UNIVERSIDAD DE GRANADA

Metaheurísticas Problema del Aprendizaje de Pesos en Características

Práctica 3: Técnicas de Búsqueda basadas en Trayectorias para el Problema del Aprendizaje de Pesos en Características

Miguel Ángel Fernández Alonso
mfernandez1@correo.ugr.es
76065289-B
Grupo 3 - Jueves 17:30h

Grado en Ingeniería Informática. Tercero.
Curso 2023-2024

Universidad de Granada
Departamento de Ciencias de la Computación e Inteligencia Artificial

I. Descripción del Problema

El problema del APC consisnte en optimizar el rendimiento de un clasificador basado en vecinos más cercanos a partir de la inclusión de pesos asociados a las características del problema que modifican su valor en el momento de calcular las distancias entre ejemplos.

La variante del problema del APC que afrontaremos busca optimizar tanto la precisión como la complejidad del clasificador. Así se puede formular como:

$Maximizar F(W) = \alpha tasaclas(W) + (1-\alpha) tasared(W)$

Partimos de un vector de datos $T = t_1, ..., t_n$ donde cada dato contiene un conjunto de características o *traits* $x_1, ..., x_n$.

El problema se reduce en calcular un vector de pesos

$$W = w_1, ..., w_n$$

donde el peso

$$w_i \in [0, 1]$$

pondera la característica

$$x_i$$

A este valor lo denominaremos vector de entrenamiento o aprendizaje y nos permitirá clasificar otro conjunto de datos desconocido al que llamaremos vector de test o validación.

Para aumentar la fiabilidad del proceso usaremos el método *5-fold cross validation* que consiste en crear cinco particiones distintas de datos repartidos equitativamente según su clase. Las particiones las dividiremos de tal forma que dediquemos una a *test* y cuatro a *train*.

Para clasificar los datos, implementamos el algoritmo **k-NN** en su versión con *k = 1*. Este algoritmo asigna a cada dato la clase su vecino más cercano calculado a partir de la *distancia euclídea*.

Calcularemos el vector de pesos con los diferentes algoritmos y usaremos nuestro clasificador **k-NN** para valorar el rendimiento de los resultados. Además es necesario calcular los diferentes porcentajes para cada conjunto de datos valorados. [1] [2]

Definiremos:

$tasaclas = 100 \frac{ninstanciasbienclasificadasenT}{ninstanciasenT}$

$tasred = 100 \frac{nvaloresw_i<0,2}{ncaracteristicas}$

Para proceder a los cálculos debemos partir de unos datos normalizados. En la primera prácitica, implementamos un algoritmo **Greedy Relief** y uno de **Búsqueda Local** y en la tercera hemos desarrollado cuatro algoritmos **Búsqueda Multiarranque Básica**, **Enfriamiento Simulado**, **Greedy Aleatorio con Búsqueda Local**, **Búsqueda Local Reiterada**, **Hibridación de ILS y ES**. Los algoritmos son ejecutados sobre un conjunto de datos que contienen los archivos *ecoli, parkinsons, breastcancer* todos ellos en el formato *.csv*.

II. Descripción de los Algoritmos empleados en el Problema

Nuestro esquema de representación de la solución, será un vector de pesos W que nos servirá para valorar la bondad de los datos en clasificaciones futuras. El vector será del tipo $W = w_1, ..., w_n$ donde cada

$$w_i \in [0, 1]$$

y cada w_i gradúa el peso asociado a cada característica y pondera su importancia.

Nuestro algoritmo **k-NN** medirá la bondad de los pesos calculados mediante cada algoritmo, recibiendo como parámetros un vector de datos de entrenamiento, un vector de datos de test y el vector de pesos calculado por el algoritmo seleccionado. El clasificador devolverá un struct *Resultados* que contiene las tasas de clasificación y reducción calculadas a partir del número de aciertos entre ambos conjuntos, así como el tiempo que ha tardado en ejecutarse.

La tasa de clasificación se calcula como la media de los porcentajes de acierto obtenidos por cada método en cada partición del conjunto de datos. A mayor tasa de clasificación, mejor será el vector de pesos W generado por nuestro algoritmo.

La tasa de reducción corresponde al porcentaje de reducción obtenido en la selección del subconjunto de características respecto al total. Una tasa de reducción alta indica que necesitaremos menos atributos para clasificar los datos en un futuro.

Algorithm 1 KNN_LOO

```
Require: train, test, w
Ensure: tasa_clas, tasa_red
aciertos ← 0
num_w_menor ← 0
for i = 0 to tamaño_test do
    pos ← nearestNeighbour_LOO(train, test[i], w)
    if train[pos].clase = test[i].clase then
        aciertos ← aciertos + 1
    end if
end for
for i = 0 to tamaño_w do
    if w[i] <0.2 then
        num_w_menor ← num_w_menor + 1
    end if
end for
tasa_clas ← 100.0 * (aciertos / tamaño_test)
tasa_red ← 100.0 * (num_w_menor / tamaño_w)
```

Para calcular el vecino más cercano, calculamos el que tenga menor distancia euclídea respecto del que estemos valorando. Además, ha sido programado aplicando leave-one-out, ya que cuando se utiliza en la **Búsqueda Local**, no podemos calcular el mejor vecino sin tener en cuenta esto, porque estamos aplicando el **k-NN** sobre el mismo conjunto de entrenamiento y podríamos obtener como mejor vecino el mismo dato que estamos comparando.

Finalmente utilizamos la función *euclideanDistance* para calcular la distancia euclídea entre los diferentes vectores de

Algorithm 2 nearestNeighbour_LOO

Require: train, actual, w
Ensure: pos
1: mejor_distancia ← n_traits_primera_partición_de_train + 1
2: **for** i = 0 to tamaño_train - 1 **do**
{leave-one-out}**if** actual ≠ train[i] **then** {.t accede al vector de características}
3: distancia_actual ← euclideanDistance(train[i].t, actual.t, w)
5: **if** distancia_actual < mejor_distancia **then**
6: mejor_distancia ← distancia_actual
7: pos ← i
8: **end if**
9: **end if**
10: **end for**

características teniendo en cuenta los pesos cuyo valor supere 0.2.

Algorithm 3 euclideanDistance

Require: v1, v2, w
Ensure: dist
1: dist ← 0
2: **for** i = 0 to tamaño_v1 **do**
3: **if** w[i] ≥ 0.2 **then**
4: dist ← dist + w[i] * (v2[i] - v1[i]) * (v2[i] - v1[i])
5: **end if**
6: **end for**

Además, he añadido una versión de la función **k-NN** que utiliza una modalidad del nearestNeighbour sin el criterio de Leave-One-Out. Esta versión del algoritmo será usada cuando apliquemos el **k-NN** sobre dos conjuntos de datos diferentes.

Algorithm 4 nearestNeighbour

Require: train, actual, w
Ensure: pos
1: mejor_distancia ← n_traits_primera_partición_de_train + 1
2: **for** i = 0 to tamaño_train - 1 **do**
3: distancia_actual ← euclideanDistance(train[i].t, actual.t, w)
4: **if** distancia_actual < mejor_distancia **then**
5: mejor_distancia ← distancia_actual
6: pos ← i
7: **end if**
8: **end for**

En los algoritmos implementados he creado un struct *Cromosoma* que contiene la información básica de cada elemento con el que vamos a trabajar (la lista de características y su valor de evaluación).

Listing 1. Definición de la estructura Cromosoma

```
struct Cromosoma {
    vector<double> w;
    double pts;
};
```

En la búsqueda local utilizada en el algoritmo **ILS** el número máximo de iteraciones, son 1000 en lugar de 15000, será el propio ILS el que se encargue de realizar las 15 llamadas de la **BL** para realizar como máximo las 15000 evaluaciones.

Algorithm 5 BL

Require: train, w
1: w ← distribucion_uniforme(0,1)
2: index ← 0,...,w.size()
3: mezcla los valores de index
4: // clasifica el vector de pesos w con KNN y calcula su agregado como tasa de evaluación
5: antiguo ← KNN(train, train, w)
6: agr_ant ← agregado(antiguo.clas, antiguo.red)
7: iter ← 0
8: neighbour ← 0
9: **while** iter <MAX_ITER and neighbour <tamaño w * MAX_NEIGHBOUR **do**
10: aux ← index[iter % tamaño w]
11: w_mut ← w
12: w_mut[aux] += normal(generator)
13: truncar el vector de pesos mutado
14: // clasifica el vector de pesos mutado y calcula su agregado
15: agr_new ← agregado(resultados KNN_LOO)
16: iter++
17: **if** agr_new >agr_ant **then**
18: w ← w_mut
19: agr_ant ← agr_new
20: neighbour ← 0
21: **else**
22: neighbour++
23: **end if**
24: **if** iter % tamaño w = 0 **then**
25: mezcla los valores de index
26: **end if**
27: **end while**

La función agregado calcula la tasa de agregado de los resultados obtenidos al clasificar los datos de entrenamiento con el vector de pesos en cada caso y utiliza este resultado para evaluar la bondad de la solución obtenida. En este caso usamos alpha=0.5.

Algorithm 6 agregado

Require: t_clas, t_red, alpha
1: **return** alpha * t_clas + (1.0 - alpha) * t_red

III. Pseudocódigo de los Algoritmos

III-A. Algoritmo Greedy Aleatorio con Búsqueda Local(GRASP)

Un algoritmo GRASP es un método multiarranque, en el que cada iteración consiste en la construcción de una solución

greedy aleatorizada y la aplicación de una búsqueda local que toma dicha solución como punto inicial de la búsqueda Este procedimiento se repite varias veces y la mejor solución encontrada sobre todas las iteraciones GRASP se devuelve como salida del algoritmo

Construcción de la Solución Inicial en GRASP: Cuando se utiliza la función de selección para construir una. solución greedy, se crea una lista de restringida de candidatos. con un número determinado de mejores candidatos. Se realiza una selección aleatoria de un candidato de la lista. Se adapta la función de selección para recalcular la nueva lista de candidatos para el siguiente paso del proceso constructivo.

La variante más habitual consiste en incluir en la LRC los l mejores candidatos de acuerdo a la función de selección. El parámetro l controla la diversidad de generación de soluciones. Puede ser fijo o variable LRC adaptativa (tamaño variable): Existen variantes que incluyen en la LRC todos los candidatos con un valor de selección por encima de un umbral de calidad *mu* = cmejor ±*alpha*· (cmejor- cpeor) (cmejor=coste mejor candidato; cpeor=coste peor candidato). Esto provoca que la LRC sea adaptativa y tenga un tamaño variable en cada iteración de la etapa de generación Otras variantes incluyen un sesgo en la LRC con una probabilidad mayor de selección de los mejores candidatos También se combina la construcción al azar con la construcción *greedy*, alternando ambos procedimientos aleatoriamente y en secuencias de candidatos

Algorithm 7 crearLRC

Require: costs (vector de costos), alpha
Ensure: LRC (vector de índices)
1: LRC ← {}
2: cmejor ← `min_element(costs)`
3: cpeor ← `max_element(costs)`
4: umbral ← cmejor + alpha × (cpeor - cmejor)
5: **for** i ← 0 **to** `size`(costs) - 1 **do**
6: **if** costs[i] ≤ umbral **then**
7: `push_back(LRC, i)`
8: **end if**
9: **end for**
10: **return** LRC

Algorithm 8 seleccionAleatoria

Require: LRC (vector de enteros)
Ensure: índice seleccionado aleatoriamente
1: dist ← `uniform_int_distribution<int>`(0, `size`(LRC) - 1)
2: índice ← LRC[`dist(generator_BL)`]
3: **return** índice

Junto con una función **calcularCosto**. a función de costo puede calcularse usando el resultado del **KNN_LOO** en la solución parcial.

Con todo esto ya establecido podemos dar paso al pseudocódigo del algoritmo **GRASP**:

Algorithm 9 calcularCosto

Require: train (vector de punteros a FicheroCSV), w (vector de dobles)
Ensure: costo (doble)
1: resultado ← `KNN_LOO(train, train, w)`
2: costo ← `agregado(resultado.tasa_clas, resultado.tasa_red)`
3: **return** costo

Algorithm 10 GRASP

Require: train, best_w, max_iter, alpha
1: best_agr ← -1.0
2: current_agr ← 0.0
3: current_w ← vector(train[0].n)
4: **for** iter ← 0 **to** max_iter **do**
5: // Construcción de la Solución Greedy Aleatorizada
6: current_w ← ConstruccionGreedyAleatorizada(train, alpha)
7: // Aplicar Búsqueda Local
8: BL(train, current_w)
9: // Evaluar la solución actual
10: current_result ← KNN_LOO(train, train, current_w)
11: current_agr ← agregado(current_result.tasa_clas, current_result.tasa_red)
12: // Actualizar la mejor solución
13: **if** current_agr >best_agr **then**
14: best_w ← current_w
15: best_agr ← current_agr
16: **end if**
17: **end for**

III-B. Enfriamiento Simulado (ES)

Para este algoritmo, se empleará el esquema de enfriamiento de Cauchy, que define la temperatura y el valor de *beta* de la siguiente forma:

$$T_{k+1} = \frac{T_k}{1 + \beta * T_k}$$

$$\beta = \frac{T_0 - T_f}{M * T_0 * T_f}$$

Donde *T_0* es la temperatura inicial, *T_f* es la temperatura final y *M* el número de enfriamientos a realizar. Para poder generar soluciones nuevas, deberemos modificar/mutar el vector *W* añadiendo a cada elemento un valor que siga una distribución normal de media 0 y varianza igual a 0.3, pero este método puede proporcionar soluciones negativas, por lo que debemos de truncar los valores negativos a 0. Tendremos un bucle interno para la condición de enfriamiento y uno externo para la condición de parada. La condición de enfriamiento finalizará la iteración actual bien cuando se haya generado un número máximo de vecinos o un número máximo de éxitos. La condición de parada hará finalizar el algoritmo cuando se llegue a las 15000 iteraciones o bien cuando el número de éxitos del enfriamiento actual sea 0. La temperatura inicial tomará el valor siguiente:

$$T_0 = \frac{\mu * C(S_0)}{-ln(\phi)}$$

Donde $\mu = \phi = 0{,}3$
$Tf = 10^-3 < T0$

El pseudocódigo del algoritmo **ES** es el siguiente:

Algorithm 11 ES

Require: train, w
w ← distribucion_uniforme(0,1)
max_vecinos ← 10 × tamaño_w
max_exitos ← 0.1 × max_vecinos
M ← 15000 / max_vecinos
num_exitos ← max_exitos
// Evaluamos la solución inicial
agr_ant ← agregado(resultados_KNN_LOO)
num_eval++
w_mejor ← w
agr_mejor ← agr_ant
// MU = PHI = 0.3 en las ejecuciones
To ← (MU × agr_mejor) / (-log(PHI))
Tf ← 0.001
beta ← (To - Tf) / (M × To × Tf)
Tk ← To
// Comprobamos que Tf es menor que To
while Tf >To **do**
  Tf ← Tf × 0.1
end while
while num_eval <15000 **and** num_exitos ≠ 0 **do**
  num_exitos ← 0
  neighbour ← 0
  while num_eval <15000 **and** num_exitos <max_exitos **and** neighbour <max_vecinos **do**
    mutar sobre una característica aleatoria de w
    truncar el resultado
    // Calculo la tasa de agregado del vector mutado
    agr_new ← agregado(resultados_KNN_LOO)
    num_eval++
    neighbour++
    // Cálculo del incremento de la diff
    diff ← agr_ant - agr_new
    if diff <0 **or** U(0,1) ≤ exp(-diff / Tk) **then**
      w ← w_mut
      agr_ant ← agr_new
      num_exitos++
      if agr_new >agr_mejor **then**
        w_mejor ← w_mut
        agr_mejor ← agr_new
      end if
    end if
  end while
  // Enfriamiento (Esquema de Cauchy)
  Tk ← Tk / (1.0 + beta × Tk)
end while
w ← w_mejor

III-C. Búsqueda Local Reiterada (ILS)

Este algoritmo consiste en generar una solución aleatoria y aplicar de forma reiterada **Búsqueda Local** sobre ella.

Cada vez que apliquemos una **BL** estudiaremos si la solución obtenida es mejor o peor que la mejor anterior y se realizará una mutación sobre la mejor de las dos. La mutación diferirá del resto de mutaciones realizadas en algoritmos ya que usaremos una distribución normal con varianza = 0.4 en vez del 0.3 usual. Realizaremos 15 iteraciones de **BL** y se aplicará la mutación sobre el 10 % de las características del vector de pesos. El pseudocódigo del algoritmo ILS es el siguiente:

Algorithm 12 ILS

Require: train, w
w ← distribucion_uniforme(0,1)
num_mutaciones ← 0.1 × tamaño_w
// Aplicamos la BL y tomamos el agr actual como mejor
BL(train, w)
agr_mejor ← agregado(resultados_KNN_LOO)
w_mejor ← w
// Hacemos 14 iteraciones
for i = 1 **to** 14 **do**
  // Mutamos la mejor solución
  for j = 0 **to** num_mutaciones - 1 **do**
    mutar y truncar una característica aleatoria
  end for
  BL(train, w)
  agr_new ← agregado(resultados_KNN_LOO)
  // Si la nueva solución es mejor se toma esa
  if agr_new >agr_mejor **then**
    w_mejor ← w
    agr_mejor ← agr_new
  end if
  w ← w_mejor
end for

III-D. Búsqueda Multiarranque Básica (BMB)

Una Búsqueda con Arranque Múltiple es un algoritmo de búsqueda global que itera las dos etapas siguientes: Generación de una solución inicial: Se genera una solución S de la región factible Búsqueda Local: Se aplica una BL desde S para obtener una solución optimizada S' Estos pasos se repiten hasta que se satisfaga algún criterio de parada Se devuelve como salida del algoritmo la solución S' que mejor valor de la función objetivo presente La Búsqueda Multiarranque Básica se caracteriza porque las soluciones iniciales se generan de forma aleatoria

Procedimiento **Búsqueda con Arranque Múltiple**:

Para implementar este algoritmo primero he creado la siguiente función: La cual genera una solución.

En algunas aplicaciones, la Etapa 1 se limita a la simple generación aleatoria de las soluciones, mientras que en otros modelos se emplean sofisticados métodos de construcción que consideran las características del problema de optimización para obtener soluciones iniciales de calidad En cuanto a

Algorithm 13 Optimización Iterativa con Generación y Búsqueda Local

```
Sact ← GeneraSolucion() {ETAPA 1}
MejorSolucion ← Sact
repeat
    S' ← BusquedaLocal(Sact) {ETAPA 2}
    if S' es mejor que MejorSolucion then
        MejorSolucion ← S'
    end if
    Sact ← GeneraSolucion() {ETAPA 1}
until (criterio de parada)
return MejorSolucion
```

Algorithm 14 GeneraSolucion

```
Require: Vector de pesos w
for i ← 0 to w.size() − 1 do
    w[i] ← uniform_BL(generator_BL)
end for
```

la Etapa 2, se puede emplear una búsqueda local básica, o procedimientos de búsqueda basados en trayectorias más sofisticados En cuanto a la condición de parada, se han propuesto desde criterios simples, como el de parar después de un número dado de iteraciones, hasta criterios que analizan la evolución de la búsqueda. En muchas de las propuestas que se encuentran en la literatura especializada se fija un número de iteraciones de la **búsqueda local**.

III-E. Hibridación de ILS y ES (ILS-ES)

La hibridación de los algoritmos **ILS** y **ES** (**ILS-ES**) se puede realizar integrando la estructura iterativa y de mutación del ILS con el proceso de enfriamiento y selección del **ES**. El objetivo es aprovechar las ventajas de ambos métodos: la exploración y explotación del **ILS** y el proceso de búsqueda dirigido y adaptativo del **ES**.

Se realizan iteraciones para mutar y mejorar la solución usando ILS y ES. Mutación (ILS): Se mutan un número fijo de componentes del vector w utilizando una distribución normal. Búsqueda Local (ILS): Se aplica la búsqueda local BL a la solución mutada. Evaluación (ILS): Se evalúa la solución mutada y se compara con la mejor solución encontrada. Aceptación de Soluciones (ES): Se aplica un criterio de aceptación basado en una probabilidad dependiente de la temperatura (Tk). Enfriamiento (ES): Se ajusta la temperatura según un esquema de enfriamiento tipo Cauchy. Este enfoque combina la robustez de ILS para encontrar buenas soluciones locales con la capacidad de ES para escapar de óptimos locales a través de un enfriamiento controlado.

IV. Descripción en Pseudocódigo de los Algoritmos de Comparación

El proceso para comparar los datos obtenidos para cada algoritmo es siempre el mismo y se realiza en la función ejecutar para cada algoritmo programado. Primero obtenemos los pesos usando el algoritmo a comparar y seguidamente clasificamos con el **k-NN** los datos de train y test sobre ese

Algorithm 15 BMB

```
Require: Conjunto de entrenamiento train, vector de mejores pesos best_w
Vector de pesos actuales current_w ← crear vector de tamaño best_w.size()
Vector temporal de pesos temp_w ← crear vector de tamaño best_w.size()
Mejor agregado best_agr, agregado actual current_agr
GenerarSolucion(current_w)
BL(train, current_w)
best_w ← current_w
Resultados best_result ← KNN_LOO(train, train, best_w)
best_agr ← agregado(best_result.tasa_clas, best_result.tasa_red
for i to 20 do
    GenerarSolucion(temp_w)
    BL(train, temp_w)
    Resultados temp_result ← KNN_LOO(train, train, temp_w)
    current_agr ← agregado(temp_result.tasa_clas, temp_result.
    if current_agr > best_agr then
        best_w ← temp_w
        best_agr ← current_agr
    end if
end for
return best_w
```

vector de pesos obtenido. Por último devolvemos en un struct Resultados las tasas de clase, reducción, agregado y el tiempo que ha tardado en ejecutar y repetimos el proceso cambiando el índice de la partición de test.

V. Procedimiento del Desarrollo de la Práctica y Manual de Usuario

La práctica ha sido programada en el lenguaje C++. Para el desarrollo de la práctica, primero he necesitado leer los archivos que nos proporcionan los datos, para ello, he optado por pasar los archivos .arff a .csv, un formato que al menos para mí, es más manipulable. Seguidamente y con ayuda de la función read_csv, guardo los datos en memoria en un vector de estructuras que he denominado *FicheroCSV*. Cada *FicheroCSV* contiene la información de cada línea del fichero csv original incluyendo un vector de los datos y un string que indica la clase de los mismos. Por tanto, consideraremos cada conjunto de datos como un *vector<FicheroCSV>*.

Seguidamente, y para aplicar la *5-fold cross validation*, creo un vector con 5 particiones que contienen punteros (para reducir el coste de memoria) a estructuras *FicheroCSV* repartidas de forma equitativa. Los datos se introducen en la partición ya normalizados.

Una vez que tenemos el vector<vector<FicheroCSV*, ya podemos repartir los datos en train y test y proceder como hemos descrito anteriormente con los algoritmos programados.

Para la primera práctica, desarrollé una versión del **k-NN** sin leave-one-out, seguidamente programé el algoritmo **Greedy (Relief)** y finalmente la Búsqueda Local con la cual tuve que

Algorithm 16 ILS-ES

Require: Conjunto de entrenamiento $train$, vector de pesos w
$ENTRENO \leftarrow$ Array de enteros de tamaño $w.size()$
$mejor\_w \leftarrow$ Array de reales de tamaño $w.size()$
$antiguo, nuevo \leftarrow$ Resultados
$agr\_mejor, agr\_nuevo \leftarrow$ Reales
$num\_mutaciones \leftarrow \lceil 0{,}1 \times w.size() \rceil$
$max\_vecinos, max\_exitos, num\_eval, neighbour, num\_exitos \leftarrow$ Enteros
$agr\_ant, diff \leftarrow$ Reales
$To, Tf, beta, Tk \leftarrow$ Reales
$iter \leftarrow$ Entero
// Generación de solución inicial
for $i \leftarrow 1$ to $w.size()$ **do**
    $w[i] \leftarrow distribucion\_uniforme(0, 1)$
end for
// Aplicación de Búsqueda Local y evaluación de la solución inicial
$BL(train, w)$
$antiguo \leftarrow KNN\_LOO(train, train, w)$
$agr\_mejor \leftarrow agregado(antiguo.tasa\_clas, antiguo.tasa\_red)$
for $i \leftarrow 1$ to $w.size()$ **do**
    $mejor\_w[i] \leftarrow w[i]$
end for
// Parámetros de Enfriamiento Simulado (ES)
$To \leftarrow (MU \times agr\_mejor)/(-1{,}0 \times \log(PHI))$
$Tf \leftarrow VAL\_TF$
$beta \leftarrow (To - Tf)/(MAX\_ITER \times To \times Tf)$
$Tk \leftarrow To$
// Ajuste de Tf
while $Tf > To$ **do**
    $Tf \leftarrow Tf \times 0{,}1$
end while
// Iteraciones de ILS-ES
for $iter \leftarrow 1$ to 15 **do**
    // Mutación de la mejor solución (ILS)
    for $j \leftarrow 1$ to $num\_mutaciones$ **do**
        $idx \leftarrow seleccionAleatoria([1..w.size()])$
        $w[idx] \leftarrow w[idx] + normal(generator)$
        if $w[idx] > 1{,}0$ **then**
            $w[idx] \leftarrow 1{,}0$
        else if $w[idx] < 0{,}0$ **then**
            $w[idx] \leftarrow 0{,}0$
        end if
    end for
    // Aplicación de Búsqueda Local y evaluación
    $BL(train, w)$
    $nuevo \leftarrow KNN\_LOO(train, train, w)$
    $agr\_nuevo \leftarrow agregado(nuevo.tasa\_clas, nuevo.tasa\_red)$
    // Verificación y actualización de la mejor solución
    $diff \leftarrow agr\_ant - agr\_nuevo$
    if $diff < 0$ or $U(0, 1) \leq \exp(-diff/Tk)$ **then**
        for $i \leftarrow 1$ to $w.size()$ **do**
            $mejor\_w[i] \leftarrow w[i]$
        end for
        $agr\_mejor \leftarrow agr\_nuevo$
        $agr\_ant \leftarrow agr\_nuevo$
    else
        for $i \leftarrow 1$ to $w.size()$ **do**
            $w[i] \leftarrow mejor\_w[i]$
        end for
    end if
    // Enfriamiento (ES)
    $Tk \leftarrow Tk/(1{,}0 + beta \times Tk)$
    $num\_eval \leftarrow num\_eval + 1$
    if $num\_eval \geq MAX\_ITER$ **then**
        break
    end if
end for
// Asignación de la mejor solución encontrada
for $i \leftarrow 1$ to $w.size()$ **do**
    $w[i] \leftarrow mejor\_w[i]$
end for

Algorithm 17 ejecutarAlgoritmo

Require: particion, i
test $\leftarrow$ toma la partición i
train $\leftarrow$ toma la suma de datos de las particiones $\neq i$
$w \leftarrow$ Algoritmo(train, w)
resultados $\leftarrow$ KNN(train, test, w)

modificar parte del código del **k-NN** para que aplicase el leave-one-out.

Para la segunda práctica, primero programé el código para los cruces **BLX** y **Aritmético** y más tarde desarrollé los 4 algoritmos Genéticos. Por últimos, cambié el código del **Búsqueda Local** para adaptarlo a las restricciones que se imponen en el guión y poder usarlo en los 3 algoritmos Meméticos que se nos piden.

Para la tercera práctica programé los algoritmos en este orden: **Enfriamiento Simulado**, **Búsqueda Local Reiterada** (usando la versión de la práctica 1), **Búsqueda Multiarranque Básica** y **Greedy Aleatorio con Búsqueda Local**.

En el archivo *LEEME.txt* se encuentra explicado el proceso para replicar las ejecuciones del programa. Así que me limitaré a resumirlo: Existe un make que genera el ejecutable del programa. El programa ejecutable ha sido compilado con g++ y optimizado con -O2 para reducir los tiempos de ejecución. ./bin/p3 ./data/archivo.csv seed Ejecuta los seis algoritmos programados en ambas prácticas sobre el conjunto de datos contenido en el archivo.csv (con archivo = colposcopy, ionosphere, texture) usando como semilla el número seed pasado como parámetro en el **ES**, **ILS**, **ILS-ES** y **BMB**.

VI. Experimentos y Análisis de Resultados

Para la obtención de los resultados de esta práctica, hemos utilizado los siguientes conjuntos de datos:

Ecoli: es una base de datos para identificar la posición de proteínas tras obtener métricas mediante una serie de técnicas distintas. Consta de 366 ejemplos. Consta de 8 atributos (clase incluida). Consta de 8 clases (citoplasma, membrana interna, periplasma, ...). Atributos: Método de McGeoch's, Método de Heijne's...

Parkinsons: contiene datos que se utilizan para distinguir entre la presencia y la ausencia de la enfermedad de Parkinson en una serie de pacientes a partir de medidas biomédicas de la voz. Consta de 195 ejemplos. Consta de 23 atributos (clase incluida). Consta de 2 clases (1 Sano, 2 Enfermo), La distribución de ejemplos está desbalanceada (147 enfermos, 48 sanos). Atributos: Distintas métricas sonoras (como periodo pitch, Ratio harmónico-ruido).

Breast-Cancer: es una base de datos para identificar la gravedad de cancer de máma a partir de distintos atributos. Consta de 569 ejemplos. Consta de 31 atributos (clase incluida). Consta de 2 clase (B Benigno, M Maligno). Atributos: Métricas usando 3 imágenes con distinta orientación, cada una con radio, textura, perímetro, área, ... 10 por imágenes, obteniendo 30 en total.

El algoritmo de Búsqueda Local y los Genéticos y Meméticos dependen de un parámetro que especifica la semilla para le generación de números aleatorios. Voy a utilizar para el análisis de estos resultados SEED=42.

Para la generación de vecinos y mutación se van a usar los datos generados por una distribución normal de media 0 y varianza 0.3 (a excepción de **ILS** donde es 0.4). Como criterio de parada en la **Búsqueda Local** utilizada en **ILS**, se va a usar el número de evaluaciones que se han realizado así como el número vecinos por característica explorados. En

nuestro caso los valores serán 1000 y 20 · tam_vector_pesos respectivamente. Las tablas se generan automáticamente con la ejecución del programa y contienen los resultados para cada partición, así como la media de los resultados para cada algoritmo. A continuación, se muestran las tablas para cada fichero y cada algoritmo utilizado:

Teniendo en cuenta solo los algoritmos de la primera práctica, podemos ver que los mejores resultados son obtenidos por el algoritmo **ILS** en las tasas de clasificación, mientras que **BMB** ofrece la mejor tasa de reducción y la mejor medidad global de desemepeño *((Agr.)*. Sin embargo, **BL** es el más rápido en términos de tiempos de ejecución.

Vamos a analizar los resultados globales obtenidos para cada conjunto de datos:

Mejor Tasa de Clasificación: ILS-ES (93.49Mejor Tasa de Reducción: MEJOR P2 (89.86Mejor Agr: MEJOR P2 (90.81) Mejor Tiempo de Ejecución: BL (2.97E-01)

Para el conjunto *Breast-Cancer.csv*, el algoritmo **ILS-ES** presenta la mejor tasa de clasificación, mientras que el mejor algoritmo de la práctica 2 (**Algoritmo Memético BLX (10, 0.1) mejor**) ofrece la mejor tasa de reducción y medida global de desempeño. **BL** es nuevamente el más rápido en términos de ejecución.

Ecoli: ILS destaca en clasificación, BMB en reducción y medida global, y BL en tiempo de ejecución. Parkinsons: **ILS-ES** es el mejor en clasificación y medida global, ES en reducción, y BL en tiempo de ejecución. Breast-cancer: **ILS-ES** se destaca en clasificación, MEJOR P2 en reducción y medida global, y BL en tiempo de ejecución.

En general, **ILS-ES** y **BMB** muestran un rendimiento robusto en varios conjuntos de datos, mientras que BL es consistentemente el más rápido.

REFERENCIAS

[1] M. Mitchell, "Genetic algorithms: An overview." in *Complex.*, vol. 1, no. 1. Citeseer, 1995, pp. 31–39.

[2] B. Melián, J. A. M. Pérez, and J. M. M. Vega, "Metaheurísticas: Una visión global," *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, vol. 7, no. 19, p. 0, 2003.

	Ecoli				Parkinsons				Breast-cancer			
	%_clas	*%red*	*Agr,*	T	*%_clas*	*%red*	*Agr,*	T	*%_clas*	*%red*	*Agr,*	T
BL	62,10	74,00	67,82	5,76E-03	79,50	80,00	79,75	1,63E-02	90,86	83,48	87,17	2,97E-01
MEJOR P2	61,67	76,50	69,09	2,91E-01	82,79	87,30	85,05	2,45E-01	91,94	89,86	90,81	2,49E+00
BMB	65,29	77,50	71,39	1,18E-01	81,00	87,48	84,24	3,13E-01	92,99	88,13	90,56	3,47E+00
ES	60,74	76,00	68,37	2,83E-01	80,64	88,70	84,67	2,54E-01	90,84	89,55	90,20	2,42E+00
ILS	66,40	76,50	71,45	8,95E-02	78,14	87,30	82,72	2,32E-01	91,00	88,13	89,57	2,63E+00
ILS-ES	64,41	76,00	70,20	9,51E-02	84,21	86,26	85,24	2,45E-01	93,49	87,35	90,42	2,81E+00

Figura 1. Tabla datos comparativa

	Ecoli			
	BMB			
	%_clas	*%red*	*Agr,*	T
Partición 1	57,14	75,00	66,07	1,23E-01
Partición 2	57,14	85,00	71,07	1,31E-01
Partición 3	75,16	75,00	75,08	1,20E-01
Partición 4	71,98	75,00	73,49	1,18E-01
Partición 5	65,00	77,50	71,25	9,80E-02
Media	65,29	77,50	71,39	1,18E-01
	ES			
Partición 1	58,57	72,50	65,54	3,08E-01
Partición 2	47,14	82,50	64,82	2,99E-01
Partición 3	63,19	72,50	67,84	2,92E-01
Partición 4	76,48	75,00	75,74	2,86E-01
Partición 5	58,33	77,50	67,92	2,28E-01
Media	60,74	76,00	68,37	2,83E-01
	ILS			
Partición 1	60,00	75,00	67,50	9,70E-02
Partición 2	58,57	80,00	69,29	1,09E-01
Partición 3	72,31	75,00	73,65	8,72E-02
Partición 4	79,45	75,00	77,23	8,58E-02
Partición 5	61,67	77,50	69,58	6,88E-02
Media	66,40	76,50	71,45	8,95E-02
	ILS-ES			
Partición 1	57,14	75,00	66,07	9,80E-02
Partición 2	60,00	77,50	68,75	1,07E-01
Partición 3	65,05	75,00	70,03	9,72E-02
Partición 4	74,84	75,00	74,92	9,06E-02
Partición 5	65,00	77,50	71,25	8,26E-02
Media	64,41	76,00	70,20	9,51E-02

Figura 2. BMB, ES, ILS, ILS-ES Ecoli.csv

	Parkinsons			
	BMB			
	%_clas	*%red*	*Agr,*	T
Partición 1	80,00	87,83	83,91	3,39E-01
Partición 2	82,50	85,22	83,86	3,29E-01
Partición 3	85,00	88,70	86,85	3,22E-01
Partición 4	77,50	84,35	80,92	3,24E-01
Partición 5	80,00	91,30	85,65	2,52E-01
Media	81,00	87,48	84,24	3,13E-01
	ES			
Partición 1	77,50	88,70	83,10	3,01E-01
Partición 2	82,50	86,96	84,73	2,67E-01
Partición 3	82,50	88,70	85,60	2,43E-01
Partición 4	75,00	87,83	81,41	2,56E-01
Partición 5	85,71	91,30	88,51	2,03E-01
Media	80,64	88,70	84,67	2,54E-01
	ILS			
Partición 1	75,00	86,96	80,98	2,58E-01
Partición 2	77,50	84,35	80,92	2,42E-01
Partición 3	75,00	88,70	81,85	2,39E-01
Partición 4	77,50	85,22	81,36	2,37E-01
Partición 5	85,71	91,30	88,51	1,86E-01
Media	78,14	87,30	82,72	2,32E-01
	ILS-ES			
Partición 1	80,00	87,83	83,91	2,63E-01
Partición 2	82,50	84,35	83,42	2,60E-01
Partición 3	87,50	87,83	87,66	2,52E-01
Partición 4	82,50	83,48	82,99	2,57E-01
Partición 5	88,57	87,83	88,20	1,94E-01
Media	84,21	86,26	85,24	2,45E-01

Figura 3. BMB, ES, ILS, ILS-ES Parkinsons.csv

	Breast Cancer			
	BMB			
	%_clas	*%red*	*Agr,*	T
Partición 1	93,91	89,03	91,47	3,5498
Partición 2	93,91	89,68	91,80	3,4912
Partición 3	89,57	85,16	87,36	3,5606
Partición 4	93,91	87,74	90,83	3,5438
Partición 5	93,64	89,03	91,33	3,20E+00
Media	92,99	88,13	90,56	3,47E+00
	ES			
Partición 1	94,78	90,32	92,55	2,47E+00
Partición 2	92,17	91,61	91,89	2,4736
Partición 3	82,61	84,52	83,56	2,4614
Partición 4	93,91	91,61	92,76	2,4798
Partición 5	90,74	89,68	90,21	2,2062
Media	90,84	89,55	90,20	2,42E+00
	ILS			
Partición 1	93,91	88,39	91,15	2,6414
Partición 2	95,65	89,03	92,34	2,7306
Partición 3	86,09	83,87	84,98	2,6624
Partición 4	93,04	90,32	91,68	2,6964
Partición 5	86,32	89,03	87,68	2,44E+00
Media	91,00	88,13	89,57	2,63E+00
	ILS-ES			
Partición 1	99,13	89,03	94,08	2,8058
Partición 2	94,78	85,81	90,29	2,8076
Partición 3	88,70	85,81	87,25	2,9256
Partición 4	92,17	89,68	90,93	2,8786
Partición 5	92,68	86,45	89,57	2,6358
Media	93,49	87,35	90,42	2,81E+00

Figura 4. BMB, ES, ILS, ILS-ES BC.csv