Asignatura (2018-2019)
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1.b: Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema del Aprendizaje de Pesos en Características

---

Simón López Vico, 26504148Y
simondelosbros@correo.ugr.es
Grupo de prácticas: Miércoles

1 de abril de 2019

Índice

1. Breve descripción del problema.

En esta práctica nos encargaremos de resolver el problema del Aprendizaje de Pesos en Características, que consistirá en ajustar un vector de pesos $\{\omega_1, \ldots, \omega_n\}$ donde cada una de sus componentes determinará el valor de la característica i-ésima del conjunto de datos. Si $\omega_i = 1$ significará que el atributo i-ésimo es muy importante, mientras que si $\omega_i = 0$ la característica i-ésima no será relevante; concretamente, cuando $\omega_i < 0{,}2$ descartaremos dicho atributo.

Para ajustar el vector de pesos utilizaremos dos algoritmos distintos: Búsqueda Local y RELIEF. Una vez optimizado el vector, usaremos el clasificador K-NN (con K=1) para clasificar según el vecino más cercano usando una distancia ponderada según los pesos:

```
function Distancia(vector a, vector b, vector pesos)
    Si tamaño(a) != tamaño(b)
        Devuelve -1

    Para i=0 hasta tamaño(a)
        Si pesos[i]>=0.2
            Distancia += ( pesos[i]*(a[i]-b[i]) )^2

    Devuelve Distancia;
```

Véase que no estamos calculando la raíz para computar la distancia euclídea; prescindimos de esto porque los resultados serán los mismos calculando y sin calcular la raíz, y además la operación `sqrt(x)` tiene un alto tiempo de computo.

Para ajustar el vector de pesos W sobre el clasificador trataremos de maximizar la función $F(W) = \alpha * tasa_{clas}(W) + (1-\alpha) * tasa_{red}(W)$, donde $tasa_{clas}$ determinará el porcentaje de instancias bien clasificadas sobre el total de instancias, y $tasa_{red}$ el porcentaje de atributos con un peso asociado menor a 0.2 sobre el total de atributos.

2. Descripción de la aplicación de los algoritmos empleados al problema.

2.1. Esquema de representación del problema y formateo de los ficheros de datos.

La resolución de este problema APC ha sido implementada en C++. Para el almacenamiento de los datos recogidos en los distintos ficheros `arff` se ha usado un vector llamado `data` en el que cada componente será un `pair <vector<double>, int >`; el primer elemento del `pair` recogerá los atributos de una instancia del conjunto total de datos, mientras que el segundo será un `int` con la etiqueta de la instancia. El vector `data`

será separado en 5 particiones disjuntas más adelante para realizar el *5-fold validation*.

Por otra parte, los ficheros `arff` han sido ligeramente modificados para facilitar la lectura de éstos:

- **colposcopy.arff**: ninguna modificación.
- **ionosphere.arff**: cambio de las clases b y g por 0 y 1.
- **texture.arff**: inicialmente cada uno de los atributos de cada instancia se encuentra separado por una coma y un espacio; se modifica para solamente se separen mediante una coma.
  Clases iniciales: {2, 3, 4, 9, 10, 7, 6, 8, 12, 13, 14}.
  Clases modificadas: {0, 1, 2, 6, 7, 4, 3, 5, 8, 9, 10}.

2.2. Lectura y preprocesado.

Comenzaremos leyendo el fichero de datos pasado al programa con la función `read_data`, la cual leerá un archivo en formato `arff` y guardará todos sus valores en el vector `data`; además, almacenará el total de atributos y de clases del dataset leído.

A continuación normalizaremos los datos para que se encuentren en el intervalo $[0, 1]$. Para ello usaremos la siguiente fórmula:

$$x_j^N = \frac{x_j - Min_j}{Max_j - Min_j},$$

donde Max_j y Min_j serán el máximo y el mínimo de cada uno de los atributos. Hay que tener en cuenta que si un atributo toma el mismo valor para todas las instancias, la diferencia entre el máximo y el mínimo será 0 por lo que no podremos realizar la división; en este caso, $x_j^N = 0$.

Una vez normalizados todos los datos, realizaremos las particiones necesarias para el *K-fold validation*, en nuestro caso K=5. Para ello usaremos el siguiente código:

```
function k folds()
    Mezclar(data)

    Para i=0 hasta n_clases
        Para j=0 hasta tamaño(data)
            Si data.clase == i
                particiones[total_introducidos%n_part].push_back(data[j]);
```

De esta manera, el número de clases estará balanceada respecto a cada partición, es decir, en cada partición habrá el mismo porcentaje (aproximadamente) de elementos con una

clase determinada.

Notemos que la normalización se debe de aplicar siempre antes de realizar las particiones de elementos, pues si se hiciera al contrario cada partición estaría normalizada por un máximo y mínimo distinto.

Finalmente generaremos los conjuntos *train* y *test* con los que vamos a entrenar y validar nuestro clasificador.

3. Algoritmo de Búsqueda Local.

El algoritmo de búsqueda local comenzará generando un vector de pesos mediante una distribución uniforme el cual consideraremos como solución inicial. Además, ya que en cada paso de exploración es necesario mutar una componente distinta sin repetición, crearemos un vector de índices de longitud `n_atrib`. Cada vez que este vector se recorra por completo, barajaremos sus componentes.

La búsqueda local se ejecutará hasta que se hayan generado un máximo de 20*`n_atrib` número de vecinos sin mejora o cuando se hayan realizado 15000 evaluaciones de la función objetivo. En cada iteración de este bucle, se copiará el vector de pesos a un vector `tmp` sobre el cual se realizará una mutación en el atributo i-ésimo y se comprobará el valor de la función objetivo con esta mutación. La mutación consistirá en sumar en la coordenada i-ésima del vector un valor generado aleatoriamente mediante una distribución normal; si se superara el valor 1 o quedara por debajo de 0, trucaremos a 1 o 0. Si dicha mutación mejora el valor de la función de evaluación, guardaremos el vector `tmp` como nuevo vector de pesos, y en caso contrario mantendremos el vector de pesos anterior.

El pseudocódigo para realizar una mutación será:

```
mutacion(pesos, index)
    pesos[i]+=distribution_normal()

    Si pesos[i]>1
        pesos[i]=1
    Si pesos[i]<0
        pesos[i]=0
```

Finalmente, el pseudocódigo de la búsqueda local será el siguiente:

```
function busqueda_local()
    pesos=generar_solucion_inicial()
    indices=generar_indices()
    valor_eval=0
```

```
Mientras n_generados < max_generados && n_eval < max_eval
    Si se ha recorrido el vector indices
        Mezclar(indices)

    tmp=pesos
    mutacion(tmp, indices[i])
    new_valor=f_evaluacion(tmp)

    Si nuevo_valor > valor_eval
        pesos=tmp
        valor_eval=new_valor

Devuelve pesos
```

4. Algoritmo de Comparación: RELIEF.

Este algoritmo se encargará de ir modificando los pesos de acuerdo a la distancia de cada elemento a su amigo (elemento con la misma etiqueta) y enemigo (elemento con distinta etiqueta) más cercano. Si la distancia al amigo más cercano es mayor que al enemigo más cercano, los pesos disminuirán, y en caso contrario aumentarán.

El algoritmo comenzará inicializando el vector de pesos a 0 y recorrerá todos los elementos del conjunto de entrenamiento, buscando el enemigo y el amigo más cercano y modificando el valor de los pesos según éstos. Tras ello, normalizará los pesos coordenada a coordenada, dividiéndolos entre el máximo de todos ellos (si el valor es mayor que 1) o igualando a 0 (si el valor es menor que 0).

Las funciones para buscar los enemigos y amigos más cercanos devolverán el índice donde se encuentran éstos. El pseudocódigo para enemigos más cercanos será:

```
function close_enemy_index(e)
    indice_min=-1
    dist_min=(Valor muy grande)

    Para i=0 hasta tamaño(train)
        Si train[i].clase != e.clase
            dist_tmp=distancia_euclidea(e, train[i])
            Si d_tmp<d_min
                indice_min=i;
                dist_min=dist_tmp
    Devuelve indice_min
```

El pseudocódigo para amigos más cercanos será:

```
function close_friend_index(e, index)
    indice_min=-1
    dist_min=(Valor muy grande)

    Para i=0 hasta tamaño(train)
        Si train[i].clase == e.clase && index != i
            dist_tmp=distancia_euclidea(e, train[i])
            Si d_tmp<d_min
                indice_min=i;
                dist_min=dist_tmp
    Devuelve indice_min
```

Finalmente, el pseudocódigo del algoritmo RELIEF será:

```
function relief()
    pesos={0}
    Para i=0 hasta tamaño(train)
        enemy_ind=close_enemy_index(train[i])
        friend_ind=close_friend_index(train[i], i)
        pesos += |train[i]-train[enemy_ind]| - |train[i]-train[friend_ind]|

    peso_max=maximo(pesos)

    Para j=0 hasta n_atrib
        Si pesos[j]<0
            pesos[j]=0
        Si pesos[j]>1
            pesos[j]=pesos[j]/peso_max

    Devolver pesos
```

5. Manual de usuario.

Para replicar los resultados de mis algoritmos y el funcionamiento de mi código, habría que empezar por formatear los archivos de datos tal y como se han comentado en la subsección **Esquema de representación del problema y formateo de los ficheros de datos.** Tras ello, bastaría con compilar el código con la instrucción `make` y ejecutar el archivo `ejecutar.sh`, el cual generará un fichero `solucion.txt` con los resultados de la ejecución de los algoritmos sobre los 3 ficheros de datos. Es posible que hubiera que dar permisos de ejecución al archivo `ejecutar.sh`.

El archivo `solucion.txt` será del siguiente estilo:

```
---------------------- Datos usados: datos -------------------------

========================================================================
========================================================================
                              PARTICION N
========================================================================
========================================================================

Algoritmo:          Tasa clas: VALOR               Tasa red: VALOR
*************** F evaluacion: VALOR (tiempo: VALOR) ******************
```

6. Experimentos y análisis de resultados.

Aparte de los tres algoritmos que pide la práctica, he modificado un poco la búsqueda local para que funcione con distintos valores de *alpha*, generando así dos tablas de resultados más las cuales contendrán la búsqueda local teniendo en cuenta solo la tasa de clasificación ($\alpha = 1$) y la búsqueda local teniendo en cuenta solo la tasa de reducción ($\alpha = 0$). La tabla con los resultados globales obtenida es la siguiente: [1]

Tabla 5.6: Resultados globales en el problema del APC

	Colposcopy				Ionosphere				Texture			
	%_clas	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T	*%_clas*	*%red*	*Agr.*	T
1-NN	73,91018	0	36,95509	0,0034644	86,32596	0	43,16298	0,0046984	92,36366	0	46,18183	0,0109022
RELIEF	76,00358	28,0645	52,03404	0,0128048	86,61568	2,94118	44,77843	0,0152	93,45456	10	51,72728	0,0476242
BL	83,62772	63,2258	73,42676	8,862942	94,01608	81,17646	87,59627	6,885478	93,0909	78	85,54545	31,01356
BL (alpha=1)	78,41212	17,09676	47,75444	5,874614	91,17504	24,11764	57,64634	3,238906	93,63636	18,5	56,06818	11,519358
BL (alpha=0)	73,86262	87,0968	80,47971	14,725676	88,03222	84,11764	86,07493	8,364536	69,8182	88	78,9091	29,27454

Nota: El resto de tablas se encontrarán en el archivo `Tablas_APC_2018-19.xls` adjunto con la práctica.

Comencemos comparando solamente los algoritmos concernientes a la práctica. Cada uno de ellos tiene un tiempo de ejecución diferente, 1-NN el más rápido, RELIEF tras él y Búsqueda Local el más lento; por tanto, al realizar el análisis habrá que tener en cuenta ésta velocidad de ejecución.

Veamos con una gráfica generada sobre las tablas de Excel una comparación entre los valores de tasa de clasificación de cada uno de los algoritmos.

[1] Los valores para BL (alpha=1) y BL (alpha=0) no son exactamente los mismos en la tabla y en la ejecución de los algoritmos, ya que pasé todos los datos a la tabla y cuando quedaba poco tiempo para la entrega encontré un pequeño fallo en el código. Aún así, los resultados difieren poco.

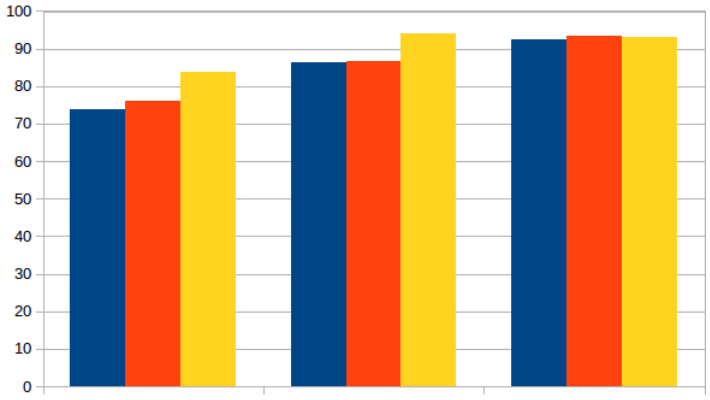

Figura 6.1: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. 1-NN en azul, RELIEF en naranja, BL en amarillo.

Con estos datos podemos ver que la búsqueda local gana a los otros dos algoritmos salvo en el dataset *texture*, en el cual el algoritmo RELIEF supera por poco a BL (93,45456 frente a 92,36366). El hecho de que RELIEF supere en este caso a BL puede ser simplemente por el dataset usado, cambiando de valores si se tomaran unos datos distintos de texturas; pero también puede haber sido causado por haber dado una importancia del 50 % a la tasa de clasificación en el algoritmo de búsqueda local, y éste habría tratado de encontrar un equilibrio entre la tasa de clasificación y la de reducción.

Veamos ahora la tasa de reducción:

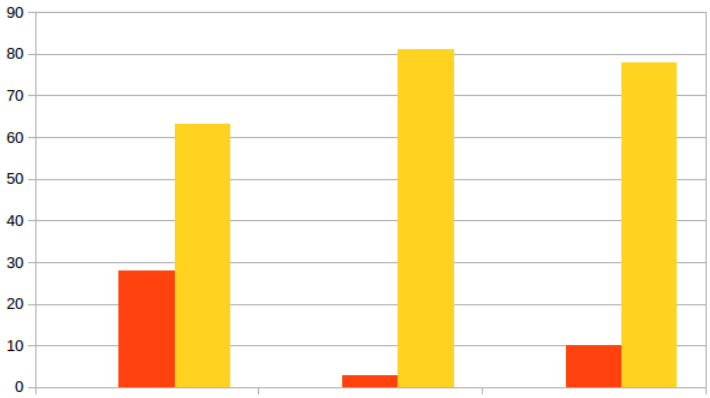

Figura 6.2: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. 1-NN en azul (inexistente), RELIEF en naranja, BL en amarillo.

Tal y como se esperaba, la tasa de reducción del algoritmo RELIEF es mucho menor a la de BL; esto es debido a que el algoritmo de búsqueda local busca un equilibrio entre la tasa de reducción y la tasa de clasificación (como antes comentábamos), mientras que RELIEF se basa en modificar los pesos en función de los amigos y enemigos más cerca-

nos, sin tener en cuenta la tasa de reducción para estos cálculos, por lo que la tasa de reducción asociada a los resultados de estos algoritmos no es relevante durante el ajuste de los pesos, dándole un valor bastante menor.

Finalmente, comentemos los datos de la función de evaluación:

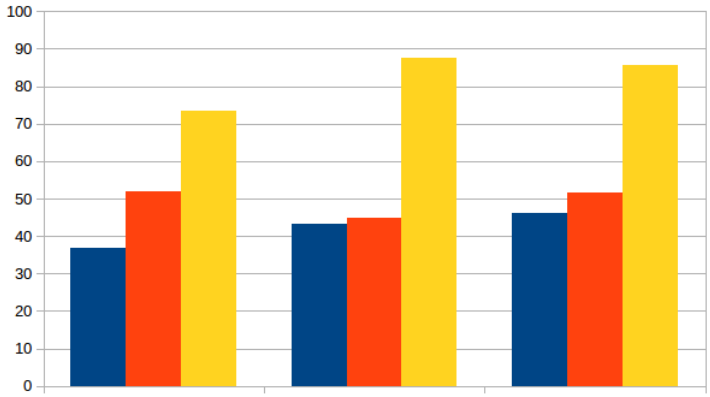

Figura 6.3: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. 1-NN en azul, RELIEF en naranja, BL en amarillo.

Simplemente con los valores de la tasa de clasificación y de reducción comentados anteriormente ya deberíamos de haber adivinado la forma de esta gráfica final, pues el valor de tasa de clasificación y de reducción del algoritmo de búsqueda local supera en (casi) todos los casos a los otros dos algoritmos.

Aunque el tiempo de ejecución de la búsqueda local sea mucho mayor que el de 1-NN y RELIEF (teniendo en cuenta que los tiempos de estos dos últimos son muy pequeños), este tiempo apenas supera los 30 segundos para un conjunto de datos de 550 ejemplos como es el dataset *texture*, por lo que será conveniente usar el algoritmo de búsqueda local para realizar el ajuste de pesos para el clasificador 1-NN. Aún así, en ciertos casos habremos de tener en cuenta el tamaño del dataset sobre el que vamos a trabajar, ya que si el tiempo es una restricción a tener en cuenta, nos podría convenir usar el algoritmo RELIEF, mucho más rápido que BL.

6.1. Modificaciones de la Búsqueda Local.

Si tomamos distintos valores de α para calcular la función de evaluación en el algoritmo de Búsqueda Local podremos obtener distintos resultados, dándole más o menos importancia a la tasa de clasificación y la tasa de reducción. En este experimento adicional vamos a ser drásticos, comparando BL por defecto (con $\alpha = 0{,}5$), BL con $\alpha = 1$ y BL con $\alpha = 0$. Comencemos generando una gráfica con los valores de la tasa de clasificación:

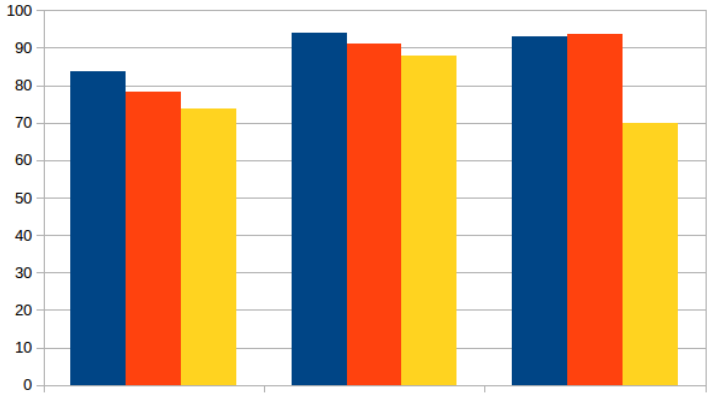

Figura 6.4: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. BL en azul, BL-1 en naranja, BL-0 en amarillo.

Aún habiendo modificado el valor de α para el cálculo de la búsqueda local, el caso donde $\alpha = 0{,}5$ sigue dando mejores resultados de tasa de clasificación que el resto, y tal cosa puede ser causada por el hecho de que los datasets estén preparados para un ajuste equilibrado de tasa de clasificación y reducción.

Continuemos con la tasa de reducción:

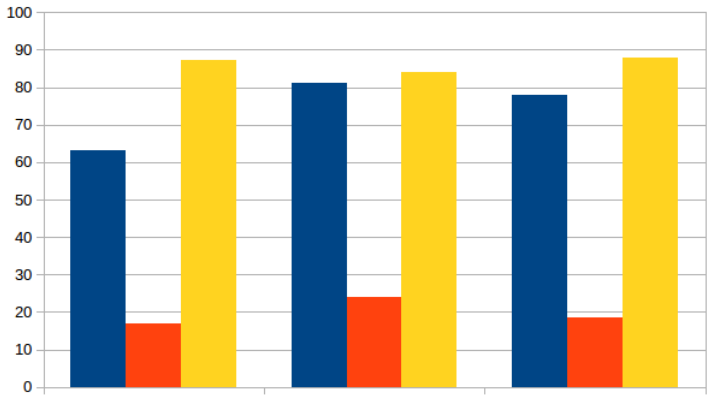

Figura 6.5: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. BL en azul, BL-1 en naranja, BL-0 en amarillo.

En este caso sí que podemos comprobar la gran importancia que el algoritmo BL-0 le ha dado a la tasa de reducción, superando en los tres casos a BL y a BL-1. Por otra parte, el algoritmo BL-1 ha obtenido una tasa de reducción muy baja, lo que le llevará a malos resultados para la función de evaluación.

Finalmente, los resultados sobre la función de evaluación son:

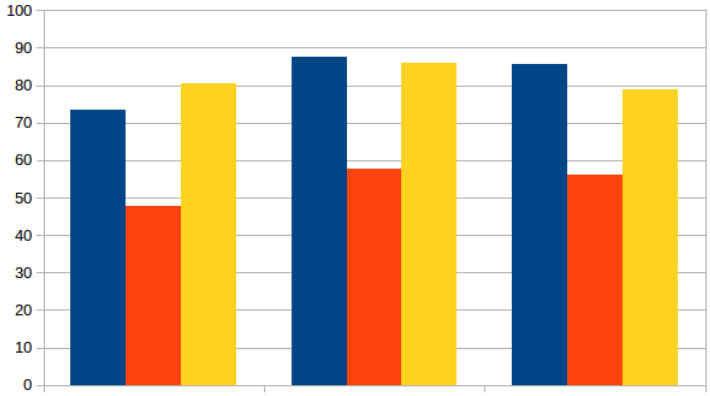

Figura 6.6: A la izquierda, *colcoscopy*; en el centro *ionosphere*; a la derecha, *texture*. BL en azul, BL-1 en naranja, BL-0 en amarillo.

Podemos observar que en el primer caso el algoritmo BL-0 supera a BL, en el segundo están muy igualados (ganando BL) y en el tercero supera BL a BL-0. Respecto al algoritmo BL-1, no gana en ninguno de los casos y ni si quiera se acerca a los resultados de los otros dos, debido a sus valores pequeños de tasa de reducción.

Así, podemos comprobar que en algunos dataset la tasa de reducción es más importante que la de clasificación y viceversa, y habría que realizar algún estudio previo para optimizar el valor de α y obtener un ajuste mejor sobre el vector de pesos.