function CalculoTasaRed(W, N)
2:     caracRed ← 0
3:     for each w_i ∈ W do
```

Algorithm 12 Cálculo de la tasa de reducción (II)

```
4:        if w_i < 0.2 then
5:            caracRed ← caracRed + 1
6:    tasa_red ← caracRed / N
7:    return tasa_red
```

Y finalmente, para poder calcular la función a optimizar (nuestra función *fitness* u objetivo), teniendo en cuenta que usamos un $\alpha = 0.5$ para ponderar las dos tasas, y que anteriormente hemos calculado ambas tasas, podemos seguir el siguiente esquema:

Algorithm 13 Cálculo de la función objetivo o *fitness*

```
1: function CalculoFuncionFitness(tasa_clas, tasa_red, α)
2:     fitness ← α · tasa_clas + (1 − α) · tasa_red
3:     return fitness
```

Para acabar, y antes de pasar a ver la implementación de los algoritmos, veamos otra funcionalidad que se usa en todos los algoritmos, que es la forma en la que se evalúa la función objetivo. Para eso, se usa una función que permite evaluar a toda la nueva población, la cuál a su vez llama a una función que evalúa cada elemento de la población. Veamos su funcionamiento, empezando por la función más específica (la cuál solo evalúa un vector de pesos W) hasta la más genérica:

Algorithm 14 Función para evaluar un vector de pesos W

```
1: function Evaluar(datos, etiquetas, W)
2:     datosPesos ← aplicar w_i ∈ W sobre los x_i ∈ datos donde w_i > 0.2
3:     arbolKD ← KDTree(datosPesos)
4:     vecinos ← arbolKD.ObtenerVecinosMasCercanoL1O(datosPesos)
5:     pred ← etiquetas[vecinos]
6:     tasa_clas ← CalcularTasaClas(etiquetas, pred, num. etiquetas)
7:     tasa_red ← CalcularTasaRed(W, num. caracteristicas)
8:     fitness ← CalculoFuncionFitness(tasa_clas, tasa_red)
9:     return fitness
```

Algorithm 15 Función para evaluar una población

```
1: function EvaluarPoblacion(datos, etiquetas, poblacion)
2:     listaFitness ← NuevoVector()
3:     for each W ∈ poblacion do
4:         listaFitness.Añadir(Evaluar(datos, etiquetas, W))
5:     return listaFitness
```

2.2. Algoritmos de comparación

2.2.1. Algoritmo greedy *RELIEF*

El algoritmo con el que vamos a comparar es el algoritmo para el cálculo de pesos *RELIEF*. Es un algoritmo greedy que, comenzando con un W cuyos pesos valen 0, actualiza W para cada $x_i \in X$, buscando para cada x_i cuál es su aliado más cercano (elemento que tiene la misma etiqueta que x_i con el criterio de *leave-one-out*, ya que él mismo podría ser su vecino más cercano) y su enemigo más cercano (elemento que tiene diferente etiqueta a la que tiene x_i).

A la hora de implementarlo, vamos a utilizar 2 KDTree en cada iteración que se van a construir sobre la marcha. En uno se encontrarán todos los aliados de e y en el otro estarán todos sus enemigos. Esto puede suponer una gran penalización por el tiempo de creación de los árvoles, pero es un tiempo insignificante ya que el algoritmo es muy rápido. Después de construir los árboles, se buscará en el caso del aliado más cercano, por el criterio de *leave-one-out*, cuál es el índice de este aliado. En el caso del enemigo más cercano, como este no puede ser él mismo, se buscará el índice del vecino más cercano en ese árbol. Una vez hecho eso, obtendremos los respectivos aliado y enemigo del conjunto de aliados y enemigos. Una vez teniéndolos, ya se puede actualizar el valor de W.

Cuando se ha terminado de iterar sobre todos los elementos de X, se normaliza W para que esté en el rango $[0, 1]$ eligiendo el $w_i \in W$ que sea más grande. Todos aquellos valores por debajo de 0 se truncan a 0, y el resto se normaliza dividiéndolos entre w_m (el w_i más grande).

Antes de ver su implementación, veamos como se inicializa una solución:

Algorithm 16 Inicialización de un vector de pesos W en *RELIEF*

```
1: function GenerarWRelief(N)
2:     W ← VectorVacioCapacidad(N)
3:     for each w_i ∈ W do
4:         w_i ← 0
5:     return W
```

Una vez dicho esto, veamos cómo sería la implementación:

Algorithm 17 Cálculo de los pesos mediante *RELIEF* (I)

```
1: function RELIEF(X, Y)
2:     N ← ObtenerNumElementos(Y)
3:     numCarac ← ObtenerNumCarac(X)
```

Algorithm 18 Cálculo de los pesos mediante *RELIEF* (II)

```
4:    W ← GenerarWRelief(numCarac)
5:    for i ← 0 to N − 1 do
6:        x, y ← X[i], Y[i]
7:        aliados ← e_a ⊂ X : etiqueta(e_a) = y
8:        enemigos ← e_e ⊂ X : etiqueta(e_e) ≠ y
9:        arbolAliados ← KDTree(aliados)
10:       arbolEnemigos ← KDTree(enemigos)
11:       aliadoCercano ← arbolAliados.ObtenerVecinoMasCercanoL1O(x)
12:       enemigoCercano ← arbolEnemigos.ObtenerVecinoMasCercano(x)
13:       aliado ← aliados[aliadoCercano]
14:       enemigo ← enemigos[enemigoCercano]
15:       W ← W + |x − enemigo| − |x − aliado|
16:   w_m ← max(W)
17:   for each w_i ∈ W do
18:       if w_i < 0 then
19:           w_i ← 0
20:       else
21:           w_i ← w_i / w_m
22:   return W
```

2.3. Algoritmos de búsqueda basados en poblaciones y trayectorias

2.3.1. Búsqueda Local

En la práctica anterior se implementó una búsqueda local (búsqueda basada en trayectorias), así que vamos a rescatarla para tener una metaheurística de la que poder partir (además de que los AM la necesitarán). Esta búsqueda, como debemos recordar, está basada en el primer mejor (Simple Hill Climbing).

La búsqueda local parte de un vector W con valores aleatorios y busca optimizarlo mediante la exploración de los vecinos. Esta exploración se realiza modificando con un valor aleatorio generado a partir de una distribución normal con $\mu = 0$ y $\sigma = 0.3$ un $w_i \in W$, quedándose con el cambio en caso de mejorar la función objetivo, o descartándolo en otro caso.

Para realizar lo explicado anteriormente, se genera una permutación del conjunto $\{0, 1, \ldots, N-1\}$, donde N es el número de características, y se van escogiendo las características según el orden de la permutación, aplicándoles el cambio anteriormente descrito. Si no se produce una mejora, se descarta el cambio realizado. Si no se ha producido mejora en la función objetivo con la permutación, se escoge una nueva permutación y se repite el proceso, hasta un máximo de $20 \cdot N$ evaluaciones sin éxito seguidas de la función objetivo. Si se produce mejora, se acepta el cambio y se genera una nueva permutación, repitiendo el proceso. Todo esto se realiza hasta que se hayan realizado 15000 evaluaciones de la función objetivo, o hasta que se dé la condición anterior (demasiadas iteraciones sin mejora).

Para inicializar una solución de la forma mencionada anteriormente podemos utilizar la siguiente función:

Algorithm 19 Inicialización de un vector de pesos W en BL

```
1: function GenerarWBL(N)
2:     W ← vector[N]
3:     for each w_i ∈ W do
4:         w_i ← ValorAleatorioUniformeRango0-1()
5:     return W
```

El pseudocódigo de la búsqueda local se puede ver aquí:

Algorithm 20 Cálculo de los pesos mediante la Búsqueda Local (I)

```
1: function BusquedaLocal(datos, etiquetas)
2:     N ← ObtenerNumCaracteristicas(datos)
3:     W ← GenerarWBL(N)
```

Algorithm 21 Cálculo de los pesos mediante la Búsqueda Local (II)

```
4:  evaluaciones ← 0
5:  evaluacionesMalas ← 0
6:  fitness ← Evaluar(datos, etiquetas, W)
7:  while evaluaciones < 15000 do
8:      Wactual ← W
9:      ordenCaracteristicas ← Permutacion(0 to N − 1)
10:     for each carac ∈ ordenCaracteristicas do
11:         W[carac] ← W[carac]+ GenerarValorDistribucionNormal(μ, σ)
12:         W ← NormalizarW(W)
13:         evaluaciones ← evaluaciones + 1
14:         nuevoFitness ← Evaluar(X, Y, W)
15:         if nuevoFitness > fitness then
16:             fitness ← nuevoFitness
17:             evaluacionesMalas ← 0
18:             break
19:         else
20:             evaluacionesMalas ← evaluacionesMalas + 1
21:             W[carac] ← Wactual[carac]
22:         if evaluaciones > 15000 or evaluacionesMalas > 20 · N then
23:             return W
24:  return W
```

2.3.2. Algoritmos genéticos generacionales

La primera metaheurística implementada en esta práctica son los AGG (algoritmos genéticos generacionales). Esta versión parte de una población inicial de 30 cromosomas generada aleatoriamente y pretende optimizarla con el esquema generacional. Este esquema consiste en generar una nueva población, evaluarla y sustituir a la anterior. Sin embargo, con el objetivo de no perder la mejor solución hasta el momento, se sigue un esquema de **elitismo**, el cuál es selectivo. Es decir, si el mejor cromosoma de la población anterior es mejor que el mejor cromosoma de la nueva población, se reintroduce el mejor de la población anterior sustituyendo al peor de la nueva población. Esto se puede ver a continuación:

Algorithm 22 Esquema de elitismo selectivo

```
1: procedure ELITISMO(poblacionAnt, fitAnt, poblacionNueva, fitNuevo)
2:     fitMejorPobAnt ← fitAnt[0]
3:     fitMejorPobNueva ← fitNuevo[0]
4:     if fitMejorPobAnt > fitMejorPobNueva then
5:         poblacionNueva.InsertarPrimero(poblacionAnt[0])
6:         poblacionNueva.EliminarUlitmo()
7:         fitNuevo.InsertarPrimero(fitMejorPobAnt)
8:         fitNuevo.EliminarUltimo()
```

Antes de mostrar la implementación del AGG, es importante destacar algunos aspectos. La probabilidad de cruce es de 0.7, mientras que la de mutación es de 0.001. Para evitar generar muchos números aleatorios tanto a la hora de cruzar como a la hora de seleccionar los padres, se siguen algunas estrategias de optimización. Por ejemplo, se calcula el número de parejas que se van a cruzar a priori de la forma que $numParejasCruce = (NumCromosomas \ / \ 2) \cdot probCruce$ y se trunca este valor. Por lo tanto, según el número de cromosomas que se escogen en el torneo binario (el número depende del cruce a utilizar), los $2numParejasCruce$ primeros cromosomas se van a cruzar, de la forma que el priemero se cruza con el segundo, el tercero con el cuarto y así para el resto. Los que no se crucen pasarán su material genético copiándolo. En cuanto a las mutaciones, se calcula antes el número de mutaciones a realizar con la forma $numMutaciones = numCromosomas \cdot numGenes \cdot probMutacion$. Como este valor puede ser menor que 1, se tiene un contador que acumula $numMutaciones$ hasta que sea mayor o igual a 1, y en cuanto se dé eso, se trunca el número de mutaciones y se seleccionan qué cromosomas y genes mutar.

En cuanto a las contidiciones de parada, se tienen que realizar **15000 evaluaciones** de la función objetivo. Con esto dicho, veamos la implementación del AGG:

Algorithm 23 Algoritmo Genético Generacional con varios cruces posibles

```
1: function AGG(datos, etiquetas, opCruce)
2:     numGenes ← ObtenerNumCaracteristicas(datos)
3:     Determinar numParejasCruce y numPadres en función de opCruce
4:     numMutaciones, contMutaciones ← CalcularNumMutaciones()
5:     numEvaluaciones ← 0
6:     poblacion ← GenerarPoblacionInicial(numCrom, numGenes)
7:     fitness ← EvaluarPoblacion(datos, etiquetas, poblacion)
8:     fitness, poblacion ← OrdenarPoblacion(fitness, poblacion)
9:     numEvaluaciones ← numEvaluaciones + numCrom
10:    while numEvaluaciones < 15000 do
11:        nuevosCrom ← VectorBooleanoFalsoCapacidad(numCromosomas)
12:        listaPadres ← NuevaLista()
13:        for i ← 0 to numPadres − 1 do
14:            indPad1, indPad2 ← Generar2IndicesAleatorios()
15:            listaPadres.Insertar(TorneoBinario(fitness, indPad1, indPad2))
16:        parejas ← GenerarParejasCruce(listaPadres, numParejasCruce)
17:        nuevaPob ← NuevaMatrizVacia()
18:        nuevoFit ← NuevoVectorVacio()
19:        for each pareja ∈ parejas do
20:            hijos ← opCruce(poblacion, pareja.pad1, pareja.pad2, numGenes)
21:            nuevaPob.InsertarFila(hijos)
22:        Copiar los cromosomas de los padres que no cruzan en nuevaPoblacion
23:        Actualizar nuevosCrom donde se hayan generado descendientes
24:        if contMutaciones ≥ 1 then
25:            Truncar contMutaciones
26:            Generar las parejas de índices (gen, cromosoma) a mutar
27:            Mutacion(nuevaPob, gen, cromosoma) para cada pareja
28:            Actualizar nuevosCrom donde se haya mutado
29:        else
30:            contMutaciones ← contMutaciones + numMutaciones
31:        for i ← 0 to numCrom − 1 do
32:            if nuevoCrom[i] then
33:                nuevoFit.insertar(Evaluar(datos, etiquetas, nuevaPob[i]))
34:            else
35:                nuevoFit.insertar(ftiness[listaPadres[i]])
36:        numEvaluaciones ← numEvaluaciones + nuevosCrom.Verdaderos()
37:        nuevoFit, nuevaPob ← OrdenarPoblacion(nuevoFit, nuevaPob)
38:        Elitismo(poblacion, fitness, nuevaPob, nuevoFit)
39:        poblacion, fitness ← nuevaPob, nuevoFit
40:    W ← poblacion[0]
41:    return W
```

2.3.3. Algoritmos genéticos estacionarios

La segunda metaheurística implementada en esta práctica son los AGE (algoritmos genéticos estacionarios). Esta versión parte de una población inicial de 30 cromosomas generada aleatoriamente y pretende optimizarla con el esquema estacionario. Este esquema consiste en generar 2 nuevos cromosomas y hacer que combatan contra los últimos 2 cromosomas de la población para ver quién se queda dentro (se comparan sus valores *fitness* y se eligen los 2 mejores). A diferencia del anterior esquema, no se sustituye toda la población, si no que se determina si los nuevos cromosomas son lo suficientemente buenos como para entrar en la población. Como no hay mucha posibilidad de perder la mejor solución, no se utiliza elitismo.

En esta versión, la probabilidad de mutación sigue siendo la misma (0.001), pero ahora la probabilidad de cruce es 1, para que se generen siempre esos 2 descendientes. En cuanto al esquema de generar padres, es casi igual que en el caso anterior, solo que se tienen que hacer los torneos binarios suficientes para generar 2 hijos (dependerá del cruce, ya que para el BLX-*alpha* solo se necesitan 2 torneos binarios, mientras que para el cruce aritmético se necesitarán 4 ya que cada pareja produce solo un descendiente). El esquema de mutaciones, por otra parte, se mantiene casi intacto, ya que ahora solo van a poder mutar los nuevos cromosomas generados, que serán solo 2. El resto de condiciones se mantiene igual.

Una vez dicho esto, veamos el pseudocódigo del AGE:

Algorithm 24 Algoritmo Genético Estacionario con varios cruces posibles (I)

```
1:  function AGG(datos, etiquetas, opCruce)
2:      numGenes ← ObtenerNumCaracteristicas(datos)
3:      numHijos ← 2                          ▷ Se tienen que generar 2 hijos
4:      Determinar numPadres en función de opCruce
5:      numMutaciones, contMutaciones ← CalcularNumMutaciones()
6:      numEvaluaciones ← 0
7:      poblacion ← GenerarPoblacionInicial(numCrom, numGenes)
8:      fitness ← EvaluarPoblacion(datos, etiquetas, poblacion)
9:      fitness, poblacion ← OrdenarPoblacion(fitness, poblacion)
10:     numEvaluaciones ← numEvaluaciones + numCrom
11:     while numEvaluaciones < 15000 do
12:         listaPadres ← NuevaLista()
13:         for i ← 0 to numPadres − 1 do
14:             indPad1, indPad2 ← Generar2IndicesAleatorios()
15:             listaPadres.Insertar(TorneoBinario(fitness, indPad1, indPad2))
16:         parejas ← GenerarParejasCruce(listaPadres, numParejasCruce)
17:         descendientes ← NuevaMatrizVacia()
18:         for each pareja ∈ parejas do
```

Algorithm 25 Algoritmo Genético Estacionario con varios cruces posibles (II)

```
19:             hijos ← opCruce(poblacion, pareja.pad1, pareja.pad2, numGenes)
20:             descendientes.InsertarFila(hijos)
21:         if contMutaciones ≥ 1 then
22:             Truncar contMutaciones
23:             Generar las parejas de índices (gen, cromosoma) a mutar
24:             Mutacion(descendientes, gen, cromosoma) para cada pareja
25:             Actualizar nuevosCrom donde se haya mutado
26:         else
27:             contMutaciones ← contMutaciones + numMutaciones
28:         descendientesFit ← EvaluarPoblacion(datos, etiquetas, descendientes)
29:         descendientesFit, descendientes ← OrdenarPoblacion(descendientesFit,
    descendientes)
30:         numEvaluaciones ← numEvaluaciones + numHijos
31:         pobTorneo ← Concat(poblacion.Obtener2UltCrom(), descendientes)
32:         fitTorneo ← Concat(fitness.Obtener2UltFit(), descendientesFit)
33:         fitTorneo, pobTorneo ← OrdenarPoblacion(fitTorneo, nuevaPob)
34:         Eliminar 2 últimos cromosomas de poblacion y valores de fitness
35:         poblacion.InsertarFinal(pobTorneo.Obtener2Primeros())
36:         fitness.InsertarFinal(fitTorneo.Obtener2Primeros())
37:         fitness, poblacion ← OrdenarPoblacion(fitness, poblacion)
38:     W ← poblacion[0]
39:     return W
```

2.3.4. Algoritmos meméticos

La tercera y última metaheurística implementada en esta práctica son los AM (algoritmos meméticos). Los algoritmos meméticos son una hibridación entre los algoritmos genéticos y la búsqueda local con el objetivo de conseguir unos mejores resultados. En la implementación realizada se ha utilizado una población de **10 cromosomas**, con una probabilidad de cruce de 0.7 y una probabilidad de mutación de 0.001. La búsqueda local se aplica cada **10 generaciones**, contando también la población inicial. El AGG sobre el que se ha decidido implementar es el AGG con cruce BLX-α, ya que es el que ofrecía mejores resultados.

Respecto a la búsqueda local que utiliza el AM, esta se parece mucho a la que se ve en la sección 2.3.1, solo que con las siguientes modificaciones:

- La búsqueda local recibe como parámetros también el valor inicial de W con su valor *fitness*, con lo cual no tiene que calcularlos (se eliminarían esas líneas).
- Realiza siempre 80 evaluaciones de la búsqueda local, con lo cuál esa es su condición de parada. No comprueba si han pasado $20 \cdot N$ iteraciones sin que se haya producido mejora.
- Devuelve, a parte del W, su valor *fitness*, para no tener que calcularlo de nuevo.

Respecto a los AM, se han implementado 3 versiones, las cuáles aplican la búsqueda local con diferentes criterios: una versión que aplica la búsqueda local sobre **toda la población**, una versión que aplica la búsqueda local sobre los $0.1 \cdot numCromosomas$ **mejores cromosomas**, y una versión que aplica la búsqueda local sobre un número de **cromosomas aleatorios** dado por $0.1 \cdot numCromosomas$.

El resto de criterios, tanto selección de padres como mutaciones se hace igual que en los AGG. Por tanto, sabiendo todo esto, veamos como sería una posible implementación del AM, para cualquiera de los 3 criterios:

Algorithm 26 Algoritmo Memético con cruce BLX-α y múltiples criterios BL (I)

```
1: function AGG(datos, etiquetas, criterioBL)
2:     numGenes ← ObtenerNumCaracteristicas(datos)
3:     numParejasCruce ← Truncar((numCrom / 2) · probCruce)
4:     numMutaciones, contMutaciones ← CalcularNumMutaciones()
5:     numEvaluaciones ← 0
6:     generacion ← 0
7:     poblacion ← GenerarPoblacionInicial(numCrom, numGenes)
8:     fitness ← EvaluarPoblacion(datos, etiquetas, poblacion)
```

Algorithm 27 Algoritmo Memético con cruce BLX-α y múltiples criterios BL (II)

```
9:    fitness, poblacion ← OrdenarPoblacion(fitness, poblacion)
10:   numEvaluaciones ← numEvaluaciones + numCrom
11:   generacion ← generacion + 1
12:   while numEvaluaciones < 15000 do
13:       nuevosCrom ← VectorBooleanoFalsoCapacidad(numCromosomas)
14:       listaPadres ← NuevaLista()
15:       for i ← 0 to numCrom − 1 do
16:           indPad1, indPad2 ← Generar2IndicesAleatorios()
17:           listaPadres.Insertar(TorneoBinario(fitness, indPad1, indPad2))
18:       parejas ← GenerarParejasCruce(listaPadres, numParejasCruce)
19:       nuevaPob ← NuevaMatrizVacia()
20:       nuevoFit ← NuevoVectorVacio()
21:       for each pareja ∈ parejas do
22:           hijos ← opCruce(poblacion, pareja.pad1, pareja.pad2, numGenes)
23:           nuevaPob.InsertarFila(hijos)
24:       Copiar los cromosomas de los padres que no cruzan en nuevaPoblacion
25:       Actualizar nuevosCrom donde se hayan generado descendientes
26:       if contMutaciones ≥ 1 then
27:           Truncar contMutaciones
28:           Generar las parejas de índices (gen, cromosoma) a mutar
29:           Mutacion(nuevaPob, gen, cromosoma) para cada pareja
30:           Actualizar nuevosCrom donde se haya mutado
31:       else
32:           contMutaciones ← contMutaciones + numMutaciones
33:       for i ← 0 to numCrom − 1 do
34:           if nuevoCrom[i] then
35:               nuevoFit.insertar(Evaluar(datos, etiquetas, nuevaPob[i]))
36:           else
37:               nuevoFit.insertar(ftiness[listaPadres[i]])
38:       numEvaluaciones ← numEvaluaciones + nuevosCrom.Verdaderos()
39:       generacion ← generacion + 1
40:       nuevoFit, nuevaPob ← OrdenarPoblacion(nuevoFit, nuevaPob)
41:       Elitismo(poblacion, fitness, nuevaPob, nuevoFit)
42:       poblacion, fitness ← nuevaPob, nuevoFit
43:       if generacion mod 10 = 0 then
44:           Escoger W ∈ poblacion sobre los que aplicar BL según criterioBL
45:           Aplicar BusquedaLocal(datos, etiquetas, W, fitness_W)
46:           Actualizar numEvaluaciones
47:           fitness, poblacion ← OrdenarPoblacion(fitness, poblacion)
48:   W ← poblacion[0]
49:   return W
```

3. Desarrollo de la práctica

La práctica se ha implementado en **Python3** y ha sido probada en la versión 3.7.1. Por tanto, se recomienda encarecidamente utilizar un intérprete de Python3 al ejecutar el código y no uno de la versión 2.X, debio a problemas de compatibilidad con ciertas funciones del lenguaje. Se ha probado el código sobre Linux Mint 19 y al estar basado en Ubuntu 18 no debería haber problemas de compatibilidad con otros sistemas, además de que Python es un lenguaje muy portable. No se ha probado en el entorno **conda**, pero si se consiguen instalar los módulos necesarios, no debería haber problemas.

A la hora de implementar el software, se han utilizado tanto módulos ya incluidos en Python, como el módulo **time** para la medición de tiempos, como módulos científicos y para *machine learning*, como por ejemplo **numpy** y **sklearn**. Este último se ha utlizado para poder dividir los datos para el **5 Fold Cross Validation** y para obtener un clasificador KNN que poder utilizar para poder probar los resultados obtenidos por cada uno de los algoritmos. Para la visualización de datos se ha utilizado **pandas**, ya que permite conseguir una visualización rápida de estos gracias a los DataFrames.

Adicionalmente, la estructura de **KDTree** utilizada ha sido sacada de un módulo externo llamado **pykdtree**[1]. Este módulo está implementado en **Cython** y **C** y también utiliza **OMP**, con lo cuál su rendimiento va a ser muy superior a otras implementaciones como por ejemplo el **cKDTree** de **scipy**[1]. En cuanto a su uso, las funciones y la forma de construirlo son las mismas que las de cKDTree, con lo cuál se puede consultar su documentación[2] para obtener más información sobre su uso.

Siendo ahora más concretos en cuanto a la implementación, se ha creado un módulo que contiene tanto código reutilizado de la práctica anterior (creación de particiones, búsqueda local, función de normalización de los datos y funciones objetivo y de evaluación de los datos) como código referente a esta práctica (es decir, la implementación de los algoritmos genéticos y meméticos). Se ha implementado un algoritmo por cada estrategia del genético (un algoritmo para el AGG y uno para el AGE), dando la posibilidad de elegir el operador de cruce a utilizar en cada caso, y una función para el algoritmo memético, que de nuevo, ofrece la posibilidad de escoger la estrategia a utilizar. Además, se han implementado una serie de funciones a las que se ha llamado clasificadores, que se encargan de recorrer las particiones creadas, de ejecutar los respectivos algoritmos pasándoles los datos, entrenar luego un clasificador 1-NN con los pesos calculados y predecir las clases, además de

[1] De hecho, pykdtree está basado en cKDTree y libANN, cogiendo lo mejor de cada implementación y paralelizando el código con OMP para conseguir unos rendimientos muy superiores a ambos, tanto a la hora de crear el árbol como para hacer consultas.

recopilar información estadística para mostrarla luego por pantalla.

Se han utilizado dos semillas aleatorias las cuáles están fijas en el código: una para dividir los datos, y otra para los algoritmos implementados, que se fija al justo antes de llamar a la función que le pasa los datos al algoritmo que se vaya a ejecutar. Los ficheros ARFF proporcionados se han convertido al formato CSV con un script propio, con el objetivo de facilitar la lectura de los datos. Estos archivos también se proporcionan junto con el código fuente implementado.

4. Manual de usuario

Para poder ejecutar el programa, se necesita un intérprete de **Python3**, como se ha mencionado anteriormente. Además, para poder satisfacer las dependencias se necesita el gestor de paquetes **pip** (preferiblemente **pip3**).

Se recomienda instalar las dependencias, las cuáles vienen en el archivo **requirements.txt**, ya que sin ellas, el programa no podrá funcionar. Se recomienda utilizar el script de bash incluido para realizar la instalación, ya que se encarga de instalarlo en un entorno virtual para no causar problemas de versiones con paquetes que ya se tengan instaladas en el equipo o para no instalar paquetes no deseados. Una vez instalados[2], para poder utilizar el entorno creado se debe ejecutar el siguiente comando:

```
$ source ./env/bin/activate
```

Para desactivar el entorno virtual, simplemente basta con ejecutar:

```
(env) $ deactivate
```

Para ejecutar el programa basta con ejecutar el siguiente comando:

```
$ python3 practica2.py [archivo] [algoritmo]
```

Los argumentos **archivo** y **algoritmo** son obligatorios, y sin ellos el programa lanzará una excepción. En cuanto a sus posibles valores:

- **archivo** puede ser: **colposcopy**, **ionosphere** o **texture**.
- **algoritmo** puede ser:
  - Para AG: **genetics-generational-blx**, **genetics-generational-ac**, **genetics-stationary-blx** o **genetics-stationary-ac**.
  - Para AM: **memetics-all**, **memetics-best** o **memetics-rand**.

A continuación, para ilustrar mejor lo explicado hasta el momento, se ofrece una captura de un ejemplo de ejecución del programa. En la imagen se puede ver la siguiente información:

[2]Si se produce algún error durante la instalación de los paquetes, puede ser debido a pykdtree, ya que al necesitar un compilador que soporte OMP puede fallar en los sistemas OSX. Para evitar estos problemas, el programa puede utilizar un cKDTree de scipy en caso de que a la hora de importar pykdtree se produzca un error, suponiendo a cambio una penalización en el tiempo de ejecución.

- Se muestra primero el conjunto de datos sobre el que se va a ejecutar, el clasificador que se va a ejecutar, las opciones que se le pasan a ese clasificador (que determinaran qué algoritmo utilizar) y el tiempo total.
- Se puede ver una tabla en la que aparecen los datos referentes a cada partición (tasa de clasificación, tasa de reducción, agrupación y tiempo).
- Se muestran valores estadísticos para cada variable (valores máximo, mínimo, medio, mediana y desviación típica).

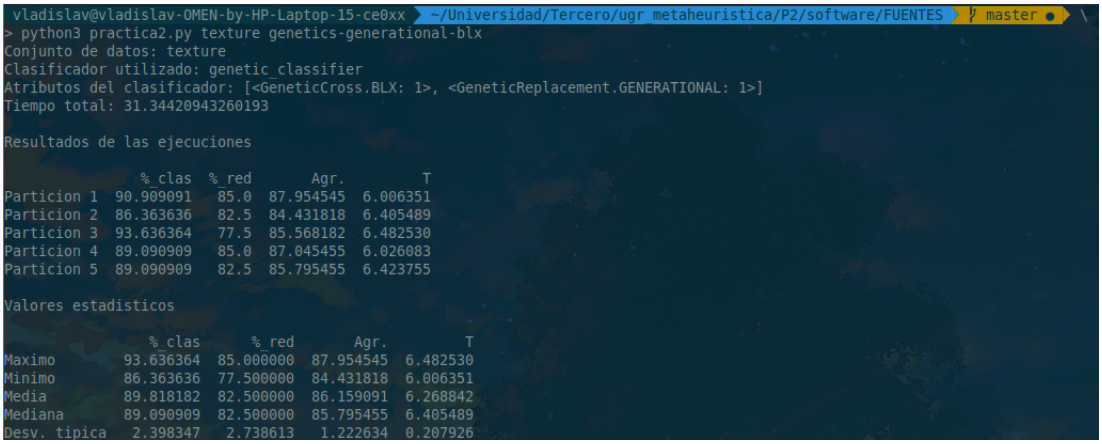

Figura 1: Ejemplo de salida de la ejecución con los datos **texture** y el **AGG** con operador de cruce BLX-α.

5. Análisis de resultados y experimentación

5.1. Descripción de los casos del problema

Para analizar el rendimiento de los algoritmos, se han realizado pruebas sobre 3 conjuntos de datos:

- **Colposcopy**: Conjunto de datos de colposcopias adquirido y anotado por médicos profesionales del Hospital Universitario de Caracas. Las imágenes fueron tomadas al azar de las secuencias colposcópicas. 287 ejemplos con 62 características que deben ser clasificados en 2 clases.
- **Ionosphere**: Conjunto de datos de radar que fueron recogidos por un sistema en *Goose Bay*, Labrador. 352 ejemplos con 34 características que deben ser clasificados en 2 clases.
- **Texture**: Conjunto de datos de extracciones de imágenes para distinguir entre 11 texturas diferentes (césped, piel de becerro prensada, papel hecho a mano, rafia en bucle a una pila alta, lienzo de algodón,...). 550 ejemplos con 40 características que deben ser clasificados en 11 clases.

5.2. Análisis de los resultados

Referencias

[1] Repositroio de GitHub de pykdtree.
`https://github.com/storpipfugl/pykdtree`

[2] Documentación de cKDTree.
`https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html`