METAHEURÍSTICAS

Práctica 1 - APC

Grupo 1

Mario Muñoz Mesa
mario43514@correo.ugr.es

10 de abril de 2022

Índice

1. Descripción del problema.

Nos ocupamos del problema del Aprendizaje de Pesos en Características (APC). Plantea, en el contexto de un problema de clasificación, elegir o ajustar un vector de pesos $\mathbf{w} = (w_1, \ldots, w_d) \in [0,1]^d$ asociado a las características en base a un criterio de mejora del clasificador. Hemos denotado con d al número de características.

En nuestro caso trabajaremos con clasificador tipo 1-NN, y el criterio será maximizar:

$$F(\mathbf{w}) := \alpha \cdot \text{tasa\_clas}(\mathbf{w}) + (1-\alpha) \cdot \text{tasa\_red}(\mathbf{w})$$

donde

$$\text{tasa-clas} := 100\frac{\text{nº instancias bien clasificadas en training}}{\text{nº instancias en training}}, \qquad \text{tasa-red} := 100\frac{\text{nº valores } w_i < 0.2}{\text{nº características}}$$

y $alpha = 0.5$ que pondera la importancia entre el acierto y la reducción de características para el nuevo mejor clasificador que se pretende encontrar.

De lo que nos ocupamos por tanto es de obtener $\arg\max_{\mathbf{w}\in[0,1]^d} F(\mathbf{w})$ para un clasificador 1-NN que utilizará la distancia ponderada

$$d_{\mathbf{w}}(u,v) = \sqrt{\sum_{i=1}^{d} w_i(u_i - v_i)^2}, \quad u, v \in \mathbb{R}^d$$

para clasificar. Queremos que aumente el acierto y reduzca el número de características, cualquier característica con peso menor estricto a 0.2 se descarta.

2. Descripción de la aplicación de los algoritmos empleados.

2.1. Asociados al preprocesado y representación.

Partimos de un conjunto de entrenamiento T formado por instancias de una muestra. Cada instancia se ha representado mediante una estructura de datos `SampleElement` que recoge las valores de las características en un vector `vector<double>` y la clase como una cadena de caracteres.

De esta forma, cada dataset se lee mediante la función `read_arff` y lo almacenamos como un vector de elementos muestrales `vector<SampleElement>`.

Para la normalización utilizamos min-max normalization, se ha implementado en `normalization`.

Las particiones para la validación cruzada $k-fold$ se realizan respetando la proporción de clases en cada partición, para ello cada elemento de una clase se va asignando a una partición de forma cíclica.

La solución será un vector de pesos con valores entre 0 y 1, cada componente indica el peso de una característica, se representa como un `vector<double>`

2.2. Funciones generales.

Nuestro clasificador utiliza la distancia euclídea, para manejarla se han implementado dos funciones: `euclidean_distance2` y `euclidean_distance2_w`.

Algorithm 1: euclidean_distance2

Input: vector de reales a, vector de reales b
Output: la distancia euclídea al cuadrado entre a y b
begin
  dist $\leftarrow 0$
  for $i = 0$ *to* $a.size() - 1$ **do**
    dist $\leftarrow$ dist $+ (a[i] - b[i])^2$
  end
  return dist
end

Algorithm 2: euclidean_distance2_w

Input: vector de reales a, vector de reales b, vector de pesos $weights$
Output: la distancia euclídea ponderada y al cuadrado entre a y b
begin
  dist $\leftarrow 0$
  for $i = 0$ ***to*** $a.size() - 1$ **do**
    `// se descartan las características con peso <2`
    if $weights[i] \geq 0.2$ **then**
      dist $\leftarrow$ dist $+ (a[i] - b[i])^2$
    end
  end
  return dist
end

La función `euclidean_distance2` se utiliza exclusivamente para el cálculo de amigo y enemigo más cercano en el algoritmo Relief. Con `euclidean_distance2_w` calculamos la distancia ponderada descartando pesos menores a 0.2, se utiliza en 1-NN y Búsq. Local.

En ambos casos calculamos la distancia euclídea al cuadrado por motivos de eficiencia y porque lo que nos interesa es comparar distancias, y como la raíz cuadrada es una función creciente, si $a > b \Rightarrow \sqrt{a} > \sqrt{b}$, no hay inconveniente en trabajar con la distancia al cuadrado.

El algoritmo 1-NN, haciendo uso de distancia ponderada, se ha implementado en `one_NN`.

Algorithm 3: one_NN

Input: elemento muestral *sam_el* a clasificar, conjunto de elementos muestrales sobre el que buscar el más cercano *sam_elements*, vector de pesos *weights*
Output: la clase o etiqueta del elemento más cercano, *min_l*

```
begin
    min_dist ← ∞
    for e in sam_elements do
        d ← euclidean_distance2_w(e.features, sam_el.features, weights)
        if d < min_dist then
            min_l ← e.label
            min_dist ← d
        end
    end
    return min_l
end
```

La versión con leave-one-out, que se utiliza en Búsq. Local, simplemente consiste en dejar un elemento fuera.

Algorithm 4: one_NN_lo

Input: elemento muestral *sam_el* a clasificar, conjunto de elementos muestrales sobre el que buscar el más cercano *sam_elements*, vector de pesos *weights*, posición de elemento a dejar fuera *leave_out*
Output: la clase o etiqueta del elemento más cercano, *min_l*

```
begin
    min_dist ← ∞
    for i = 0 to sam_elements.size() − 1 do
        if i ≠ leave_out then
            d ← euclidean_distance2_w(sam_elements[i].features, sam_el.features, weights)
            if d < min_dist then
                min_l ← sam_elements[i].label
                min_dist ← d
            end
        end
    end
    return min_l
end
```

2.3. Función objetivo.

Para el cálculo de las tasas se han implementado las funciones `class_rate` y `red_rate` que simplemente implementan la propia de definición de tasa-class y tasa-red.

Algorithm 5: class_rate

Input: vector con clases o etiquetas de elementos clasificados $class\_labels$, elementos de test $test$

Output: el porcentaje de acierto

begin

$n \leftarrow 0$

$class\_labels\_s \leftarrow class\_labels.size()$

for $i = 0$ ***to*** $class\_labels\_s - 1$ **do**

if $class\_labels\_s[i] == test[i].label$ **then**

$n \leftarrow n + 1$

end

end

return $100\frac{n}{class\_labels\_n}$

end

Algorithm 6: red_rate

Input: vector de pesos $weights$

Output: el porcentaje de reducción

begin

$feats\_reduced \leftarrow 0$

for w ***in*** $weights$ **do**

if $w < 0.2$ **then**

$feats\_reduced \leftarrow feats\_reduced + 1$

end

end

return $100\frac{feats\_reduced}{initial\_number\_of\_features}$

end

La función objetivo, teniendo en cuenta que trabajamos con $\alpha = 0.5$, nos quedaría:

Algorithm 7: obj_function

Input: tasa de clasificación $class\_rate$, tasa de reducción red_rate

Output: el valor de la función objetivo o fitness

begin

return $\alpha \cdot class\_rate + (1 - \alpha) \cdot red\_rate$

end

3. Descripción de los algoritmos.

3.1. RELIEF (Greedy).

Este algoritmo será el de comparación. El objetivo es generar un vector de pesos a partir de las distancias de cada ejemplo o elemento al amigo (misma clase) y enemigo (clase distinta) más cercano.

La idea subyacente es, para cada característica, premiar en peso la lejanía de un ejemplo a su enemigo más cercano (separa bien), y penalizar la lejanía de un ejemplo a su amigo más cercano.

Se comienza con vector de pesos 0. Se recorren los elementos de entrenamiento, identificando enemigo y amigo más cercano, y se actualizan las componentes de los pesos sumando la distancia del enemigo más cercano y restando la del amigo más cercano en cada característica. Luego se

restringen los pesos a $[0, 1]$ truncando los valores negativos a 0 y normalizando el resto dividiendo por el máximo peso en el vector de pesos.

Para la implementación se ha creado previamente la función `closest_friend_enemy` que busca el amigo y enemigo más cercanos a un elemento.

Algorithm 8: closest_friend_enemy

Input: posición del elemento *pos*, conjunto de entrenamiento *training*, elemento que será el amigo más cercano $friend\_e$, elemento que será el enemigo más cercano *enemy_e*

begin
- $min\_friend\_d \leftarrow \infty$
- $min\_enemy\_d \leftarrow \infty$
- **for** $i \in \{0, \ldots, pos - 1, pos + 1, \ldots, training.size() - 1\}$ **do**
  - $distance \leftarrow euclidean\_distance2(training[i].features, training[pos].features)$
  - **if** *elemento i y elemento pos son amigos* **then**
    - **if** $distance < min\_friend\_d$ **then**
      - $min\_friend\_d \leftarrow distance$
      - $min\_friend\_p \leftarrow i$
    - **end**
  - **end**
  - **else**
    - **if** $distance < min\_enemy\_d$ **then**
      - $min\_enemy\_d \leftarrow distance$
      - $min\_enemy\_p \leftarrow i$
    - **end**
  - **end**
- **end**
- $friend\_e \leftarrow training[min\_friend\_p]$
- $enemy\_e \leftarrow training[min\_enemy\_p]$

end

El agoritmo RELIEF:

Algorithm 9: relief

Input: conjunto de elementos muestrales $training$
Output: vector de pesos $weights$
begin
  $weights \leftarrow (0, \ldots, 0)$
  `// se va buscando enemigo y amigo más cercano para cada elemento`
  for $i = 0$ ***to*** $training.size() - 1$ **do**
    $closest\_friend\_enemy(i, training, friend, enemy)$
    `// se actualizan los pesos`
    for $j = 0$ ***to*** $num\_feats - 1$ **do**
      if $distance < min\_friend\_d$ **then**
        $weights[j] \leftarrow weights[j] + |training[i].features[j] - enemy.features[j]| - |training[i].features[j] - friend.features[j]|$
      end
    end
  end
  $max\_weight \leftarrow \max_{i \in \{0,\ldots,num\_feats-1\}} weights[i]$
  `// se restringe a [0,1]`
  for $j = 0$ ***to*** num_feats **do**
    if $weights[j] < 0$ **then**
      $weights[j] \leftarrow 0$
    end
    else
      $weights[j] \leftarrow \frac{weights[j]}{max\_weight}$
    end
  end
  return weights
end

3.2. Búsqueda Local.

Para la Búsqueda Local se emplea la estrategia del primer mejor. La idea es generar soluciones (vectores de pesos) vecinas hasta encontrar la primera mejor que la actual, en ese caso se sustituye la actual por la encontrada. Este proceso es iterativo.

Para ello se parte de una solución (vector de pesos) inicial; se toman valores de una distribución uniforme $\mathcal{U}(0, 1)$.

Para generar las soluciones vecinas, se va eligiendo una componente aleatoria sin repetición. Las soluciones vecinas se generan mutando el vector de pesos actual al sumar un elemento de una distribución normal de media 0 y desviación típica 0.3 (pesos fueras del rango [0,1] se truncan). Si se mejora la función objetivo con el nuevo vector de pesos, éste pasará a ser el actual. Además, en ese caso, o en el caso de haber recorrido todas las componentes, se establece un nuevo orden aleatorio de las componentes a recorrer.

Este proceso se repite iterativamente hasta alcanzar 15000 evaluaciones de la función objetivo o hasta la generación de un número de vecinos igual a 20 veces el número de características.

Para representar los índices de componentes del vector de pesos, que eventualmente se mezclan según lo anterior, se ha construido un vector de enteros. Este vector junto con el vector de pesos se inicializan en la función `rand_init_sol_gen_indexes`

Algorithm 10: rand_init_sol_gen_indexes

Input: vector en el que inicializar los pesos $weights$, vector de índices $indexes$
begin
  for $i = 0$ ***to*** $number\_of\_features - 1$ **do**
    $weights.push\_back(e \in \mathcal{U}(0,1))$
    $indexes.push\_back(i)$
  end
end

También hemos construido una función que realiza la mutación en el vector de pesos:

Algorithm 11: mutation

Input: vector de pesos $weights$, índice a mutar i, distribución normal n_dist
begin
  $weights[i] \leftarrow weights[i] + e \in \mathcal{N}_{n\_dist}(0, 0.3)$
  if $weigths[i] > 1$ **then**
    $weights[i] \leftarrow 1$
  end
  else
    if $weights[i] < 0$ **then**
      $weights[i] \leftarrow 0$
    end
  end
end

Y una función que clasifica un conjunto de elementos mediante leave one out, con distancia ponderada, y obtiene el correspondiente valor de la función objetivo para esa clasificación:

Algorithm 12: classifyloo_and_objf

Input: elementos de entrenamiento conjunto de entrenamiento $training$, vector de pesos $weights$, vector a contener las etiquetas de clasificación $class\_labels$
Output: el valor de la función objetivo de la clasificación, obj
begin
  for $k = 0$ ***to*** $training.size() - 1$ **do**
    $class\_labels.push\_back(one\_NN\_lo(training[k], training, weights, k))$
  end
  $obj \leftarrow obj\_function(class\_rate(class\_labels, training), red\_rate(weights))$
  $class\_labels.clear()$
  return obj
end

Finalmente tenemos la función que realiza la búsqueda local:

Algorithm 13: local_search

Input: conjunto de elementos muestrales *training*
Output: mejor vector de pesos alcanzado *weights*
begin
  $rand\_init\_sol\_gen\_indexes(best\_weights, comp\_indexes)$
  `// clasificamos con los mejores pesos hasta el momento mediante leave one out, y obtenemos valor de la función objetivo`
  $best\_obj \leftarrow classifyloo\_and\_objf(training, best\_weights, class\_labels)$

  while $gen\_neighbours < max\_gen\_neighbours$ ***and*** $obj\_eval\_count < max\_objf\_evals$ **do**
    `// aleatorizamos componentes a mutar si se han recorrido todas o se mejoró f.obj`
    if new_best_obj ***or*** $mod\_pos \% number\_of\_features == 0$ **then**
      $new\_best\_obj \leftarrow false$
      $shuffle(comp\_indexes)$
      $mod\_pos \leftarrow 0$
    end
    `// Se toma componente a mutar y se muta`
    $comp\_to\_mut \leftarrow comp\_indexes[mod\_pos \% number\_of\_features]$
    $muted\_weights \leftarrow best\_weights$
    $mutation(muted\_weights, comp\_to\_mut, n\_dist)$
    $gen\_neighbours \leftarrow gen\_neighbours + 1$
    `// clasificamos con los nuevos pesos mutados mediante leave one out, y obtenemos valor de la función objetivo`
    $obj \leftarrow classifyloo\_and\_objf(training, muted\_weights, class\_labels)$
    `// Si se ha mejorado, actualizamos mejor objetivo y weights, y vecinos generados`
    if $obj > best\_obj$ **then**
      $best\_weights \leftarrow muted\_weights$
      $best\_obj \leftarrow obj$
      $gen\_neighbours \leftarrow 0$
      $new\_best\_obj \leftarrow true$
    end
    $obj\_eval\_count \leftarrow obj\_eval\_count + 1$
    $mod\_pos \leftarrow mod\_pos + 1$
  end
  return $best\_weights$
end

4. Procedimiento considerado en el desarrollo.

Para el desarrollo de la práctica no se ha utilizado ningún framework, se han implementado las funciones necesarias en C++. Todo el código se encuentra en `p1.cpp`.

Para la ejecución basta indicar un argumento con el archivo a ejecutar, en este caso se utiliza la semilla por defecto 1234, que es la misma que se utilizó para los resultados de esta memoria.

También se incluye un fichero `p1_all_datasets.sh` que ejecuta, con la semilla por defecto, `p1` sobre los 3 datasets.

5. Experimentos y análisis de resultados.

Los 3 datasets con los que se ha trabajado son:

- **Ionosphere**: Conjunto de datos de radar que fueron recogidos por un sistema *Goose Bay*, Labrador. Consta de 352 instancias con 34 características y 2 clases.
- **Parkinsons**: Conjunto de datos orientado a distinguir entre la presencia y ausencia de la enfermedad de Parkinson. Consta de 195 ejemplos con 22 características y 2 clases.
- **Spectf-Heart**: Conjunto de datos de detección de enfermedades cardiacas a partir de imágenes médicas de tomografía computerizada del corazón de pacientes. Consta de 267 ejemplos con 44 características y 2 clases.

Los resultados obtenidos son:

1-NN

Nº part.	Ionosphere				Parkinsons				Spectf-Heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	87.3239	0	43.662	0.373	95	0	47.5	0.181	85.7143	0	42.8571	0.468
P2	64.2857	0	32.1429	0.325	25	0	12.5	0.073	72.8571	0	36.4286	0.407
P3	64.2857	0	32.1429	0.325	25.641	0	12.8205	0.072	72.8571	0	36.4286	0.397
P4	64.2857	0	32.1429	0.325	23.6842	0	11.8421	0.07	72.8571	0	36.4286	0.397
P5	64.2857	0	32.1429	0.381	23.6842	0	11.8421	0.071	73.5294	0	36.7647	0.388
Media	68.89334	0	34.44672	0.3458	38.60188	0	19.30094	0.0934	75.563	0	37.78152	0.4114

Relief

Nº part.	Ionosphere				Parkinsons				Spectf-Heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	88.7324	2.94118	45.8368	1.753	95	9.09091	52.0455	0.364	90	0	45	2.28
P2	84.2857	5.88235	45.084	1.757	97.5	4.54545	51.0227	0.366	87.1429	0	43.5714	2.222
P3	90	2.94118	46.4706	1.76	94.8718	4.54545	49.7086	0.371	80	0	40	2.201
P4	88.5714	2.94118	45.7563	1.761	92.1053	4.54545	48.3254	0.369	84.2857	0	42.1429	2.172
P5	87.1429	2.94118	45.042	1.766	100	0	50	0.374	77.9412	0	38.9706	2.237
Media	87.74648	3.529414	45.63794	1.7594	95.89542	4.545452	50.22044	0.3688	83.87396	0	41.93698	2.2224

Búsqueda Local

Nº part.	Ionosphere				Parkinsons				Spectf-Heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
P1	78.8732	85.2941	82.0837	3311.36	87.5	86.3636	86.9318	390.064	85.7143	88.6364	87.1753	6719.91
P2	92.8571	85.2941	89.0756	4673.61	82.5	90.9091	86.7045	408.803	85.7143	86.3636	86.039	7458.06
P3	95.7143	82.3529	89.0336	1986.46	87.1795	86.3636	86.7716	339.172	88.5714	88.6364	88.6039	5512.85
P4	88.5714	88.2353	88.4034	3914.9	86.8421	50	68.4211	265.049	85.7143	77.2727	81.4935	3448.21
P5	90	85.2941	87.6471	2388.89	97.3684	86.3636	91.866	581.204	80.8824	88.6364	84.7594	3612.81
Media	89.2032	85.2941	87.24868	3255.044	88.278	79.99998	84.139	396.8584	85.31934	85.9091	85.61422	5350.368

Y la tabla resumen:

Nº part.	Ionosphere				Parkinsons				Spectf-Heart			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
1-NN	68.89334	0	34.44672	0.3458	38.60188	0	19.30094	0.0934	75.563	0	37.78152	0.4114
Relief	87.74648	3.529414	45.63794	1.7594	95.89542	4.545452	50.22044	0.3688	83.87396	0	41.93698	2.2224
Búsq. Local	89.2032	85.2941	87.24868	3255.044	88.278	79.99998	84.139	396.8584	85.31934	85.9091	85.61422	5350.368

En cuanto a las tasas, en primer lugar vemos gráficamente la tasa de clasificación:

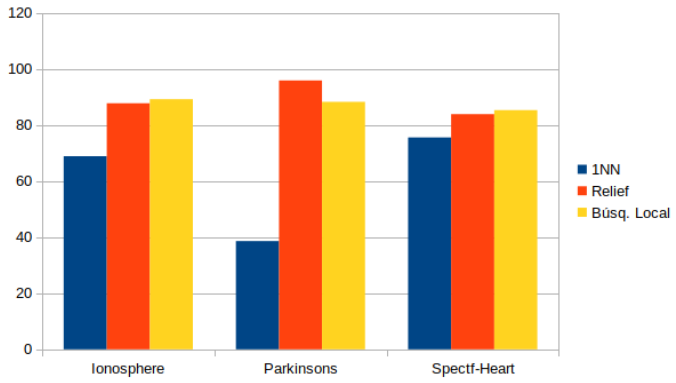

Figura 1: Tasa de clasificación.

Podemos observar que en general 1-NN ofrece peores tasas de clasificación, mientras que Relief y Búsq. Local alcanzan tasas similares. Esto tiene sentido pues en 1-NN se tiene que elegir el ejemplo o elemento más cercano usando todas las características mientras que con Relief y Búsq. Local, tras el ajuste de pesos, se tiene la posibilidad de no considerar ciertas características que probablemente no sean relevantes y solo estén introduciendo "ruido" en la clasificación (se descartan las de peso < 0.2).

Vemos ahora las tasas de reducción:

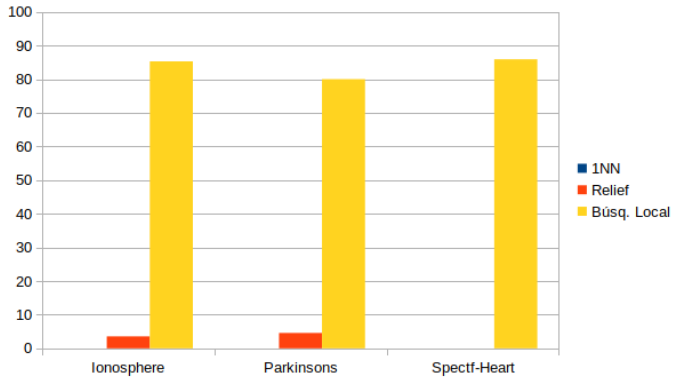

Figura 2: Tasa de reducción.

Claramente las tasas de reducción con 1-NN son 0 por lo comentado. Se observa además una gran diferencia entre Búsq. Local y Relief, alcanzando mucha más reducción Búsq. Local. Esto también era de esperar pues en Búsq. Local, con $\alpha = 5$ como nuestro caso, lo que se busca es un equilibrio entre tasa de clasificación y tasa de reducción (se va evaluando la función objetivo y es el criterio de elección) mientras que en Relief simplemente se van modificando los pesos en función de los amigos y enemigos más cercanos, la tasa de reducción no se tiene en cuenta en Relief.

Por último el valor agregado:

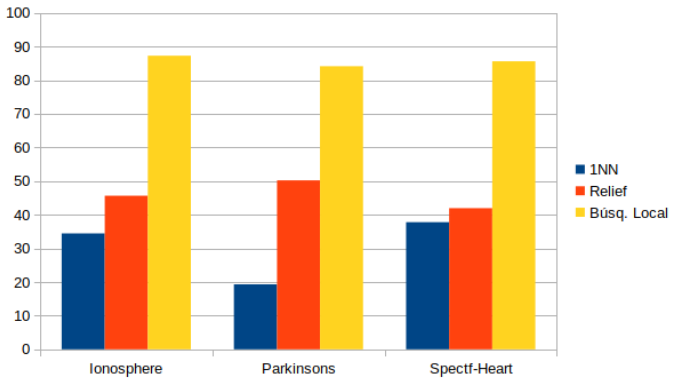

Figura 3: Valor de función objetivo o agregado.

Observamos una clara superioridad de Búsq. Local, seguido de Relief, y por último 1-NN. Era predecible pues 1-NN ha sido el que peores tasas ha mostrado, y aunque Búsq. Local y Relief tenían tasas de clasifiación similares, la tasa de reducción en Búsq. Local es mucho mayor que la de Relief.

En cuanto a los tiempos:

Como observamos en la tabla resumen, Relief es bastante rápido, tiene tiempos de ejecución similares a 1-NN y alcanza buenas tasas de clasificación, supone una gran mejora respecto a 1-NN. Búsq. Local sin embargo tiene tiempos de ejecución bastante superiores a ambos, aunque como hemos visto, los valores que se alcanzan en la función objetivo son muy superiores aunque sean de óptimos locales.