Aprendizaje lógico inductivo. Una implementación de FOIL.

César Antonio Enrique Ramírez

Universidad de Huelva

Resumen FOIL es un sistema de aprendizage lógico de primer orden que usa información en una base de conocimiento, o colección de relaciones para construir teorias expresadas en Prolog. Esta memoria proporciona las principales ideas y tecnicas usadas para la implementacion de una versión sencilla de FOIL en haskell, así como los problemas encontrados por no incluir toda la funcionalidad descrita por J. R. Quinlan y R. M. Cameron-Jones en *Induction of Logic Programs: Foil and Related Systems*.

1. Introducción

La programación lógica inductiva se basa en la lógica de primer orden para desarrollar teorías. Los datos de entrenamiento constan de una relación objetivo, definida extensionalmente con un nombre y una tupla de terminos constantes, y un conjunto de infomación de fondo definido a su vez como relaciones. En el Desarrollo original de FOIL se posibilitaba que esta información de fondo estuviera definida tanto extensionalmente como intencionalmente, es decir, por relaciones contantes cubriendo todos los casos positivos de esa relación, o mediante otras relaciones, respectivamente. En este trabajo solo se permite la definición de dichas relaciones extensionalmente. Al igual que en otros sitemas basados en arboles de decisión, y pares atributo-valor, los ejemplos que pertenecen y los que no pertenecen a la relación objetivo los denominamos $\oplus$ y $\ominus$ respectivamente.

Decimos que una teoría completa cubre a una tupla si la evaluación de al menos una de sus reglas es satisfactoria. El objetivo, por tanto, de los sistemas de apredizade basados en lógica de primer orden es construir teorías que cubren a todos los ejemplos $\oplus$ y excluyen a todos los ejemplos $\ominus$.

FOIL es un sistema de aprendizaje lógico inductivo que presenta una estrategia *separate-and-conquer*. Así mismo, el método usado para encontrar una regla adecuada es *top-down*, es decir, partiendo de una regla sin cuerpo (que cubre a todos los ejemplos), va añadiendo cláusulas que van restringiendo la cobertura hasta que no cubre ningún ejemplo $\ominus$.

2. Planteamiento teórico de FOIL

Como hemos mencionado antes, FOIL utiliza información sobre relaciones como entrada para generar una teoría, compuesta de reglas, que describan la relación objetivo. Podemos plantear los ejemplos como *n-tuplas* representando las variables de la relación objetivo. Un ejemplo de relación objetivo sería la relación *componentes*:

⟨[1,1],1,[1]⟩	⟨[2,1],2,[1]⟩	⟨[3,1],3,[1]⟩
⟨[1,2],1,[2]⟩	⟨[2,2],2,[2]⟩	⟨[3,2],3,[2]⟩
⟨[1,3],1,[3]⟩	⟨[2,3],2,[3]⟩	⟨[3,3],3,[3]⟩

Cuadro 1: Ejemplo de relación *componentes* de listas de tamaño 2

donde se establece que una lista [1,2] está compuesta por la cabeza 1 y la cola [2].

Todos los ejemplos descritos pertenecen a los ejemplos $\oplus$. Los ejemplos $\ominus$ se pueden determinar usando la asumpción de mundo cerrado. Por lo tanto, todos los ejemplos que no aparezcan explícitamente mencionados, como por ejemplo $\langle$[2,2],1,[2]$\rangle$, son ejemplos $\ominus$.

2.1. Algoritmo

```
1   teoria := null
    ejemplos+ := todos los ejemplos ⊕
3
    mientras ejemplos+ no sea vacio
5       regla := R(X, Y, ...) :-
        mientras regla cubra ejemplos ⊖
7           calcula el mejor literal L
            añade el literal L a regla
9       elimina los ejemplos ⊕ cubiertos por regla
        añade regla a teoria
```

Figura 1: Pseudo-código de FOIL

Como podemos observar el algoritmo original utiliza un método iterativo, en el que va especializando progresivamente una regla hasta que no cubre ningun ejemplo $\ominus$. En muestro caso, la implementación varía ligeramente utilizando una estrategia recursiva para la implementación. En cualquier caso, son equivalentes.

2.2. Selección de literales

Los diferentes literales permitidos en el cuerpo de una regla son de una de las dos formas siguientes:

- $L(X_0, X_1, ..., X_n)$ donde L es una relación y los X_i denotan variables que aparecen antes en la regla o nuevas variables.
- $X_i = x_j$ y $X_i \neq X_j$ donde X_i y X_j no son nuevas variables.

Además, los literales del tipo $L(...)$ tienen que tener al menos una variable usada antes en la regla. Según el artículo original, dicha variable podría ser una variable usada anteriormente en el cuerpo de la regla, pero dada la sencillez de nuestra implementación, los literales quedan restringidos a usar al menos una variable que aparezca en la cabeza de la regla. Esto supone una solución fácil, aunque mala, a alguno problemas provocados por no implementar técnicas de *pruning* o detección de reglas excesivamente complejas[1]

La selección del mejor literal se realiza haciendo uso de una heurística. Siendo el número de ejemplos $\oplus$ y ejemplos $\ominus$ que cubre una regla parcial $n^{\oplus}$ y $n^{\ominus}$ respectivamente, la información que proporciona una sustitución positiva es

$$I(n^{\oplus}, n^{\ominus}) = -log_2 \frac{n^{\oplus}}{(n^{\oplus} + n^{\ominus})} \tag{1}$$

Por lo tanto, suponiendo que k ejemplos $\oplus$ no son excluidos al incluir un nuevo literal a una regla parcial, y que el número de ejmplos que cubre la nueva regla son $m^{\oplus}$ y $o^{\ominus}$ respectivamente. La ganancia total obtenida al añadir ese literal es

$$k \times (I(n^{\oplus}, n^{\ominus}) - I(m^{\oplus}, m^{\ominus})) \tag{2}$$

[1] Hablaremos sobre esto en el apartado **??**

$$k \times (log_2 \frac{m^{\oplus}}{(m^{\oplus} + m^{\ominus})} - log_2 \frac{n^{\oplus}}{(n^{\oplus} + n^{\ominus})}) \quad (3)$$

3. Implemetación en Haskell

Con la base de cómo funciona FOIL, podemos pasar a ver con un poco más de detalle las caracteristicas de nuestra implementación[2].

Como hemos comentado en el apartado anterior, la implementación que presentamos está hecha en Haskell[3], un lenguaje funcional, por lo que el uso de estructuras iterativas propias de lenguajes imperativos como los bucles *while* no es posible. En vez de eso, se utiliza la recursividad.

3.1. Tipos de datos

Antes de entrar a explicar algunas consas concretas de la implementación, es conveniente exponer rápidamente los tipos de datos definidos, para poder emplearlos a la hora de describir partes del algoritmo más adelante.

- **Variable** → puede representar tanto a variables libres, como X_i, como valores concretos del dominio.
- **Literal** → representa todos los tipos de literales aceptados en el cuerpo de una regla, como son los del tipo $L(...)$, $X_i = X_j$ y $X_i \neq X_j$.
- **Rule** → representa una regla. Está implementado usando un Literal para la cabeza de la regla (con la restrincción de que no puede ser un Literal de la forma $X_i = X_j$ y $X_i \neq X_j$) y una lista de Literales para el cuerpo.
- **BC** → representa la base de conocimiento, y está implementado como una lista de Literales
- **Ejemplo** → está implementado como una lista de Variables (en vez de como una tupla, tal como se describe en el apartado 2)

3.2. Ejemplo de la estructura típica del algoritmo en Haskell

Esta forma de describir los algoritmos propicia a que ciertas cosas que realmente representan lo mismo, a primera vista puedan parecer totalmente distintas, por poner un ejemplo esto se ve claramente en la definición de la función

```
cubre :: BC -> BC -> [Variable] -> Rule -> Ejemplo -> Bool
cubre bc bcEj dom r ej = or . map (evalRule bc bcEj r dom) $ buildrule r dom ej
```

Figura 2: Función *cubre*, determina si una regla cubre a un ejemlo.

en la que podemos ver como el funcionamiento de la función se expresa en término de lo que hace, no de cómo lo hace. Tenemos que **bc** es la base de conocimiento, **bcEj** es una base de conocimiento (porque usa el mismo tipo de dato) pero que sólo contiene los ejemplos positivos, **dom** son las constantes del dominio, **r** es la regla a evaluar y **ej** es el ejemplo con el que evaluar la regla. Y según la función, se hace una llamada a *buildrule*, que toma como parámetos la regla, las constantes

[2] Aunque incluiremos algunos trozos de código relevantes para el concepto que estemos tratando, esta sección no pretende documentar el código. Para esa función se adjunta a la memoria la documentación del código elaborada con la herramienta Haddock - `https://www.haskell.org/haddock/doc/html/`.

[3] `https://www.haskell.org/`

del dominio y el ejemplo. Esta función devuelve una lista de reglas, que representa todas las posibles reglas que se pueden construir realizando la sustitución de las variables de la cabeza por el ejemplo, y asignado a las variables libres todas las posibles combinaciones de las constantes del dominio.

Teniendo una lista de reglas, le aplicamos con un *map* la función *evalRule*, que toma como parámetros la base de conocimiento, la base de conocimiento de los ejemplos, la regla, el dominio, y una regla ya construida. Esta función determina si la regla contruida con unos valores determinados es cierta o no.

Por lo tanto, este trozo nos devolverá una lista de *True* o *False*, representando cada una de las posibles sustituciones es verdadera o no. Por lo tanto, y si necesitamos que al menos haya una sustitución que haga cierta la regla, solo nos queda hacer un *or* de todos los valores de la lista.

Si el resultado es *True*, significa que la regla en cuestión cubre el ejemplo dado. Esto contrasta directamente con la forma de describir este proceso de los lenguajes imperativos, mediante bucles y valores mutables. Haciendo uso de funciones como *map* encapsulamos los procesos recursivos, y generalizamos, dando un significado más abstracto al algoritmo y más cercano a la forma en la que los pensamos, resultando así en un algoritmo descrito con un leguaje de más alto nivel.

3.3. Predicados recursivos

La capacidad de FOIL de descirbir predicados recursivos viene descrita por J. R. Quinlan y R. M. Cameron-Jones en el artículo original, dando además una serie de restricciones para garantizar que no se generen predicados que provoquen recursividad infinita[4]. Esta implementación de FOIL es capaz de describir predicados recursivos, sin embargo, no incluye dichas restricciones, provocando eso que los resultados obtenidos pueden no terminar si fuesen a ejecutarse como código Prolog.

La esencia de la capacidad de generar predicados recursivos reside en la evaluación de las reglas, en la parte del algoritmo que determina si una regla con un ejemplo concreto es cierta o no. Esa parte del algoritmo la realiza la función que hemos visto arriba *evalRule*.

```
    evalRule :: BC -> BC -> Rule -> [Variable] -> Rule -> Bool
2   evalRule _ _ _ _ (R _ []) = True
    evalRule bc bcEj r dom (R h (t:ts))
4     | h `litEq` t     = (evalLit bcEj t) && (evalRule bc bcEj r dom (R h ts))
      | otherwise       = (evalLit bc t)   && (evalRule bc bcEj r dom (R h ts))
```

Figura 3: Implementación de la función *evalRule*, que determina si una regla con un ejemplo ya sustituído es cierta o no.

Podemos ver que la forma de evaluar los literales recursivos es mediante el uso de *bcEj*, comprobando si existe un ejemplo con esos valores. Podemos observar también que la función *evalRule* está definida en función de sí misma.

4. Pruebas

Debido a la naturaleza de los datos con los que trata FOIL, no podemos hacer pruebas generando datos aleatoriamente, por lo que las pruebas tienen que ser cuidadosamente diseñadas. Para intentar suplir esta cuestión hemos hecho 2 tipos de pruebas diferentes: pruebas con 3 datasets sencillos diseñados a mano, y pruebas usando el criterio de validación *Holdout*.

[4] Hablaremos más extensamente sobre esta cuestión en el apartado **??**

4.1. Pruebas con 3 datasets sencillos

Los tres datasets están basados en la relación *nieta(X, Y)* usando las relaciones *padre(X, Y)* y *mujer(X)*.

Con el primer dataset FOIL genera la regla:

$$nieta(X, Y) : -mujer(Y), padre(Y, Z0), padre(Z0, X). \tag{4}$$

que es cierta en el 100 % de los casos. Es con el segundo dataset donde se empiezan a ver algunos problemas de esta implementación. En concreto problemas de sobre ajuste. Foil considera que todas las *nietas* que existen están descritas por los ejemplos, por lo tanto, cuando analiza la base de conocimiento y encuentra una estructura familiar en la que la única nieta que existe tiene tanto un abuelo como una abuela, la regla que produce

$$nieta(X, Y) : -mujer(Y), nieta(Z0, Y), X \neq Y, mujer(X). \tag{5}$$

$$nieta(X, Y) : -mujer(Y), padre(Y, Z0), padre(Z0, X). \tag{6}$$

describe la relación nieta, como X e Y tal que Y es mujer, X es mujer, son distintas personas y además Y es nieta de otra persona Z0, o bien, de forma que Y sea mujer, haya un Z0 que es padre de Y, y X es padre de Z0.

Esto, auque es cierto para el dataset en cuestión, no es cierto en general, por lo que no debería ser una solución aceptable. Nuevamente, esto es debido a la falta de restricciones y mejoras descritas en la publicación original de FOIL, pero que no se han implementado en esta versión.

4.2. Puebas usando *Holdout* en el dataset de las relaciones familiares

5. Problemas encontrados y posibles soluciones

A lo largo del desarrollo del algoritmo y durante la fase de pruebas nos hemos encontrado con una serie de comportamientos que no son los más deseables. Pasamos a describir dichos problemas y sus posibles soluciones.

5.1. Reglas recursivas que no terminan

Uno de los problemas encontrados es que las reglas recursivas generadas por esta implementación de FOIL no tienen la ganrantía de generar predicados que terminen, si se ejecutan como programa Prolog. Esto es debido a que no implementamos ninguna forma de garantizar que cada vez que se hace una llamada recursiva el tamaño del problema se reduce.

Para solucionar esto en el artículo original en el que se describe FOIL se establece una técnica que hace uso del ordenamiento parcial de las constantes del dominio.

Los predicados mutuamente recursivos no se tiene en cuenta, siendo las reglas

$$marido(X, Y) : -mujer(Y, X). \tag{7}$$

$$mujer(X, Y) : -marido(Y, X). \tag{8}$$

definiciones válidas.

5.2. Mínimos locales

En algunos casos, como con el dataset 3, la generación de las reglas llega a un punto en el que no excluye a todos los ejemplos $\ominus$ a menos que incluya un literal que excluye también a todos los ejemplos $\oplus$. Esto provoca que el cálculo de la ganancia de 0 para todos los posibles literales y, por tanto, no se elimine ningún ejemplo $\oplus$ para el cálculo de la siguiente regla. Esto provoca un bucle infinito del que el algoritmo no tiene forma de salir.

Para solucionar este tipo de situaciones en el artículo original se introduce en casos excepcionales el uso de *backtracking*. Esto se hace estableciendo una *etiqueta* en el estado de la regla cuando el literal seleccionado para ser el siguiente añadido, no es mucho mejor que el resto, de forma que si después de varias iteraciones después se llega a una situación no deseable, se vueve a como estaba en la etiqueta y se contiúa desde ahí con otro literal diferente.

5.3. Generación de reglas muy complejas

Otro problema que muestra esta implementación es que con ciertos datasets, genera definiciones de la relación objetivo demasiado complejas, en algunos casos con más coste de codificación de la regla en sí, que si se codificaran mediante una descripción extensional todos los ejemplos positivos que cubre esa regla.

Esto se soluciona aplicando por una parte algún tipo de chequeo a las reglas generadas, para determinar si pasa de una complejidad establecida, y por otra partde aplicando lo que en el artículo original llaman *pruning*.

El *pruning* no es más que podar las reglas una vez generadas para simplificarlas sin que éstas pierdan significado. Se basa en que cláusulas añadidas después pueden estar cubriendo y excluyendo a los mismo ejemplos $\oplus$ y $\ominus$ respectivamente, que otra añadida antes, por lo que puede ser productivo eliminar la primera.

Si aparece un literal que sólo contiene variables de la cabeza de la regla, se descartan todos los literales que contienen variables libre y se continúa desde ahí.

Esto también soluciona algunos problemas de sobreajuste.

6. Caso de investigación: generación de gramática para descirbir canciones que suenen bien