一个不受常量序限制的归纳逻辑程序设计算法*

张润琦[1] 陈小平[2] 刘贵全[2]

[1](中国科技大学少年班系 合肥 230026)

[2](中国科技大学计算机科学系 合肥 230027)

E-mail: gqliu@mail.ustc.edu.cn

摘要 文章分析了 FOIL(first-order inductive learner)递归谓词学习算法理论上的不足以及由此导致的应用范围的局限,并通过两个例子给予详细说明.为了克服这一缺陷,文章引入了反映递归规则集 R 与实例空间 E 本质关系的实例图 $H_{(R,E)}$ 和实例序的概念,奠定了算法的理论基础.在此基础上,给出了基于实例图的 FOILPlus 算法.算法通过对悬例、悬弧的操作把握住实例序,自然而然地防止了病态递归规则的产生,从而保证了 FOILPlus 可以不受常量序限制地完成学习任务;同时,算法的时空复杂度较之 FOIL 算法没有增加.FOILPlus 算法已经编程实现,并用它尝试了两个 FOIL 学习失败的递归任务,都获得了成功.

关键词 归纳逻辑程序设计,FOIL(first-order inductive learner),递归,实例图,实例序,悬例,悬弧,FOILPlus.

中图法分类号 TP18

归纳逻辑程序设计(inductive logic programming,简称 ILP)是机器学习的一条重要途径.不仅已取得重要的理论研究成果,而且在不同领域获得了一系列成功的应用[1~4].一个 ILP 学习任务可描述如下[4]:

给定 背景理论 B 和靶谓词的示例集 $E=E^+\cup E^-$,满足 $B\not\models E^+$ 且 $B\wedge E^+\not\models\square$,

求 靶谓词的内涵定义(逻辑程序)H 使 $H\wedge B\models E^+$ 且 $H\wedge B\wedge E^-\not\models\square$.

由上述描述可知,ILP 算法的一个必要的基本功能是实例的可推出性(即覆盖关系)的检验.在一般条件下,当 H 包含递归子句时,这种检验过程可能不终止.为了使 ILP 技术能在 KDD 等应用中直接为终端用户服务,需要提供某种一般机制以检验上述过程的有效性,特别是有限终止性.为此,必须使 H 不含"病态递归"[5].现有的各种方法都基于文字的排序.相对而言,FOIL(first-order inductive learner)[5]采用的"常量序"方法最为先进、完善,这使 FOIL 系统的实用化程度最高,而且使它成为目前唯一能够学习出 Ackermann 函数的 ILP 系统[5].

然而,"常量序"方法也有不利的一面,它排斥了一大类本来可解的、常见的学习任务,使 FOIL 的应用范围受到很大限制.本文提出了一个新的 ILP 算法——FOILPlus,除了规则搜索(生成)的启发式机制的不完备(这是目前无法避免的)和覆盖性检验过程实现的不完备(一阶 ILP 学习系统都存在此问题)之外,FOILPlus 本身对可学到的规则集没有任何限制.

1 FOIL 的递归谓词学习算法

FOIL[5,6]在一个常量集合有限的域内学习目标关系(或称靶谓词)的内涵定义.常量可以是不同的类型,例如元素或元素的表,也可以是离散的或连续的.关系中的变量只能属于某个类型,关系本身外延定义为实例的集

* 本文研究得到国家自然科学基金资助.作者张润琦,1975 年生,学士,主要研究领域为算法机器学习,ILP,多谓词学习.陈小平,1955 年生,博士,副教授,主要研究领域为机器学习,MAS,AI 逻辑.刘贵全,1970 年生,博士,主要研究领域为机器学习,多 Agent 系统,AI 基础.

本文通讯联系人:刘贵全,合肥 230026,中国科技大学计算机科学系

本文 1998-05-18 收到原稿,1998-09-11 收到修改稿

合.正的实例(用⊕表示)是那些关系为真的实例,负的实例(用⊖表示)是那些关系为假的实例.FOIL 通过对目标或背景关系的外延定义(即实例)学习目标关系的没有函数的 Horn 子句定义.训练数据包含了一个目标关系和一组背景关系(均为外延定义).学习目标是建立一个逻辑程序(或称规则集),该逻辑程序给出了用目标关系和背景关系表达的目标关系的内涵定义.该定义允许递归和有限的量化,二者有利于处理难于用属性值表示法描述的结构对象.

FOIL 将所有训练实例放在一起加以考虑,每次选择一个子句,该子句覆盖一部分正例,不覆盖任何反例.反复执行此操作,直到任何正例都至少被一个子句所覆盖为止.FOIL 是一个 top-down 系统,即由一般到特殊.当它产生一个子句时,总是开始时的子句体为恒真,每次加入一个文字,使之特殊化,直到符合要求为止.

为了保证不会出现循环递归,FOIL 在递归文字(recursive literal)[1,2]的选取上借助"常量序"进行了限制.

1.1　常量排序

对于每一种类型 T 的常量,如果用户没有指定其顺序的话,FOIL 就试图发现一个似乎合理的顺序.FOIL 检查每个关系 R 中的每对类型为 T 的变元 a_i,a_j(a_i,a_j 分别指 R 中第 i 和第 j 个变元),看是否对于所有定义 R 的实例,a_i,a_j 之间都存在一致的偏序,记作 $a_i<a_j$(因为不可能分辨 $a_i<a_j$ 还是 $a_i>a_j$).如果这种偏序存在,则 R 的每一个实例将给出常量序 $c_i<c_j$(其中 c_i,c_j 分别表示实例中第 i 和第 j 个位置的常量).如果出现某个常量 c_k,有 $c_k<c_k$,则此偏序不存在.

1.2　递归文字排序

类型 T 的常量的排序可能暗示了在一个部分子句(patial clause)中的类型 T 的变量 v_i,v_j 之间可能存在偏序.如果该部分子句的每个例化赋给 v_i,v_j 的值 c_i,c_j 总有 $c_i<c_j$ 成立,则 $v_i<v_j$ 成立.基于此,如果对于所有头为 $R(v_1,v_2,\ldots,v_n)$,体文字为 $R(w_1,w_2,\ldots,w_n)$ 的子句,总有 $R(w_1,w_2,\ldots,w_n)<R(v_1,v_2,\ldots,v_n)$,则递归终止将被确保.定义基于常量序的文字序为

$$\begin{aligned}&R(w_1,w_2,\ldots,w_n)<R(v_1,v_2,\ldots,v_n)\quad \text{如果}\\&w_{i_1}<_{i_1}v_{i_1}\text{或}\\&w_{i_1}=v_{i_1}\text{且 }w_{i_2}<_{i_2}v_{i_2}\quad \text{或}\\&\qquad\vdots\\&w_{i_1}=v_{i_1}\text{且 }w_{i_2}=v_{i_2}\text{且}\cdots w_{i_{n-1}}=v_{i_{n-1}}\text{且 }w_{i_n}<_{i_n}v_{i_n}\end{aligned}\tag{1}$$

其中 $i_1,i_2,\ldots,i_n$ 是 $1,2,\ldots,n$ 的某个全排列,$<_k$(其中 $k\in\{1,\ldots,n\}$)表示"$<$",如果常量真实的顺序已知的话;否则表示$<$或$>$二者之一.每当一个递归文字被考虑,FOIL 总试图构造一个文字序,使之能够同时满足本文字和当前规则集中所有其他文字.如果这样的文字序不存在,则该递归文字从规则空间中排除.

2　FOIL 递归谓词学习算法的缺陷及其分析

如前所述,FOIL 递归谓词学习算法的核心思想是,通过用户定义或机器搜索人为地给常量排序,在此基础上,从规则空间排除那些不存在文字序的递归文字,以保证递归最终会终结于初始条件.FOIL 不能对常量排序,则它将把所有递归文字从规则空间中排除;FOIL 对常量错误排序,则它也会把一部分必须的递归文字从规则空间中排除.新的规则空间不再包含任何满足要求的递归规则集,导致 FOIL 学习失败.所以,正确的常量排序是 FOIL 学习出正确的递归规则集的必要条件.

实际的学习任务其本身的特点或个别噪声数据的影响可能会使 FOIL 找不到或找错常量的顺序,从而学习不出或学习不全正确的递归规则(注意,不是因为数据过于一般化或噪声太多等缘故).

例 1:设 $(a_1,b_1,\ldots,h_1)$ 和 $(a_2,b_2,\ldots,h_2)$ 为两个八边形,背景关系 *succ* 和 *next* 分别为其点的顺序关系,目标关系 *map* 为两者之间点的映射关系($*g_1$ 表示常量 g_1 是理论常量——可以用于子句体的常量;*map*(X,Y) 表示关系 *map* 有两个变量,第 1 个类型为 X,第 2 个类型为 Y).

常量类型:X: $a_1,b_1,c_1,d_1,e_1,f_1,*g_1,h_1$

Y: $a_2,b_2,c_2,d_2,e_2,f_2,*g_2,h_2$

背景关系：$succ(X,X)\oplus$：$(a_1,b_1),(b_1,c_1),(c_1,d_1),(d_1,e_1),(e_1,f_1),(f_1,g_1),(g_1,h_1),(h_1,a_1)$

$next(Y,Y)\oplus$：$(a_2,b_2),(b_2,c_2),(c_2,d_2),(d_2,e_2),(e_2,f_2),(f_2,g_2),(g_2,h_2),(h_2,a_2)$

目标关系：$map(X,Y)\oplus$：$(a_1,a_2),(b_1,b_2),(c_1,c_2),(d_1,d_2),(e_1,e_2),(f_1,f_2),(g_1,g_2),(h_1,h_2)$

所有关系的反例均由封闭世界假定(closed world assumption，简称 CWA)[7]给定.

显然，规则空间内有满足要求的规则集合

$$\begin{cases} map(g_1,g_2)\leftarrow \\ map(A,B)\leftarrow A\neq g_1,succ(C,A),next(D,B),map(C,D) \end{cases}' \tag{2}$$

但经实际测试(本文中所有对 FOIL 的测试都是使用目前最高版本 FOIL6.4)，FOIL 无法学习出. 因为背景关系，*succ* 和 *next* 都分别赋予类型 X,Y 的常量以明显的环，而目标关系 *map* 更没有蕴涵任何常量有序的信息，所以，FOIL 认为常量无序，虽然常量序 $g_1\prec h_1\prec a_1\prec b_1\prec c_1\prec d_1\prec e_1\prec f_1$ 和 $g_2\prec h_2\prec a_2\prec b_2\prec c_2\prec d_2\prec e_2\prec f_2$ 确实存在. 常量无法排序使得 FOIL 将所有递归文字从候选文字中排除，导致学习失败. 事实上，若用户定义这两个常量顺序中的任意一个，FOIL 都能学习出规则集(2). 作为比较，如果将 $(f_1,g_1),(g_1,h_1),(f_2,g_2),(g_2,h_2)$ 之一改为反例，则 FOIL 因为能将类型 X 或 Y 的常量正确排序，从而学习出规则集(2). 所以，实际数据中的有序性过于隐蔽或存在噪声数据(可以认为 $(f_1,g_1),(g_1,h_1),(f_2,g_2),(g_2,h_2)$ 之中有一个是噪声数据)，都可能导致 FOIL 常量排序失误，从而不能产生正确的递归规则集.

例 2：考虑任何自然数的二进制表示中 1 的个数的递归谓词的学习.

常量类型：N：$*0,1,2,\ldots,100$

背景关系：$odd(N)\quad\oplus$：$(1),(3),(5),\ldots,(99)$

$even(N)\quad\oplus$：$(0),(2),(4),\ldots,(100)$

$half(N,N)\oplus$：$(0,0),(1,0),(2,1),(3,1),(4,2),(5,2),(6,3),(7,3),\ldots,(99,49),(100,50)$

$succ(N,N)\oplus$：$(0,1),(1,2),(2,3),\ldots,(98,99),(99,100)$

目标关系：$binary(N,N)\quad\oplus$：$(0,0),(1,1),(2,1),(3,2),(4,1),(5,2),\ldots,(99,4),(100,3)$

所有关系的反例均由 CWA 给定.

规则空间内有满足要求的规则集合如下所示，

$$\begin{cases} binary(0,0)\leftarrow \\ binary(A,B)\leftarrow even(A),half(A,C),binary(C,B) \\ binary(A,B)\leftarrow odd(A),half(A,C),binary(C,D),succ(D,B) \end{cases}. \tag{3}$$

根据测试结果，FOIL 没能学出 3 个规则中的任何一个. 虽然 FOIL 将常量排序为 $100,99,\ldots,1,0$，但事实表明，这个常量序对学习没有任何帮助. 如果勉强说本例中的常量存在顺序的话，那也不是全序. 若将常量按自然数顺序由小到大排列为 $0,1,\ldots,99,100$，将其看做堆，则堆中每个数的双亲结点是它在该偏序中的前序元.

通过上面两个例子，我们很容易看出 FOIL 对常量排序以帮助递归文字的选取这一方法的局限性：对于常量无序、常量序为偏序、常量序隐蔽或噪声等导致 FOIL 不能学习出正确的常量序的递归学习任务，FOIL 都是不能胜任的. 而且，即使 FOIL 能够找到正确的常量序，如前所述，在 FOIL 对递归文字的排序过程中，这种递归文字的序也有可能有多个(即符合 I 的全排列 $(i_1,i_2,\ldots,i_n)$ 有多个)，而 FOIL 找到的却不能够反映主要矛盾，这也可能导致 FOIL 学习不出正确的递归规则. 虽然 FOIL 可以依赖用户给出常量序，但这只对少数较为简单的学习任务有效，且这是以增加用户负担为代价的. 最后，需要指出一点，常量序与递归规则没有本质联系，能够反映递归规则本质的，是下一节将要讨论的实例图和实例序.

3 实例图和实例序

3.1 规则空间与实例空间中的几个基本概念

定义 1. e 为子句 c 的覆盖例是指，c 无递归体文字，且存在 c 的某个头为 e 的例化 c'，c' 的所有体文字均为背

景正例. 此时也说 c 覆盖 e(记作 $c\models e$),例化 c' 是覆盖的,c' 是(c 的)覆盖例化.

定义 2. e 为子句 c 的抛弃例是指,c 的任意头为 e 的例化 c',c' 中存在体文字为背景反例. 此时也说 c 抛弃 e(记作 $c\not\models e$),例化 c' 是抛弃的,c' 是(c 的)抛弃例化.

定义 3. e 为子句 c 的悬挂例是指,c 有递归体文字,且存在 c 的某个头为 e 的例化 c',c' 的所有体文字均为背景正例. 此时也说 c 悬挂 e(记作 $c\models\!\!\models e$),例化 c' 是悬挂的,c' 是(c 的)悬挂例化.

定义 4. e 为规则集 R 的覆盖例是指,存在 R 的某个非递归子句 c 覆盖 e,或存在 R 的某个递归子句 c 的头为 e 的例化 c',c' 的非递归体文字均为背景正例,递归体文字全被 R 覆盖. 也说 R 覆盖 e(记作 $R\models e$).

定义 5. e 为规则集 R 的抛弃例是指,R 的任意子句 c 均抛弃 e. 也说 R 抛弃 e(记作 $R\not\models e$).

定义 6. e 为规则集 R 的悬例是指,R 的任意非递归子句 c 抛弃 e,且存在 R 的某个递归子句 c 的头为 e 的例化 c',c' 的非递归体文字均为背景正例,递归体文字不全被 R 覆盖. 也说 R 悬挂 e(记作 $R\models\!\!\models e$).

定义 7. 子句 c(或规则集 R)在实例空间 E 中的覆盖例集、悬例集、抛弃例集分别记作 $E_{c(c)}$(或 $E_{c(R)}$),$E_{h(c)}$(或 $E_{h(R)}$),$E_{a(c)}$(或 $E_{a(R)}$),在不引起混淆的情况下简记为 E_c,E_h,E_a.

3.2 实例图和实例序

若无特殊说明,以下所说的图均指超图,所说的弧均指超弧,所说的(有向)边均指两点间的(有向)线段. 如果某个目标关系的实例空间为 $E=\{e_1,e_2,\ldots,e_q\}$,规则空间中任意一个规则集 $R=\{c_1,c_2,\ldots,c_p\}$,则由 E 和 R 可唯一确定一个有向带权超图 $H_{(R,E)}$(简记为 H). E 的元素 e_i 与 H 的点 d_i 一一对应;R 在 E 下的非抛弃例化的集合的元素与 H 的超弧一一对应,其中非递归子句 c_u 的头为 e_i 的例化映射为如图 1 所示的超弧(图中 true 是我们定义的一个特殊点),递归子句 c_w 的头为 e_k 的例化映射为如图 2 所示的超弧(图中 $d_{j1},\ldots,d_{jt}$ 是该子句的所有递归体文字的例化所对应的点). 我们称 $H_{(R,E)}$ 为规则集 R 在实例空间 E 下的实例图(在不引起混淆的情况下,简称实例图). 不难看出,$H_{(R,E)}$ 的每个极大无环子超图都反映了实例间的一个序,我们称之为实例序(此处的无环是指有向边无环). 显然,实例序是偏序.

图1

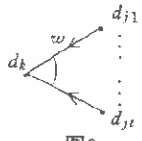

图2

3.3 实例图中的概念

非递归规则集的实例图中只存在如图 1 所示的类型的超弧,退化为一颗星与孤立的点的并集. 以后我们只考虑递归规则集的实例图. 定义 H 中的覆盖弧、悬弧分别为覆盖例化和悬挂例化对应的超弧. 覆盖点、悬点、抛弃点分别为覆盖例、悬例、抛弃例对应的点. 显然,若能从 true 到达一条弧,这条弧就是覆盖弧,否则就是悬弧. 入度为 0 的点为抛弃点;入度不为 0 的点,若能从 true 到达该点,这个点就是覆盖点,否则就是悬点.

H 中可能有权值相同或不同的重弧或重的有向边. 称存在实例 e_i 被多个子句覆盖或悬挂的规则集为二义规则集,e_i 在二义规则集下的归结过程不唯一. 相应地,二义图是指,图中存在点 d_i,以 d_i 为终点的弧存在多种权值. 不难看出,H 中任意能够从 true 到达(超弧意义下)覆盖点 d_i 的极小子图,都对应覆盖例 e_i 的一个无二义归结. 称这样的子图为 d_i(在 H 中)的一个归结图. H 的一个归结图是指 H 的这样的子图,H 中任意一个覆盖点在该子图中至少存在一个归结图.

3.4 FOIL 基于实例图的再分析

FOIL 学习过程中的递归规则集 R 对应的 $H_{(R,E)}$ 中肯定不存在环,但可能是二义的. 由实例序定义知 $H_{(R,E)}$ 只有唯一的实例序. 在常量有限的实例空间 E 中学出的规则集 R,在实际应用于常量无限的实例空间 E' 时可能存在无限递归(但不是循环递归),这个缺陷对所有一阶 ILP 系统都存在,我们不予考虑.

现在,我们通过实例图和实例序分析 FOIL 递归规则学习算法在理论上的缺陷. 下面均假定 FOIL 已通过用户定义或常量排序获得了一个常量序,且当前候选递归规则集 R 的所有递归文字满足一个基于该常量序的文字序.

缺陷 1. 如前面两个例子所示,常量有序与递归规则没有本质联系. FOIL 的文字序是基于这种可能是不恰当的常量序.

缺陷2. 文字序确定的实例顺序是全序,当前规则集确定的实例序是偏序.每个可能的文字序是满足该偏序的一个全序.随着递归规则的增加,偏序越来越紧致,可能文字序也越来越少,满足1.2节的递归文字也越来越少(也即候选递归文字空间越来越小).如果前面采纳了不恰当的递归规则,造成偏序的不恰当,使必须的递归文字过早地从候选递归文字空间排除,则势必导致学习失败.而出现这种情况的可能性是很大的.

3.5 小 结

通过以上分析,我们得出如下结论:任何递归学习任务的实例空间 E 对于规则集 R 都存在实例序,它(们)由前面定义的实例超图 $H_{(R,E)}$ 的每个极大无环子图反映出来.FOIL的递归学习算法的缺陷就在于企图用常量序和文字序来反映这种实例序,但事实上,常量序和文字序与实例序并无本质关系.我们下面给出的改进算法就是通过把握住这种更为本质的实例序,自然而然地防止了病态递归规则集的产生.

4 FOILPlus算法

下面约定正确规则集为不覆盖任何反例且覆盖所有正例的规则集.正确规则集对应的 H 可以分割为两个子图 H_p 和 H_n,H_p 的所有点均为正例点,正例点都是覆盖点;H_n 的所有点均为反例点,反例点都是非覆盖点.H 可能有环,但 H 至少有一个归结图.

4.1 基于实例图的FOILPlus算法框架

算法中的 R 是当前规则集,P,N 是目标的正、反例集,B 为背景正、反例集,允许 $P \cup N \neq E$.

```
FOILPlus(B,E,P,N)
    R:=∅;better=true;
    while better do
        发现一个子句 c,使 H(R∪{c},E) 比 H(R,E) 更令人满意;
        if 这样的 c 找不到
        then
            better=false;
        else
            R=R∪{c};
    输出 R.
```

$H_{(R,E)}$ 的满意度为综合考虑 H 中各项参数(如覆盖、悬挂正例点数,覆盖、悬挂反例点数等)和 R 的复杂程度后对 H 的一个评价.规则集 R 越简单、紧凑,覆盖正例数越多,覆盖反例数越少,$H_{(R,E)}$ 的满意度就越高.满意度随我们对所需规则集 R 的要求的改变而改变,也就是说我们对规则集 R 的要求反映为对 $H_{(R,E)}$ 的满意度的定义.例如,我们需要一个正确的规则集,则所有覆盖反例点的实例图的满意度为 $-\infty$.

4.2 FOILPlus算法

如4.1节中的FOILPlus算法框架所示,FOILPlus算法的核心思想是在规则集 R 生成的同时维护 $H_{(R,E)}$,并通过 $H_{(R,E)}$ 指导下一个子句的生成.

算法没有避免 $H_{(R,E)}$ 的环的出现,理由是:若算法禁止 $H_{(R,E)}$ 中出现环,则与3.4节中的FOIL的缺陷2所述类似,前面不恰当的递归规则的生成可能阻碍后面正确递归规则的生成;且若规则空间中不存在递归规则集 R 使 $H_{(R,E)}$ 无环,则所有 $H_{(R,E)}$ 有环的正确规则集 R 都将不可能生成,这是不应该的(例子见第5节).而且,可以在最后对正确规则集消除冗余的同时,进行 $H_{(R,E)}$ 的去环(虽然没太大必要).因为这两个操作都是NP-hard的,所以在实际系统中可以作为选项提供.

算法确保了"算法认为某个实例被规则集 R 覆盖,则该点在 $H_{(R,E)}$ 中必存在归结图"这个结论.我们认为这是有重要意义的.这确保了算法产生的规则集 R 不是FOIL担心的那种看似覆盖了所有正例,其实其中一部分正例是被悬挂的病态规则集.

为了弥补算法生成的规则中可能隐含循环递归而导致归结不终止的缺陷,我们在FOILPlus算法之外又给出了与之配套的归结算法Resolve.该算法的归结速度在最坏情况下较之FOIL依赖的归结[5]也仅有对数级的提高.我们认为这是可以接受的.

算法中 R,E,P,N,B 定义如前. E_c,E_h,E_a 分别为 R 的覆盖点集、悬点集、抛弃点集. 算法在 FOIL 算法的基础上,每个实例增加了记录 Example,Example 的各个域定义如下.

id　　实例 e 的编号(起唯一标识作用)

Aset　　集合,其每个元素是以 e 为终点的悬弧的所有始点的编号的集合

S　　以 e 为始点的有向边的终点的编号的集合

域 *Aset* 和 *S* 起一个相互索引的作用,使递归函数 unhang 高速运作.

```
FOILPlus(B,E,P,N)
    R=∅;Ec=∅;Eh=∅;Ea=E;better=true;
    for (e∈E)
        e.Aset=∅;
        e.S=∅;
    while ∃ e∈P 但 e∉Ec and better do
        c=FindClause(Ec,Eh,Ea,P,N,B);
        if c=∅
        then
            better=false;
        else
            R=R∪{c};
            adjustEcEhEa(c,Ec,Eh,Ea,P,N,B);
    输出 R
adjustEcEhEa(c,Ec,Eh,Ea,P,N,B)
    if c 为非递归子句
    then
        for (c 的每个覆盖例 e)
            if e∉Ec
            then
                将 e 由 Eh 或 Ea 移到 Ec;
                unhang(e);
    else
        for (c 的每个悬挂例化 c',记 c'头为 e,体为 e1,...,ej)
            if e∉Ec
            then
                if {e1,...,ej}⊆Ec
                then
                    将 e 由 Eh 或 Ea 移到 Ec
                    unhang(e)
                else
                    if e∉Eh then 将 e 由 Ea 移到 Eh
                    A={ek.id|ek∉Ec};
                    e.Aset=e.Aset∪{A};
                    for (每个 ei.id∈A)
                        ei.S=ei.S∪{e.id}.
unhang(e)
    for (e'.id∈e.S)
        if e'∈Eh
        then
            for (A∈e'.Aset)
                if e.id∈A
                then
                    A=A−{e.id};
```

if $A=\varnothing$
then
　　将 e' 由 E_h 移到 E_c;
　　unhang(e');
　　跳出此 for 循环.

算法中的 FindClause 函数继承了 FOIL 中 FindClause 函数[7]的所有启发式搜索功能,但在评价和加入一个文字时依赖了悬例的作用.FOIL 算法中将文字划分为 determinate 和 gainful 两类.本算法在决定是否 determinate 文字时将悬例视作覆盖例.若一个部分子句有 $n^{\oplus}$个正例化和 $n^{\ominus}$个负例化,加入某 gainful 文字后有 $m^{\oplus}$个正例化和 $m^{\ominus}$个负例化,记 $m^{\oplus}$个正例化实际覆盖了 k 个正例,则 FOIL 对该文字的评价为

$$k\times[I(n^{\oplus},n^{\ominus})-I(m^{\oplus},m^{\ominus})], \tag{4}$$

其中 $I(n^{\oplus},n^{\ominus})=-\log_2[n^{\oplus}/(n^{\oplus}+n^{\ominus})]$bits.

针对悬例,我们引入了 $n^{\odot}$和 $m^{\odot}$分别表示该部分子句加入文字前后悬例化的个数.记 $m^{\odot}$个悬例化覆盖的悬例个数为 h,则该文字的评价为

$$(k+h/2)\times[I(n^{\oplus},n^{\odot},n^{\ominus})-I(m^{\oplus},m^{\odot},m^{\ominus})], \tag{5}$$

其中 $I(n^{\oplus},n^{\odot},n^{\ominus})=-\log_2[(n^{\oplus}+n^{\odot}/2)/(n^{\oplus}+n^{\odot}+n^{\ominus})]$bits.

显然,当该部分子句和该文字中均无递归文字时,$n^{\odot}=m^{\odot}=h=0$,(5)式转化为(4)式.

4.3 归结算法 Resolve

本算法只用于常量离散的情形.当常量连续时,FOILPlus 和 FOIL 一样,不会生成递归规则,所以,此时的归结与一般的归结相同.算法中使用了平衡树(balanced tree,也称 AVL trees)[8]存当前的归结路径.平衡树的查找和维护要求任意两个实例间能够比较大小.一般地,算法均事先指定常量序为表示常量的字符串的顺序,再用式(1)(令其中 $i_k=k,k\in\{1,\ldots,j\}$)即可判断任意实例间的大小.

```
Resolve(e,R)
    AVL=∅;
    return Cover(e,R).
Cover(e,R)
    if e∈AVL then return false;
    AVL=AVL∪{e};
    for (c∈R)
        if c 为非递归子句
        then
            if 存在 c 的覆盖例化覆盖 e
            then
                AVL=AVL−{e};
                return true;
        else
            for (c 的每个头为 e 的悬挂例化 c',记 c'的体为 e1,...,ej)
                if Cover(e1,R) and ... and Cover(ej,R)
                then
                    AVL=AVL−{e};
                    return true;
    AVL=AVL−{e};
    return false.
```

4.4 算法时空复杂度分析

记 n 为实例数.在数据结构上,FOILPlus 算法比 FOIL 主要增加了实例记录 Example.因为所有实例的 Example存的都是当前 $H_{(R,E)}$中所有悬弧的信息,且每条悬弧恰好被存储一次,所以算法在整个执行过程中比 FOIL 多使用的存储空间至多为 $S(n)=O(\max\{M_R\times N_R\mid R\in\{$算法执行过程中的每个当前规则集$\}\})$,其中 $M_R=H_{(R,E)}$的悬弧数,$N_R=H_{(R,E)}$悬弧的平均始点数$=R$ 中递归规则的平均体递归文字数.因为在实际应用中,即

使像前面提到的 Ackermann 函数这种困难任务,每个递归规则的体递归文字数也不超过 2,所以,可以认为 $O(N_R)=1$;$O(M_R)=O(H_{(R,E)}$的悬弧数$)=O(H_{(R,E)}$的悬点数$)\times O$(以每个悬点为终点的平均悬弧数)$\leqslant O$(实例点数)$\times O(1)=O(n)$. 所以,$S(n)\leqslant O(n)$. 显然,FOIL 的空间复杂度至少为 $O(n)$,所以,FOILPlus 与 FOIL 的空间复杂度是相等的.

FOILPlus 算法较之 FOIL 算法主要增加了 unhang 操作. 注意,递归函数 unhang 在整个算法执行过程中对每个覆盖例调用且仅调用一次,与 FOIL 时间花费的瓶颈 FindClause 比较,完全可以忽略不记. 在 FindClause 中,FOILPlus 与 FOIL 相比,对于那些 FOIL 通过文字序排除的递归文字有更多的评价,且计算 $n^{\odot}$和 $m^{\ominus}$也要多些. 但粗略分析可知,这至多使 FOILPlus 比 FOIL 在时间花费上有常数因子(绝大多数情况下<1)的增加. 所以,算法的时空复杂度都是令人满意的.

对于归结算法 Resolve,记算法在整个归结过程中共归结了 m 个实例. 不难看出,该归结过程对应以 e 为根的非超图意义下的有向树 T,T 上任意有向边(e_i,e_j)称 e_i 为 e_j 的孩子. 记 T 上结点 e_i 的深度为 d_i,则算法对该点的 AVL 的查找和维护的时间花费为 $O(d_i)$,所以算法时间复杂度为

$$T(m)=\Sigma[O(1)+O(d_i)]=O(m)+O(\Sigma d_i)=O(m)+O(m\times d)=O(m\times\log_2 m)=O(\log_2 m)\times T'(m),$$

其中 $d=O(\log_2 m)$为 T 的平均深度,$T'(m)=O(m)$为普通归结[5,9]的时间复杂度. 算法的空间复杂度 $S(m)=O(\max(d_i))\leqslant O(m)$. 所以归结算法 resolve 的时空复杂度都是可以接受的. 当常量有限或归结实例范围已知时,可以用一个 k 维数组记录归结路径,此时,有 $T(m)=T'(m)$.

4.5 小　结

对于一个非递归学习任务,FOILPlus 算法的运作和 FOIL 完全相同;对于一个递归学习任务,算法由于在文字级别引入了"悬例"而使得 gain 值的计算更趋合理,在子句级别引入了"悬例"而避免了病态递归规则集的产生. 事实上,在一个递归学习任务中,正例加入 E_i 的顺序,与我们前面定义的"实例序"是一致的! 所以,算法在没有对规则空间加以任何限制的情况下,通过引入"悬例"、"悬弧"把握住"实例序",自然而然地避免了循环递归等病态递归规则的产生.

5　实验结果

我们用标准 C 语言实现了上述算法的全部基本功能,并将其用于学习任务 *map* 和 *binary*. 实验结果如下.

(1) 当规则空间中不含 FOIL 的内嵌文字"≠"时,*map* 的学习结果为规则集

$$\begin{cases} map(g_1,g_2)\leftarrow \\ map(A,B)\leftarrow succ(C,A),next(D,B),map(C,D) \end{cases}, \tag{6}$$

该规则集对应的 H_p 如图 3 所示.

虽然图中有环,但显然对任意点存在以 true 为始点,该点为终点的无环链,该链即为该点对应实例的归结图. 我们认为这样的规则集是可以接受的. 当我们将"≠"加入规则空间后,学习结果代之以规则集(2).

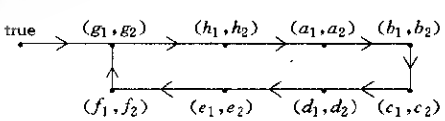

图3

(2) *binary* 的学习结果为规则集

$$\begin{cases} Binary(0,0)\leftarrow \\ binary(A,B)\leftarrow binary(C,B),half(A,C),even(A) \\ binary(A,B)\leftarrow odd(A),succ(C,B),binary(D,C),succ(D,A) \end{cases}, \tag{7}$$

式(7)与规则集(3)的第 1、2 两个子句相同,第 3 个子句略有不同,但都是本质的.

6　结论和进一步的工作

实例图的引入,在原本分割的规则空间和实例空间之间搭起一座桥梁. 我们在实例图理论指导下设计的 FOILPlus 算法,在时空复杂度没有增加的前提下,除规则搜索(生成)的启发式机制的不完备(这是目前无法避

免的)和覆盖性检验过程实现的不完备(一阶ILP学习系统都存在此问题)之外,FOILPlus本身对可学习到的规则集没有任何限制,这是FOIL等ILP学习系统都无法做到的.不难看出,实例图以及FOILPlus算法都能推广到多谓词学习中!我们相信这将显示出实例图更重要的作用——这也是我们下一步要进行的工作.

致谢 本文的研究工作得到国家自然科学基金资助,此项目编号为69875017.

参考文献

1 King R D, Muggleton S, Lewis R A *et al*. Drug design by machine learning: the use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. Proceedings of the National Academy of Sciences of USA, 1992,89:11322～11326

2 Bratko I, Dzeroski S. Engineering applications of ILP. New Generation Computing, 1995,13(3～4):313～333

3 Srinivasan A, Muggleton S H *et al*. Theories for mutagenicity: a study in first-order and feature-based induction. Artificial Intelligence, 1996,85(1/2):277～299

4 Muggleton S. Inductive logic programming: derivations, successes and shortcomings. SIGART Bulletin, 1994,5(1):5～11

5 Quinlan J R, Cameron-Jones R M. Induction of logic programs: foil and related systems. New Generation Computing, 1995,13(3～4):287～312

6 Cameron-Jones R M, Quinlan J R. Efficient top-down inductive of logic programs. Inductive Logic Programming, SIGART Bulletin, 1994,5(1):33～42

7 Quinlan J R. Knowledge acquisition from structured data. IEEE Expert, 1991,6(6):32～37

8 Knuth D E. The Art of Computer Programming (Volume 3): Sorting and Searching. Amsterdam: Addison-Wesley Publishing Company, Inc., 1973

9 Lloyd J W. Foundations of Logic Programming. Berlin: Springer-Verlag, 1984

An ILP Algorithm Without Restriction of Constant Ordering

ZHANG Run-qi[1] CHEN Xiao-ping[2] LIU Gui-quan[2]

[1](*Department of the Special Class for the Gifted Young University of Science and Technology of China Hefei* 230026)

[2](*Department of Computer Science University of Science and Technology of China Hefei* 230027)

Abstract In this paper, the shortcomings in theory and limitation in applications of FOIL (first-order inductive learner) are analyzed. To overcome these difficulties, instance graph $H_{(R,E)}$ and instance order are introduced to clarify the relationship between the set R of recursive rules and the instance space E. Based on these concepts, a new ILP (inductive logic programming) algorithm, FOILPlus, is put forward, which prevents the generation of harmful recursive rules by utilizing hung example and hung arc to hold Instance Graph. The algorithm can complete learning tasks without the restriction of constant ordering, and does not substantially raise the computational complexity compared with FOIL. FOILPlus has been implemented, and experiments show that it does complete two learning tasks which FOIL fails.

Key words ILP (inductive logic programming), FOIL (first-order inductive learner), recursive, instance graph, instance order, hung example, hung arc, FOILPlus.