UNIVERSIDAD
DE GRANADA

Práctica 1.b

Técnicas de búsqueda local y algoritmos greedy para el problema del aprendizaje de pesos en características.

Mario Román García

March 27, 2018

Contents

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. Cada solución del problema vendrá dada por un vector de pesos reales $w_i \in [0,1]$ para $1 \leq i \leq n$, donde n es el número de características que tiene el problema. Y su bondad sobre un conjunto de evaluación T viene 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.

2 Aplicación de algoritmos

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.

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

1: $\text{Obj}(w,T,S) = \alpha \cdot \text{precision}(w,T,S) + (1-\alpha)\text{simplicity}(w)$
2: $\text{TasaRed}(w) = \text{length}[x < 0.2 \mid x \in w]/\text{length } w$
3: $\text{TasaClas}(w,T,S) = \sum_{s \in S}(\text{knn}(w,T,s) == s.Clase)/\text{length } s$
4: $\text{knn}(w,T,s) = (\text{minimizador}_{t \in T}(\text{dist}^2(\text{trunca}(w),t,s))).Clase$
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.

3 Pseudocódigo de los algoritmos de búsqueda

Al usar pseudocódigo nótese que para reflejar más fielmente nuestra implementación, que intenta ser declarativa y basada en el paradigma de programación funcional, usaremos pseudocódigo basado en definiciones declarativas de funciones matemáticas.

3.1 1NN

La primera solución, que usaremos como referencia, es completamente trivial y se basa simplemente en usar directamente el clasificador 1NN con una distancia euclídea usual. Dentro de nuestra formulación del problema, esto equivale a una solución que simplemente devuelva en todos los casos un vector de pesos hecho constantemente de unos.

Algorithm 2 1NN (t : Training)

1: $1\text{NN}(t) = \text{replica } (\text{nAttr}(t)) \text{ veces } 1$

3.2 Relief

3.3 Búsqueda local

El operador de **generación de vecinos** será una mutación que tomará aleatoriamente un índice extraído de una distribución uniforme y un epsilon extraído de una distribución normal.

Algorithm 3 Mutación en búsqueda local (w : Pesos, i : Índice, ε : Epsilon)

1: $\text{Mutacion}(\varepsilon, i, w) = \text{map } (\lambda(x, i).x + \delta_{ij}\varepsilon) \text{ (indexa } w)$

El método de búsqueda consiste principalmente en dos funciones. Una de ellas busca una mejora local, aplicando repetidamente mutaciones con argumentos procedentes de una distribución normal y una permutación aleatoria y la otra ejecuta varias veces la búsqueda local. Lo usaremos como **exploración del entorno**.

Algorithm 4 Búsqueda Local (s : Semilla, t : Training)

1: $\text{busqueda}(s, t) = \text{repite busquedaLocal solucionAleatoria}$
$\text{busquedaLocal}(s, t, w) = \text{composicionDe}$

2:
$\$ \text{ busca } (\lambda w'.\text{objetivo}(w') < \text{objetivo}(w))$
$\$ \text{ entreLasPrimeras } \min(20\text{longitud}(w), 15000)$
$\$ \text{ map (Mutacion } w)$
$\$ \ \{(\sigma(n), \varepsilon) \mid \varepsilon \sim \mathcal{N}(0, 0.3), n \in \{0, .., \text{longitud}(w)\}\}$
$\$ \text{ para } \sigma \text{ permutacion aleatoria}$

La generación de la **solución aleatoria inicial** se hace directamente usando las librerías del lenguaje [Buc11], que proporcionan funciones para crear listas potencialmente

infinitas de reales distribuidos respecto a una distribución normal dada. Internamente, se usa el método de Box-Müller [BM+58] para generar los valores. De esa lista extraemos sólo los números necesarios para construir una solución.

Algorithm 5 Solución aleatoria (t : Training)

1: $\text{solucionAleatoria}(t) = \text{tomaLos } (\text{nAttr}(t)) \text{ primerosDe}$
$\text{random}\mathcal{N}(\mu = 0.5, \sigma = 0.5)$

4 Pseudocódigo de los algoritmos de comparación

Para comparar los algoritmos entre sí usaremos validación cruzada en 5 partes. Tendremos un programa que parte los conjuntos de datos en cinco subconjuntos balanceados.

Algorithm 6 Partición en 5 (t : Training)

1: $\text{5split}(t) = \text{une partes1 partes2}$
2: $\text{deClase1} = \text{filtra } (.\text{Clase} \equiv 1)$
3: $\text{deClase2} = \text{filtra } (.\text{Clase} \equiv 2)$
4: $\text{partes1} = \text{parteEnTrozosDe } \lceil \text{longitud}(t)/5 \rceil \text{ deClase1}$
5: $\text{partes2} = \text{parteEnTrozosDe } \lceil \text{longitud}(t)/5 \rceil \text{ deClase2}$

5 Procedimiento considerado, manual de usuario

El código de esta práctica está escrito en el lenguaje de programación **Haskell** [P+03]. El requisito fundamental para compilarlo es tener instalada **stack**, la herramienta de compilación de Haskell; además de ella, usa **GNU make** [Smi92] para hacer el proceso de validación y generación de soluciones completamente reproducible.

El archivo `makefile` es el encargado de ejecutar los programas de manera acorde para conseguir los datos finales. En él se encuentra una semilla de aleatoriedad general (`$SEED`) que es la que se envía a los distintos algoritmos. Este mismo archivo contiene ejemplos de llamada a los ejecutables de la práctica y permite crear las soluciones de forma reproducible.

Tenemos varios ejecutables completamente indepedientes y que pueden usarse con cualquier instancia del problema que esté en formato `.arff` o con cualquier solución dada por una cabecera `@time ...` midiendo los segundos que ha tardado y una lista de valores para los pesos en formato CSV:

- `bin/fivefold`, que toma como entrada un archivo `.arff` y crea 5 archivos entre los que reparte sus instancias, de forma que queden balanceadas;
- `bin/scorer`, evalúa usando la función objetivo descrita anteriormente, recibirá el conjunto de training por la entrada estándar y tendrá como argumentos de línea de comandos al conjunto de test y la solución;

- `bin/Onenn`, implementación trivial de la solución que devuelve todos los pesos a 1;
- `bin/Relief`, implementación del algoritmo greedy Relief; y
- `bin/LocalSearch`, implementación de la búsqueda local.

Todas las implementaciones reciben como argumento de línea de comandos una semilla aleatoria y leen por la entrada estándar un conjunto de entrenamiento; acabarán devolviendo una solución por salida estándar.

El uso común de los programas será a traves del comando `make`. Normalmente, querremos generar un reporte de la bondad de un algoritmo determinado usando validación cruzada en cinco partes. Por ejemplo, supongamos que queremos generar un reporte de la bondad del algoritmo de búsqueda local sobre el conjunto de datos `parkinsons.arff`. Para ello lanzaremos los siguientes comandos.

```
make data/parkinsons.arff.LocalSearch.report
cat data/parkinsons.arff.LocalSearch.report
```

La ventaja de este enfoque es que permite la reutilización de los resultados ya calculados (que normalmente serán costosos en tiempo) automáticamente.

6 Experimentos y análisis de resultados

El único parámetro que nuestros algoritmos usarán globalmente es la semilla de generación aleatoria. En los experimentos que describimos aquí usamos siempre la semilla $s = 42$. Además de ella, existen parámetros que vienen fijados por los requisitos de la práctica: la desviación típica usada en la generación de vecinos de la búsqueda local se fija siempre en $\sigma = 0.3$, y la distribución de importancia entre precisión y simplicidad se fija siempre en $\alpha = 0.5$.

6.1 1-NN

Analizando el 1-NN.

Table 1: Algoritmo 1-NN en el problema del APC

	Ozone				Parkinsons				Spectf			
	%clas	%red	Agr.	T	%clas	%red	Agr.	T	%clas	%red	Agr.	T
Partición 1	*				*				*			
Partición 2	*				*				*			
Partición 3	*				*				*			
Partición 4	*				*				*			

6.2 Relief

6.3 Búsqueda local

6.4 Conclusiones

Encontramos por tanto una mejora significativa al usar métodos de búsqueda local frente a soluciones triviales como la proporcionada por el 1NN; y frente a soluciones basadas en algoritmos voraces, que ofrecen comparativamente una solución muy pobre especialmente en cuanto a simplicidad.

References

[BM+58] George EP Box, Mervin E Muller, et al. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2):610–611, 1958.

[Buc11] Bjorn Buckwalter. Data.random.normal. *Hackage*, 2011. https://hackage.haskell.org/package/normaldistribution-1.1.0.3/docs/Data-Random-Normal.html.

[P+03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Smi92] Paul Smith. Gnu make. *GNU Operating system*, 1992. https://www.gnu.org/software/make/.