UNIVERSIDAD
DE GRANADA

Metaheurísticas

grado en ingeniería informática

PRÁCTICA 3

Enfriamiento Simulado, Búsqueda Local Reiterada y Evolución Diferencial 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 en todos los casos, cómo se representa la función objetivo o *fitness* y cómo se evalúan las soluciones. También vamos a comentar brevemente otros aspectos, como por ejemplo cómo mantener las soluciones factibles después de aplicarles alguna mutación o transformación (es decir, que se mantengan en el rango dado), entre otros. 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.

Como se dijo al principio, cada solución es un vector de valores reales W en el que $\forall w_i \in W,\ w_i \in [0, 1]$. En el caso de la **Evolución Diferencial** se tendrá un conjunto vectores que formarán la población, ya que sigue un esquema basado en poblaciones. Esto se volverá a comentar más adelante, cuando se hable con más detenimiento de esta técnica.

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, por ejemplo, en la búsqueda local, para hacer que al aplicar el operador de generación de un nuevo vecino la solución siguiese siendo válida. 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 iniciale se han seguido dos esquemas. En uno de ellos, el cuál es utilizado por el **Enfriamiento Simulado** y la **ILS**, se genera un único vector de pesos inicial con valores aleatorios dados por una distribución uniforme en el rango $[0, 1]$. En el otro esquema, el cuál es seguido por la **Evolución Diferencial** se lleva a cabo algo muy parecido, solo que en vez de generar una única solución inicial, se generan una serie de M soluciones iniciales, donde M es

el tamaño de la población. Los pseudocódigos de estos dos esquemas se pueden ver a continuación:

Algorithm 2 Inicialización de un vector de pesos W

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

Algorithm 3 Generación de una población inicial en **Evolución Diferencial**

```
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
```

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 4 Cálculo de la distancia Euclídea entre dos puntos

```
function DistanciaEuclidea(e_1, e_2)
    distancia ← sqrt(sum_{i=1}^{N} (e_1^i − e_2^i)^2)
    return distancia
```

Para la **Evolución Diferencial** sería interesante intentar mantener la población ordenada por valor *fitness*, ya que eso nos va a simplificar bastante el trabajo para uno de los procesos de mutación. 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 5 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 6 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 7 Cálculo de la tasa de reducción (I)

```
1: function CalculoTasaRed(W, N)
2:     caracRed ← 0
3:     for each w_i ∈ W do
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 8 Cálculo de la función objetivo o *fitness*

function CalculoFuncionFitness($tasa\_clas, tasa\_red, \alpha$)
    $fitness \leftarrow \alpha \cdot tasa\_clas + (1 - \alpha) \cdot tasa\_red$
    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. Podemos evaluar la función objetivo tanto para una solución como para una población de soluciones. Por tanto, vamos a tener dos funciones que nos permitan evaluar las soluciones: una para una única solución y una para toda una población de individuos (soluciones), la cuál por debajo utilizará la evaluación simple tantas veces como individuos tenga la población. Vamos a ver el funcionamiento de estas funciones comenzando primero por la que evalúa una única solución, y veamos luego la función que evalúa toda una población de soluciones:

Algorithm 9 Función para evaluar un vector de pesos W

function Evaluar(*datos*, *etiquetas*, W)
    $datosPesos \leftarrow$ aplicar $w_i \in W$ sobre los $x_i \in datos$ donde $w_i > 0.2$
    $arbolKD \leftarrow$ KDTree($datosPesos$)
    $vecinos \leftarrow arbolKD$.ObtenerVecinosMasCercanoL1O($datosPesos$)
    $pred \leftarrow etiquetas[vecinos]$
    $tasa\_clas \leftarrow$ CalcularTasaClas(*etiquetas*, *pred*, num. etiquetas)
    $tasa\_red \leftarrow$ CalcularTasaRed(W, num. caracteristicas)
    $fitness \leftarrow$ CalculoFuncionFitness($tasa\_clas$, $tasa\_red$)
    return $fitness$

Algorithm 10 Función para evaluar una población

function EvaluarPoblacion(*datos*, *etiquetas*, *poblacion*)
    $listaFitness \leftarrow$ NuevoVector()
    for each $W \in poblacion$ **do**
        $listaFitness$.Añadir(Evaluar(*datos*, *etiquetas*, W))
    return $listaFitness$

2.2. Algoritmos de comparación

2.2.1. Clasificador 1-NN

El primer algoritmo con el que compararemos es el 1-NN que utiliza todas las características, tal como hacíamos en la práctica anterior.

Este clasificador lo que hace es, dado un conjunto de valores X que pertenecen a una muestra y un elemento e, decir cuál es el $x \in X$ más cercano a e, y por tanto, decir que e pertenece a la misma clase x. Para determinar cuál es el elemento más cercano se puede usar alguna métrica de distancia, como por ejemplo la distancia Euclídea, descrita anteriormente. A la hora de implementarlo, para poder acelerar los cálculos, se puede usar un KDTree, ya que permite realizar una consulta rápida (en los casos más favorables su complejidad temporal es $\mathcal{O}(\log n)$, mientras que en el peor caso es $\mathcal{O}(n)$) utilizando la distancia Euclídea para determinar el vecino más cercano, con la penalización de que tarda un tiempo $\mathcal{O}(n)$ en ser construido. Para poder ver un esquema de su funcionamiento, se ofrece el siguiente pseudocódigo:

Algorithm 11 Clasificador 1-NN

```
1: function KNN(X, y, e)
2:     arbolKD ← KDTree(X)
3:     x ← arbolKD.VecinoMasCercano(e)
4:     etiqueta ← y[x]
5:     return etiqueta
```

2.2.2. Algoritmo greedy *RELIEF*

Otro 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 12 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 13 Cálculo de los pesos mediante *RELIEF* (I)

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

Algorithm 14 Cálculo de los pesos mediante *RELIEF* (II)

```
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
```

2.2.3. Búsqueda Local

En la primera práctica se implementó una búsqueda local (búsqueda basada en trayectorias), así que vamos a rescatarla para tener un algoritmo más de comparación, ademñas de que los AM la necesitarán luego, aunque ligeramente modificada. 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).

El pseudocódigo de la búsqueda local se puede ver aquí:

Algorithm 15 Cálculo de los pesos mediante la Búsqueda Local (I)

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

Algorithm 16 Cálculo de los pesos mediante la Búsqueda Local (II)

```
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
```

Referencias

[1] Texto referencia
`https://url.referencia.com`