Metaheurísticas

Aprendizaje de Pesos en Características

Práctica 1: RELIEF, BL

Antonio Coín Castro
77191012E
antoniocoin@correo.ugr.es
Grupo 1 (M 17:30-19:30)

26 de abril de 2019

1. Descripción del problema

En esta práctica y las siguientes consideraremos el marco general de un problema de clasificación. Fijado $n \in \mathbb{N}$, un *clasificador* es cualquier función $c : \mathbb{R}^n \to C$, donde C es un conjunto (finito) de clases prefijadas. Consideramos además un conjunto de *entrenamiento* $T \subseteq \mathbb{R}^n$ de elementos ya clasificados: para cada $t \in T$, denotamos $\Gamma(t) \in C$ a su clase, que es conocida.

El problema de clasificación consiste en, dado un conjunto de *prueba* $T' \subseteq \mathbb{R}^n$ no observado previamente, encontrar un clasificador c que maximice el número de clases correctamente clasificadas en T', tras haber sido entrenado sobre los elementos de T.

Uno de los clasificadores más conocidos y más sencillos es el clasificador $k-$NN, que asigna a cada elemento la clase que más se repite entre sus k vecinos más cercanos. En el caso concreto de esta práctica trabajaremos con el clasificador $1-$NN **con pesos**: a cada elemento le asignamos la clase de su vecino más cercano, pero ponderamos la distancia en función de un vector de pesos $w \in [0,1]^n$.

Para el cálculo de la distancia entre dos elementos de $\mathbb{R}^n$ utilizaremos la *distancia euclídea* ponderada por el ya mencionado vector de pesos:

$$d_w(t,s) = \sqrt{\sum_{i=1}^{n} w_i (s_i - t_i)^2}, \quad t, s \in \mathbb{R}^n$$

La idea tras estos pesos es que midan la importancia de cada una de las características que representan las coordenadas de los elementos $n-$dimensionales considerados, asignando más peso en el cálculo de la distancia a aquellas que sean más importantes. El problema de **aprendizaje de pesos en características** persigue justamente "aprender" cuál debe ser el valor de cada peso en una instancia concreta del problema.

Para medir la bondad de un clasificador con pesos utilizamos las siguientes métricas:

- La **precisión** (T). Estudiámos cuántos ejemplos del conjunto de prueba se clasifican correctamente, entrenando previamente el clasificador (que utiliza la distancia d_w) con el conjunto de entrenamiento.
- La **simplicidad** (R). Un clasificador será más simple si tiene en cuenta un menor número de características. Diremos que una característica $i \in \{1, \ldots, n\}$ no se considera en el cálculo de la distancia si el peso asociado w_i es menor que 0,2.

Así, el problema consiste en encontrar un vector de pesos $w \in [0,1]^n$ que maximice la precisión y la simplicidad, es decir, que maximice lo que llamaremos la *función objetivo*:

$$F(w) = \alpha T(w) + (1 - \alpha) R(w).$$

2. Descripción de la aplicación de los algoritmos

En esta sección se describen los elementos comunes a todos los algoritmos desarrollados, así como los esquemas de representación de datos de entrada y soluciones. Todo el código se ha desarrollado en `C++11`.

Esquemas de representación

En primer lugar, los datos de entrada se encuentran en la carpeta `data`. Se trata de tres conjuntos de datos **ya normalizados** en formato `csv`, donde cada fila representa un ejemplo con los valores de sus características separados por '`;`' y el último elemento de la fila es su clase.

Para representar los datos en el programa se emplea una estructura `Example` que recoge toda la información necesaria: un `vector<double>` con los valores de cada una de las n características del ejemplo concreto, así como un `string` que representa su clase o categoría.

```
struct Example {
    vector<double> traits;
    string category;
    int n;
}
```

Cada conjunto de datos se representa entonces por un `vector<Example>`, y se emplea la función `read_csv` para rellenar el vector, que va leyendo los archivos línea a línea.

Además, como será necesario hacer particiones de cada conjunto de datos para implementar la técnica de *K-fold cross validation*, se proporciona la función `make_partitions` que se encarga de repartir los elementos entre los K conjuntos considerados, respetando la proporción original de clases. La forma de hacer esto es simplemente ir poniendo cada clase de forma cíclica en las particiones, primando el reparto equitativo de clases al reparto equitativo de elementos.

Por su parte, la solución es un `vector<double>` del mismo tamaño que el número de categorías consideradas en cada caso. La componente $i-$ésima del vector representa el peso otorgado a la característica $i-$ésima de cada ejemplo del problema.

Operadores comunes

Todos los algoritmos hacen uso del cálculo de la distancia. Como dijimos, para este cálculo se emplea la distancia euclídea, eventualmente ponderada mediante un vector de pesos. En el caso de que la distancia deseada sea la estándar, se asume que los pesos valen siempre 1 (en la implementación realmente hay dos funciones separadas, una con pesos y otra sin pesos).

```
function DISTANCE_SQ_WEIGHTS(e1, e2, w)
    distance = 0
    for i := 0 to n − 1 do                          ▷ n es el número de características
        if w[i] ≥ 0.2 then
            distance += w[i] * (e2[i] - e1[i]) * (e2[i] - e1[i])
```

Cabe destacar que en realidad estamos calculando al distancia euclídea al cuadrado, pues solo vamos a utilizarla para comparar. Como la función $f(x) = \sqrt{x}$ es creciente para $x \geq 0$ no hay problema en que hagamos esto, pues se mantiene el orden. De esta forma ahorramos tiempo de cálculo, pues esta función va a ser llamada muchas veces a lo largo del programa.

También tenemos la función `classifier_1nn_weights`, que clasifica un ejemplo basándose en la técnica del vecino más cercano. Debemos pasarle también el conjunto de entrenamiento con los ejemplos ya clasificados, y el vector de pesos. De nuevo, si queremos que el clasificador no tenga en cuenta los pesos podemos asumir que son todos 1, aunque en realidad hay dos funciones separadas.

```
function CLASSIFIER_1NN_WEIGHTS(e, training, self, w)
    selected = 0
    dmin = ∞
    for i := 0 to n − 1 do                          ▷ n es el número de ejemplos de entrenamiento
        if i ≠ self then
            dist = distance_sq_weights(e, training[i], w)
            if dist < dmin then
                dmin = dist
                selected = i
     return training[selected].category
```

Función objetivo

La función objetivo que queremos maximizar se implementa tal y como se dijo en la descripción del problema, donde el valor α prefijado es de 0.5, dando la misma importancia a la precisión y a la simplicidad.

```
objective(class_rate, red_rate) {
  return alpha * class_rate + (1.0 - alpha) * red_rate;
}
```

Para calcular la tasa de clasificación y de reducción utilizamos otras funciones también muy sencillas. La primera mide el porcentaje de acierto sobre un vector de elementos que el clasificador

ha clasificado, y cuya clase real conocemos. La segunda simplemente contabiliza qué porcentaje de los pesos son menores que 0,2.

```
function CLASS_RATE(classified, test)
    correct = 0
    for i := 0 to n − 1 do                          ▷ n es el número de ejemplos clasificados
        if classified[i] == test[i].category then
            correct++
     return 100,0 * correct / n
```

```
function RED_RATE(w)
    discarded = 0
    for i := 0 to n − 1 do                          ▷ n es el tamaño del vector de pesos
        if w[i] < 0,2 then
            discarded++
     return 100,0 * discarded / n
```

3. Descripción de los algoritmos considerados

En esta sección se describen los dos algoritmos implementados en esta práctica para el problema del APC. En ambos lo que se pretende es rellenar un vector de pesos para maximizar la función objetivo.

Algoritmo *greedy* RELIEF

Este es un algoritmo voraz muy sencillo, que servirá como caso base para comparar las diferentes metaheurísticas desarrolladas. Se trata de buscar, para cada ejemplo, su amigo (misma clase) y su enemigo (distinta clase) más cercano. Después, componente a componente se suma al vector de pesos la distancia a su enemigo, y se resta la distancia a su amigo.

En este proceso es posible que los pesos se salgan del intervalo $[0, 1]$, por lo que al finalizar es necesario normalizarlos. Además, no debemos olvidar que el algoritmo comienza con el vector de pesos relleno de 0s. Disponemos para ello de una función `init_vector` que se encarga de darle un valor inicial de 0 a todas las componentes del vector de pesos.

Separamos la función que se encarga de buscar los amigos y enemigos más cercanos, teniendo en cuenta que el amigo más cercano no puede ser el propio ejemplo.

```
function NEAREST_EXAMPLE(training, e, self)
    dmin_friend = ∞
    dmin_enemy = ∞
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos de entrenamiento
        if i ≠ self then
            dist = distance_sq(e, training[i])
            if training[i].category ! = e.category and dist <dmin_enemy then
                n_enemy = i
                dmin_enemy = dist
            else if training[i].category == e.category and dist <dmin_friend) then
                n_friend = i
                dmin_friend = dist
     return n_enemy, n_friend
```

El algoritmo RELIEF se detalla ya en la página siguiente.

```
function RELIEF(training)
    w = init_vector()
    for i := 0 to n − 1 do                          ▷ n es el número de ejemplos de entrenamiento
        n_enemy, n_friend = nearest_example(training, training[i], i)
        for j := 0 to m − 1 do                      ▷ m es el tamaño del vector de pesos
            w[j] = w[j] + |training[i].traits[j] - training[n_enemy].traits[j]|
                   − |training[i].traits[j] - training[n_friend].traits[j]|
    max = max(w)
    for j := 0 to m − 1 do                          ▷ normalizamos los pesos
        if w[j] < 0 then
            w[j] = 0
        else
            w[j] = w[j] / max
    return w
```

Algoritmo de búsqueda local

Empleamos la técnica de búsqueda local del **primer mejor** para rellenar el vector de pesos. La idea es mutar en cada iteración una componente aleatoria y **distinta** del vector de pesos, sumándole un valor extraído de una normal de media 0 y desviación típica $\sigma = 0{,}3$. Si tras esta mutación se mejora la función objetivo, nos quedamos con este nuevo vector, y si no lo desechamos. Si algún peso se sale del intervalo $[0, 1]$ tras la mutación, directamente lo truncamos a 0 ó a 1.

Para la generación de la solución inicial sobre la que iterar, consideramos valores extraídos de una distribución uniforme $\mathcal{U}(0, 1)$, que obtenemos gracias al tipo (de la librería `<random>`) `uniform_real_distribution<double>`. A la hora de escoger qué componente vamos a mutar, tenemos un vector de índices del mismo tamaño que el vector de pesos, que barajamos de forma aleatoria y recorremos secuencialmente. Si llegamos al final, volvemos a barajarlo para seguir generando nuevas soluciones.

```
// Initialize index vector and solution
for (int i = 0; i < n; i++) {
  index.push_back(i);
  w[i] = uniform_real(gen);
}
shuffle(index.begin(), index.end(), gen);
```

Para escoger el valor con el que se muta cada componente utilizamos esta vez el tipo predefinido `normal_distribution<double>`. Para determinar si una mutación mejora, utilizamos como métrica el valor de la función objetivo, tomando la tasa de clasificación sobre el propio conjunto de entrenamiento (*leave-one-out*).

En primer lugar, separamos la comprobación de mejora en la función objetivo para más claridad.

```
function EVALUATE(training, classified, w)
    for i := 0 to n − 1 do                    ▷ n es el número de ejemplos de entrenamiento
        classified.push_back(classifier_1nn_weights(training[i], training, i, w))
    o = objective(class_rate(classified, training), red_rate(w))
    classified.clear()
    return o
```

Mostramos ahora el procedimiento de la búsqueda local.

```
function LOCAL_SEARCH(training)
    w, index = initialize()          ▷ ejecuta el código de inicialización mostrado anteriormente
    best_objective = evaluate(training, classified, w)
    while iter < MAX_ITER and neighbour < n * MAX_NEIGHBOUR_PER_TRAIT do
        comp = index[iter % n]
        w_mut = w
        w_mut[comp] += normal(gen)
        if w_mut[comp] > 1 then
            w_mut[comp] = 1
        else if w_mut[comp] < 0 then
            w_mut[comp] = 0
        current_objective = evaluate(training, classified, w)
        iter++
        if current_objective > best_objective then
            mut++
            neighbour = 0
            w = w_mut
            best_objective = current_objective
            improvement = true
        else
            neighbour++
        if iter % n == 0 or improvement then
            shuffle(index.begin(), index.end(), gen)
            improvement = false
    return w
```

Notamos que detenemos la búsqueda cuando llegamos al máximo de iteraciones, o cuando generamos un número de vecinos (dependiente del número de características) sin mejorar la función objetivo.

4. Procedimiento considerado para desarrollar la práctica

Todo el código de la práctica se ha desarrollado en C++ siguiendo el estándar 2011. Se utiliza la biblioteca `std` y otras bibliotecas auxiliares, pero no se ha hecho uso de ningún *framework* de metaheurísticas.

Para todos los procedimientos que implican aleatoreidad se utiliza un generador de números aleatorios común (llamado `gen`), inicializado con una semilla concreta. La semilla por defecto es 20, aunque se puede especificar otra mediante línea de comandos. La evaluación de los tres algoritmos considerados (1−NN, RELIEF y búsqueda local) se realiza mediante la función `run`.

Se proporciona un makefile para compilar los archivos y generar un ejecutable, mediante la orden `make`. A la hora de ejecutarlo hay dos opciones:

- Pasarle como parámetro una semilla para el generador aleatorio, y a continuación una lista de archivos sobre los que ejecutar los algoritmos (ruta relativa).
- Ejecutarlo sin argumentos. En este caso, utiliza la semilla por defecto y ejecuta los algoritmos sobre los tres conjuntos de datos de la carpeta `DATA`.

El código está disponible en la carpeta `FUENTES`, y consta de los siguientes módulos:

- `p1.cpp` Contiene la implementación de los algoritmos, y las funciones necesarias para ejecutarlos.
- `timer` Módulo para medir tiempos en UNIX.
- `util` Se trata de funciones auxiliares para el preprocesamiento de los archivos de datos, el cálculo de la distancia, etc.

Al compilar se genera un único ejecutable en la carpeta `BIN` de nombre `p1`.

5. Resultados

Descripción de los casos del problema

Se consideran tres conjuntos de datos sobre los que ejecutar los algoritmos:

- **Colposcopy.** La colposcopia es un procedimiento ginecológico que consiste en la exploración del cuello uterino. Consta de 287 ejemplos, 62 atributos reales y dos clases: positivo o negativo.
- **Ionosphere.** Datos de radar recogidos por un sistema en Goose Bay, Labrador. Consta de 351 ejemplos, 34 atributos y dos clases: retornos buenos (g) y malos (b).
- **Texture.** El objetivo de este conjunto de datos es distinguir entre 11 texturas diferentes. Consta de 550 ejemplos, 40 atributos y 11 clases (tipos de textura).

Resultados obtenidos

A continuación se muestran las tablas de unos resultados obtenidos para cada uno de los algoritmos. El orden de las columnas es siempre el mismo: primero *colposcopy*, después *ionosphere*, y por último *texture*.

Para cada conjunto de datos se muestra una tabla con cada una de las 5 ejecuciones realizadas, de acuerdo a la técnica $5-$*fold cross validation*. En cada una de ellas se muestran los valores de la tasa de clasificación (Clas), tasa de reducción (Red), función objetivo (Agr) y tiempo de ejecución (T) **en milisegundos**. Además, se muestra finalmente una tabla global con los resultados medios de cada conjunto de datos para todos los algoritmos. Esta información también se recoge en la última fila de las tablas de cada algoritmo.

Clasificador $1-$NN sin pesos

Nº	Clas	Red	Agr	T	Clas	Red	Agr	T	Clas	Red	Agr	T
1	72.88	0	36.44	0.88	83.09	0	41.54	2.50	94.54	0	47.27	1.93
2	75.43	0	37.71	0.81	84.28	0	42.14	0.92	88.18	0	44.09	1.90
3	78.94	0	39.47	0.80	94.28	0	47.14	0.62	90.90	0	45.45	1.91
4	82.45	0	41.22	0.80	85.71	0	42.85	0.61	96.36	0	48.18	1.91
5	68.42	0	34.21	0.80	88.57	0	44.28	0.61	90.00	0	45.00	1.95
$\bar{x}$	75.62	0	37.81	0.82	87.19	0	43.59	1.05	92.00	0	46.00	1.92

Algoritmo RELIEF

Nº	Clas	Red	Agr	T	Clas	Red	Agr	T	Clas	Red	Agr	T
1	71.18	20.96	46.07	4.65	84.50	2.94	43.72	13.55	94.54	2.50	48.52	10.30
2	77.19	24.19	50.69	4.19	84.28	2.94	43.61	3.82	89.09	5.00	47.04	10.48
3	71.92	30.64	51.28	4.52	94.28	2.94	48.61	3.67	92.72	5.00	48.86	10.49
4	77.19	72.58	74.88	4.05	88.57	2.94	45.75	3.61	97.27	17.5	57.38	9.97
5	63.15	32.25	47.70	4.17	90.00	2.94	46.47	3.60	92.72	5.00	48.86	11.70
$\bar{x}$	72.13	36.12	54.13	4.32	88.33	2.94	45.63	5.65	93.27	7.00	50.13	10.59

Algoritmo de búsqueda local

Nº	Clas	Red	Agr	T	Clas	Red	Agr	T	Clas	Red	Agr	T
1	71.18	83.87	77.52	10524	87.32	82.35	84.83	3634	91.81	82.50	87.15	19733
2	68.42	87.09	77.75	16717	85.71	85.29	85.50	6378	89.09	82.50	85.79	21246
3	80.70	80.64	80.67	17315	88.57	85.29	86.93	7142	90.00	82.50	86.25	17312
4	75.43	79.03	77.23	13554	84.28	88.23	86.26	5094	90.90	82.50	86.70	26170
5	57.89	85.48	71.68	15536	90.00	91.17	90.58	7746	88.18	87.50	87.84	26074
$\bar{x}$	70.72	83.22	76.97	14729	87.17	86.47	86.82	5999	90.00	83.50	86.75	22107

Resumen global

Nº	Clas	Red	Agr	T	Clas	Red	Agr	T	Clas	Red	Agr	T
1-NN	75.62	0	37.81	0.82	87.19	0	43.59	1.05	92.00	0	46.00	1.92
RELIEF	72.13	36.12	54.13	4.32	88.33	2.94	45.63	5.65	93.27	7.00	50.13	10.59
BL	70.72	83.22	76.97	14729	87.17	86.47	86.82	5999	90.00	83.50	86.75	22107

Análisis de resultados

En primer lugar, observamos que las tasas de clasificación obtenidas mediante el clasificador 1−NN sin pesos, aunque pueden llegar a ser altas según el *dataset*, no contribuyen en exceso al agregado total. Esto era de esperar, pues en este clasificador la simplicidad es siempre 0, la mínima posible.

Tomando como ejemplo el conjunto de datos **texture**, vemos que tiene una tasa de clasificación media de 92 % en este ejemplo, por lo que podemos concluir que las características medidas están bien elegidas. Sin embargo, el conjunto **colposcopy** tiene una tasa de clasificación media de solo 75 %, lo que nos hace pensar que podríamos mejorar seleccionando con más cuidado las características a medir.

En cuanto al algoritmo RELIEF, al tratarse de un algoritmo *greedy* es posible que no proporcione siempre la mejor solución. Al permitir que la tasa de reducción no sea 0 estamos aumentando

la simplicidad, y por tanto potencialmente aumentando el valor de la función objetivo. Vemos que efectivamente esto es lo que ocurre, pues consistentemente en todos los conjuntos de datos mejoramos dicho valor con respecto al clasificador sin pesos. Además, el tiempo de ejecución de este algoritmo sigue siendo despreciable, por lo que constituye un buen punto de partida para la comparación con otras metaheurísticas.

Por último, el algoritmo de búsqueda local es el más costoso en tiempo, llegando a tardar del orden de 1000 veces más por partición que los anteriores. Sin embargo, este algoritmo consigue tasas de reducción muy altas, es decir, aumenta mucho la simplicidad del clasificador. Al ser un algoritmo de búsqueda local típicamente caemos en óptimos locales, por lo que no conseguimos la solución óptima. Sin embargo, en muchas ocasiones llegamos a una solución *suficientemente buena*, como demuestran los resultados obtenidos.

En este último algoritmo conseguimos aumentar considerablemente el valor de la función objetivo en todos los casos, rondando el 80 %, a costa de un tiempo de ejecución mayor (aunque todavía viable). Aún queda ver cómo se comporta este algoritmo relativamente sencillos con otros que implementemos en el futuro.

Por último, cabe destacar que en cuanto a la tasa de clasificación, el algoritmo RELIEF es competitivo con el de la búsqueda local, llegando a ser mejor que este último en varias ocasiones. En cualquier caso, el algoritmo voraz no consigue llegar (ni se acerca) a las tasas de reducción que consigue la búsqueda local, que converge rápidamente a un óptimo local, aunque pueda tener períodos de estancamiento.