UNIVERSIDAD DE GRANADA

Práctica 2.b

Técnicas de búsqueda basadas en poblaciones para el problema del aprendizaje de pesos en características (APC).

Mario Román García

29 de abril de 2018

- DNI: 77145669N
- Email: mromang08@correo.ugr.es

- Grupo 1 (Lunes de 17:30 a 19:30)
- Algoritmos: AGE-{BLX,CA}, AGG-{BLX,CA}, AM-{10,0.1,mej}.
- Metaheurísticas, 2017-2018.

Índice

1. Formulación del problema

Trataremos un problema de **aprendizaje de pesos en características** (APC), consistente en la optimización de la simplicidad y precisión de un clasificador 1-NN; es decir, un clasificador que asigna a cada instancia la clase de la instancia más cercana a él, para una distancia euclídea modificada por un vector de pesos. Así, cada solución del problema vendrá dada por un vector de valores reales $w_i \in [0, 1]$ para $1 \leq i \leq n$, donde n es el número de características que tiene el problema. La bondad de un clasificador de este tipo sobre un conjunto de evaluación T vendrá determinada como

$$F(\{w_i\}) = \alpha \text{tasaClas}\,\{w_i\} + (1 - \alpha)\text{tasaRed}\,\{w_i\}\,.$$

En esta fórmula, *tasaClas* debe ser entendida como la **precisión** del algoritmo, midiendo el porcentaje de aciertos del clasificador en el conjunto de evaluación T. Por otro lado, *tasaRed* debe ser entendido como la **simplicidad** de la solución, que mide el número de pesos w_i que quedan por debajo de 0.2, y que consecuentemente no se tienen en cuenta al calcular las distancias. En nuestro caso tenemos determinado un valor de $\alpha = 0.5$ que equipara ambas métricas.

En resumen, el algoritmo deberá tomar un conjunto de datos y producir un clasificador 1-NN con unos pesos asignados a las características que sean sencillos, en el sentido de usar el mínimo número de características, y que sean precisos, en cuanto a su habilidad para clasificar instancias fuera del conjunto de entrenamiento.

2. Aplicación de algoritmos

2.1. Esquema de representación de soluciones

Como hemos tratado anteriormente, una solución del problema viene dada por un vector de pesos. Nuestra representación específica de una solución vendrá dada por una lista de valores reales.

$$\{w_i\}_{0 \leq i \leq n} = (w_1, w_2, \ldots, w_n)$$

Donde n será el número de características que tenga nuestro conjunto de datos y w_i. En nuestra implementación específica, los pesos vendrán dados por un vector contiguo en memoria de valores en coma flotante de precisión doble en 64 bits; para facilitar su tratamiento, en ciertos algoritmos se traducirán a listas enlazadas, pero su interpretación matemática será siempre constante y se describe explícitamente en la función objetivo.

2.2. Clasificador 1-NN, función objetivo

Se define la **distancia con pesos** $\{w_i\}$ entre dos vectores t y s como

$$\text{dist}(w, t, s) = \sum_{i=0}^{n} w_i (t_i - s_i)^2.$$

Y nuestro clasificador kNN para unos pesos $\{w_i\}$ consiste en devolver la clase del punto que minimiza la distancia. Es decir, es una implementación de un clasificador 1NN, que para cada instancia devuelve la clase de su vecino más cercano. En el caso de la función objetivo, al no tener un conjunto de test separado, usamos la técnica de *leave-one-out*.

Algorithm 1 Función objetivo (w : Pesos, T : Training)

1: $\text{Obj}(w, T) = \alpha \cdot \text{precision}(w, T) + (1 - \alpha)\text{simplicity}(w)$
2: $\text{TasaRed}(w) = \text{length}[x < 0.2 \mid x \in w]/\text{length } w$
3: $\text{TasaClas}(w, T) = \sum_{t \in T}(\text{knn}(w, T - t, t) == s.Clase)/\text{length } s$
4: $\text{knn}(w, T, t) = (\text{minimizador}_{t' \in T}(\text{dist}^2(\text{trunca}(w), t', t))).Clase$
5: $\text{trunca}(w) = \{0 \text{ si } w_i < 0.2; \quad w_i \text{ en otro caso } \mid w_i \in w\}$

En el código original esta función objetivo aparecerá implementada dos veces: una vez para el puntuador de los algoritmos y otra vez para la función objetivo. Esta duplicación tiene como ventaja que separa completamente las partes de evaluación del código de los algoritmos, reduciendo la posibilidad de error. Además,

- la implementación para evaluación es corta y es más fácil de verificar que está escrita correctamente;
- mientras que la implementación de la función objetivo está fuertemente optimizada, usando paralelismo, pero en caso de que tuviera cualquier error, eso no se vería reflejado en las puntuaciones de los algoritmos.

Es importante notar que la formulación original del problema no asume que todas las variables sean reales, y en el caso en el que son discretas asume una distancia de Hamming que simplemente indica si las dos características son exactamente iguales.

$$\text{dist}(a, b) = \delta_{ab} = \begin{cases} 1 & \text{si } a = b \\ 0 & \text{si } a \neq b \end{cases}$$

En los conjuntos de datos que trataremos en este análisis, tenemos de hecho que todas las características vienen dadas por reales, y nuestra implementación, aunque sería fácilmente extensible, no tratará el otro caso explícitamente.

2.3. Generación de soluciones aleatorias

La generación de una **solución aleatoria inicial** se realiza llamando repetidas veces a las librerías del lenguaje [Yor16]. Estas proporcionan funciones que nos permiten clacular listas potencialmente infinitas de números reales distribuidos uniformemente en el intervalo $[0, 1]$. Además, usamos replicamos el proceso aleatorio con mónadas para poder generar varios individuos cuidando que sean muestras independientes bajo el mismo generador aleatorio.

Algorithm 2 Solución inicial (t : Training)

1: solInicial(t) = tomaLos (nAttr(t)) primerosDe aleatorioUniforme$(0.0, 1.0)$
2: PoblInicial(t) = replica 30solInicial

3. Pseudocódigo de selección, cruce y mutación

3.1. Torneo binario

Un torneo binario es un proceso aleatorio que genera un individuo o desde una población de soluciones. La población de soluciones llega a este punto del algoritmo en una estructura ordenada por bondad de solución basada en árboles binarios balanceados por tamaño (véase [Lei02] y [Ada96]), por lo que simplemente debemos elegir dos índices y tomar el mayor. El conjunto de entrenamiento no se usa explícitamente en el algoritmo pero es una dependencia necesaria para haber calculado previamente la bondad de los elementos de la población.

Algorithm 3 Torneo binario (p : Población, T : Training)

1: x = randomEntre$(0, \text{size}(p) - 1)$
2: y = randomEntre$(0, \text{size}(p) - 1)$
3: torneo(p) = p[max x y]

En el código implementamos además variantes que realizan varios torneos binarios seguidos para usarlas directamente en los algoritmos.

3.2. Cruce aritmético

El cruce aritmético de dos soluciones es una operación componente a componente que devuelve el centro de gravedad n-dimensional de los dos padres. Es una operación determinista que sólo genera un hijo.

Algorithm 4 Cruce aritmético (a : Individuo, b : Individuo)

1: $media(x, y) = (x + y)/2.0$
2: $ca(a, b)$ = componenteAComponente media a b

3.3. Cruce BLX

El cruce BLX sí es no determinista y sí nos permitirá obtener dos hijos desde una sola pareja de padres. Nuestro código genera un solo hijo y simplemente repite (de nuevo usando mónadas) el procedimiento para asegurarse la independencia de los dos hijos.

En nuestro caso prepararemos el código para tratar uniformemente los dos cruces distintos, haciendo que cada uno devuelva una lista con uno y dos hijos y dejando que cada algoritmo trate estos dos casos.

Algorithm 5 Cruce BLX (a : Individuo, b : Individuo)

1: $blx2(a, b) =$ replica 2 blx(a, b)
2: $blx(a, b) =$ componenteAComponente blxComp(a, b)
3: $blxComp(x, y) =$ aleatorioUniformeEn intervalo(x, y)
4: $intervalo(x, y) = [0, 1] \cap [\text{mín}(x, y) - \alpha|x - y|, \text{máx}(x, y) + \alpha|x - y|]$

3.4. Mutación

La mutación, en su versión más común, procede directamente del operador de generación de vecinos de la búsqueda local. En ocasiones nos interesará mutar cada gen de un individuo con probabilidad 0.001.

Algorithm 6 Mutación (s : Solución)

1: $Muta(s) =$ truncaEntre0y1 map $(\lambda x.\text{if } rand() < 0.001 \text{ then } x + rand() \text{ else } x)$

Mientras que en otras ocasiones nos interesará aplicar un número fijo de mutaciones aleatorias sobre la población completa en lugar de generar un número aleatorio y comprobar si mutamos o no cada uno de los genes.

Algorithm 7 MutaPoblación (p : Población, n : Nº mutaciones)

1: $Muta(p) =$ replica n $MutaUnaVez(p[i], j),$ para $i = rand(), j = rand()$
2: $MutaUnaVezEn(s, j) =$ truncaEntre0y1 map $(\lambda(x, i).x + \delta_{ij}\varepsilon)$ (indexa s)

En esos casos controlaremos el número de mutaciones totales para que se correspondan a las que deberían producirse en caso de que usáramos la esperanza matemática para calcular el número de mutaciones.

4. Pseudocódigo del esquema de evolución y reemplazamiento

En general, nuestro algoritmo genético repite una subrutina denominada *paso generacional* que se corresponderá con un cruce y sustitución en el modelo estacionario y con el avance de una generación en la evolución generacional. El criterio de parada no se determina por el número de generación sino por el número de evaluaciones de la función objetivo, que se almacena con la estructura de datos del algoritmo.

De esta forma podemos modelar las variantes del algoritmo evolutivo uniformemente. Cuando trabajamos gestionando el generador aleatorio, podemos escribir en estilo imperativo (esto se conoce como un "bloque `do`").

4.1. Esquema de evolución estacionario

En el esquema de evolución estacionario, cada iteración hará dos evaluaciones para los dos nuevos hijos creados y que compiten con los individuos de la población anterior. Se

Algorithm 8 EsquemaEvolutivo (pasoEv : Subrutina)

1: $pobl \leftarrow$ poblacionAleatoria()
2: iteraMientras($evaluaciones > 15000$)
3: $\quad pasoEv(env)$

realizarán los torneos necesarios para generar 2 hijos (que serán 2 o 4 según el operador de cruce); luego se mutarán los hijos sin usar la esperanza matemática, usando el primero de los operadores de mutación que describimos. Finalmente, se incluyen los hijos en la población y se eliminan los peores; la estructura de datos ordenada se encargará de esto automáticamente.

Algorithm 9 EsquemaEstacionario (p : Población, cruza : Operador)

1: $padres \leftarrow$ torneosBinarios()
2: $hijos \leftarrow$ toma 2 de cruza(empareja(padres))
3: $mutados \leftarrow$ map $muta\ hijos$
4: $nuevaPopl \leftarrow$ insertaYReemplazaEn$(p, mutados)$
5: $contadorEvaluaciones \leftarrow +2$

Finalmente, el *emparejamiento* se hace tomando el primero con el segundo, tercero con el cuarto, y así sucesivamente. En el caso en el que se necesitaran más parejas, se vuelve a empezar con la lista en el segundo elemento para emparejar el segundo con el tercero y así sucesivamente.

4.2. Esquema de evolución generacional

Las iteraciones del algoritmo generacional generan 21 hijos nuevos; las parejas necesarias para ello (que serán el doble cuando estemos usando un operador de cruce de un solo hijo como el cruce aritmético) se obtendrán generando al principio una nueva población por torneo binario de 30 padres y luego usando el 70 % para generar hijos. Luego aplicaremos de nuevo mutación a toda la población usando esta vez la esperanza matemática y acabaremos incluyendo el mejor de la generación anterior en la nueva población.

Algorithm 10 EsquemaGeneracional (p : Población, cruza : Operador)

1: $mejor \leftarrow \max(p)$
2: $padres \leftarrow$ torneosBinarios()
3: $hijos \leftarrow$ cruza(empareja(70 % de los $padres$))
4: $npopl \leftarrow$ toma 30 de $padresNoCruzados + hijos + padresNoSubstituidos$
5: $npopl \leftarrow$ reemplaza($mejor$) \$ muta($npopl$)
6: $contadorEvaluaciones \leftarrow n(hijos) + n(padresMutados)$

4.3. Reemplazamiento

Explícitamente, el reemplazamiento en la población se realiza insertando y eliminando el mínimo en la estructura ordenada que describimos anteriormente.

Algorithm 11 Reemplazamiento (p : Población, h : individuo)

1: $reemplaza(p, h) = (borraMnimo \circ insertah)\ p$

En cuanto

5. Pseudocódigo de integración en algoritmos meméticos

6. Procedimiento considerado, manual de usuario

7. Experimentos y análisis de los resultados

Referencias

[Ada96] Stephen Adams. Implementing sets efficiently in a functional language. *Journal of Functional Programming*, 1996. https://groups.csail.mit.edu/mac/users/adams/BB/.

[Lei02] Daan Leijen. Data.set. *Hackage*, 2002. https://hackage.haskell.org/package/containers-0.5.11.0/docs/Data-Set.html.

[Yor16] Brent Yorgey. Control.monad.random.class. *Hackage*, 2016. https://hackage.haskell.org/package/MonadRandom-0.5.1/docs/Control-Monad-Random-Class.html.