**Instituto de Ciências Matemáticas e de Computação**

**ISSN - 0103-2569**

# Ferramentas para Conversão de Dados de Entrada dos Algoritmos de AM Proposicional $\mathcal{CN}2$ e $\mathcal{C}4.5$ para os Algoritmos de AM Relacional Foil e Progol

**Chandler Wellington Caulkins**

**Maria Carolina Monard/ILTC**

**Nº 116**

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos

Junho/2000

# Ferramentas para Conversão de Dados de Entrada dos Algoritmos de AM Proposicional $\mathcal{CN}2$ e $\mathcal{C}4.5$ para os Algoritmos de AM Relacional Foil e Progol *

**Chandler Caulkins**
**Maria Carolina Monard/ILTC**

Universidade de São Paulo
Departmento de Ciências de Computação e Estatística
Laboratório de Inteligência Computacional
Instituto de Ciências Matemáticas e de Computação
Cx. Postal 668
13560-970 - São Carlos, SP
e-mail: {chandler, mcmonard}@icmc.sc.usp.br

**Resumo** Para a expressão de qualquer paradigma de aprendizado indutivo são necessárias, no mínimo, linguagens para descrever objetos, ou exemplos, e linguagens para descrever os conceitos aprendidos. Em Aprendizado de Máquina – AM – indutivo, essas linguagens dividem-se basicamente em dois tipos de descrição: *descrição baseada em atributos ou proposicional* e *descrição relacional*. Em princípio, qualquer problema que possa ser resolvido utilizando AM proposicional também pode ser resolvido com AM relacional. Para isso, é necessário transformar os exemplos no formato atributo-valor para o formato relacional. Neste trabalho, é descrita uma metodologia para essa transformação, inicialmente proposta por Katarina Morik e Peter Brockhausen (Morik e Brockhausen, 1997). A metodologia foi implementada através de ferramentas criadas para a conversão de dados de entrada dos algoritmos de AM proposicional $\mathcal{CN}2$ e $\mathcal{C}4.5$ para os algoritmos de AM relacional Foil e Progol.

*Palavras Chaves:* conversão de dados, formato atributo-valor, aprendizado de máquina proposicional, aprendizado de máquina relacional, $\mathcal{CN}2$, $\mathcal{C}4.5$, Foil, Progol

Este documento encontra-se no site da Biblioteca do ICMC

ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_116.ps.zip

**Junho 2000**

*Trabalho realizado com auxílio parcial de CAPES e FINEP.

# Sumário

**1 Introdução** **1**

**2 Formato Atributo-Valor** **2**

**3 Método MB – Morik e Brockhausen** **3**

**4 Sistemas de PLI Contemplados** **5**

4.1 Foil . . . . . 5

4.2 Progol . . . . . 5

**5 Ferramentas para o Processo de Conversão** **5**

5.1 De $\mathcal{CN}2$ para $\mathcal{C}4.5$ . . . . . 7

5.2 De $\mathcal{C}4.5$ para Foil "Proposicional" . . . . . 10

5.3 De Foil "Proposicional" para Progol "Proposicional" . . . . . 12

5.4 De Foil "Proposicional" para Progol MB . . . . . 16

5.5 De Foil "Proposicional" para Foil MB . . . . . 19

**6 Considerações Finais** **23**

**A Código-Fonte das Ferramentas** **25**

A.1 foil2progol . . . . . 25

A.2 foil2progolmb . . . . . 32

A.3 foil2foilmb . . . . . 40

# Lista de Figuras


1 Uma visão geral do processo de conversão . . . . . 6
2 De $\mathcal{CN}2$ para $\mathcal{C}4.5$ . . . . . 7
3 Arquivo $\mathcal{CN}2$ `voyage.att` . . . . . 8
4 Arquivo $\mathcal{CN}2$ `voyage.exs` . . . . . 8
5 Arquivo $\mathcal{CN}2$ `voyage.tst` . . . . . 9
6 Arquivo $\mathcal{C}4.5$ `voyage.names` . . . . . 9
7 Arquivo $\mathcal{C}4.5$ `voyage.data` . . . . . 9
8 Arquivo $\mathcal{C}4.5$ `voyage.test` . . . . . 10
9 De $\mathcal{C}4.5$ para FOIL "Proposicional" . . . . . 10
10 Arquivo $\mathcal{C}4.5$ `voyage.names` modificado para uso com `c45_to_foil` . . . . . 11
11 Arquivo FOIL "Proposicional" `voyage.d` . . . . . 11
12 De FOIL "Proposicional" para PROGOL "Proposicional" . . . . . 13
13 Arquivo PROGOL "Proposicional" `voyage_training.pl` . . . . . 14
14 Arquivo PROGOL "Proposicional" `voyage_test.pl` . . . . . 15
15 Arquivo PROGOL "Proposicional" `voyage_extra_bg_test.pl` . . . . . 16
16 De FOIL "Proposicional" para PROGOL MB . . . . . 16
17 Arquivo PROGOL MB `voyage_training_mb.pl` . . . . . 17
18 Arquivo PROGOL MB `voyage_test_mb.pl` . . . . . 19
19 De FOIL "Proposicional" para FOIL MB . . . . . 19


# Lista de Tabelas


1 Formato Atributo-Valor (ou Planilha) para Dados . . . . . 3
2 Formato Atributo-Valor para Exemplos e Contra-exemplos de um Conceito . . . . 3
3 Condições Climáticas para uma Viagem . . . . . 7
4 Arquivo PROGOL MB `voyage_mb.d` . . . . . 22

# 1 Introdução

Dentre as várias formas de aprendizado está o *aprendizado indutivo*. Um problema de indução pode ser formulado como uma tarefa de aprendizado de conceitos a partir de dados, ou *aprendizado indutivo de conceitos*, onde regras de classificação para um determinado conceito devem ser induzidas a partir de instâncias positivas e/ou negativas do conceito.

Para a expressão de qualquer paradigma de aprendizado indutivo são necessárias, no mínimo, linguagens para descrever objetos, ou exemplos, assim como linguagens para descrever os conceitos aprendidos. Vários formalismos lógicos têm sido utilizados em sistemas de aprendizado indutivo para a representação de exemplos e conceitos. Em geral, dividem-se em dois tipos de descrição: *descrição baseada em atributos ou proposicional* e *descrição relacional*.

No primeiro caso, as linguagens de descrição de conceitos utilizadas pelos algoritmos para expressar as hipóteses induzidas, por exemplo árvores de decisão e regras de produção, podem ser tratadas como variantes de linguagens baseadas em atributo-valor, *i.e.* linguagens proposicionais.

O formato atributo-valor é o mais utilizado em AM proposicional. Objetos são descritos em termos de atributos e valores desses atributos, através de um vetor contendo valores para os atributos de um determinado exemplo. Vantagens dessa representação são a sua simplicidade e eficiência bem como uma grande quantidade de técnicas desenvolvidas para lidar com ruídos nos dados.

No entanto, o aprendizado baseado em atributo-valor é limitado a descrições não-relacionais no sentido de que as descrições aprendidas *não* especificam *relações* entre as partes dos objetos. Com isso, a inclusão de conhecimento prévio do domínio é limitada e a própria representação se torna inapropriada para domínios nos quais existem relações intrínsecas.

Uma abordagem relativamente recente de AM indutivo, chamada Programação Lógica Indutiva – PLI, busca contornar algumas das limitações das abordagens anteriores de AM através do uso de teoria do domínio (ou conhecimento prévio) estruturada e uma linguagem de descrição de exemplos e conceitos baseada em lógica de primeira ordem. A adoção por sistemas de aprendizado de linguagens lógicas de primeira ordem como linguagens de representação, permite que relações e/ou estruturas possam ser induzidas. Isto faz com que o espaço dos conceitos passíveis de serem aprendidos seja aumentado.

Em princípio, qualquer problema que possa ser resolvido utilizando AM proposicional também pode ser resolvido com AM relacional. Para tanto, cada exemplo no formato atributo-valor pode ser representado utilizando predicado(s) cujos argumentos são os valores dos atributos para aquele exemplo. Assim, um primeiro passo na conversão do formato atributo-valor para o formato relacional seria transformar os exemplos em fatos utilizando a notação relacional. No entanto,

isso também força o sistema a considerar todos os atributos que estão contidos nos predicados, embora pode ser que nem todos sejam necessários.

Em aprendizado relacional, é simples fornecer aos sistemas conhecimento de fundo sobre um domínio de aplicação para formar a hipótese sobre o conceito desejado. Uma forma de fazer isso é numerar os exemplos, e apresentar os valores dos atributos como sendo o conhecimento de fundo sobre o exemplo. Ou seja, cada atributo pode se relacionar com o exemplo através de um predicado único para esse atributo. Com isso, o sistema pode considerar somente aqueles predicados que são significativos para a definição do conceito a ser induzido. Essa metodologia para transformar exemplos no formato atributo-valor para o formato relacional foi inicialmente proposta por Katarina Morik e Peter Brockhausen (Morik e Brockhausen, 1997). Neste trabalho, é descrita em detalhes uma implementação dessa metodologia a qual denominamos de "método MB", bem como a descrição da implementação de outras ferramentas criadas para realizar a conversão de vários formatos de dados utilizados por alguns algoritmos de aprendizado proposicional e relacional comumente utilizados pela comunidade de AM. As implementações foram realizadas na linguagem PERL (Schwartz e Christiansen, 1997).

O relatório está organizado da seguinte forma: Primeiramente, é descrito o formato atributo-valor (ou proposicional) — Seção 2. Em seguida discute-se uma maneira de transformar esse formato para uma representação apropriada para o aprendizado relacional, através do chamado "método MB" — Seção 3. É feita também uma breve descrição dos dois sistemas de Programação Lógica Indutiva, FOIL (Quinlan, 1990) e PROGOL (Muggleton, 1995), para os quais foi desenvolvida a conversão — Seção 4. A partir de representações utilizadas por $\mathcal{CN}2$ (Clark e Boswell, 1991) e $\mathcal{C}4.5$ (Quinlan, 1993), dois dos sistemas de aprendizado proposicional mais conhecidos, são obtidas representações para os dois sistemas de PLI FOIL e PROGOL. Duas representações são consideradas: o formato atributo-valor representado na notação relacional, bem como a representação obtida pelo "método MB", as quais são processadas tanto por FOIL quanto por PROGOL. Para obtê-las, um processo de conversão foi desenvolvido, e ferramentas foram criadas para implementar esse processo — Seção 5. Finalmente, na Seção 6 são apresentadas as conclusões.

## 2 Formato Atributo-Valor

Em aprendizado de máquina proposicional, um conjunto de exemplos de treinamento é apresentado ao sistema de indução. Cada exemplo, ou instância, é descrito por um vetor contendo valores de atributos e um rótulo que atribui uma classe ao exemplo. A Tabela 1 mostra o formato geral de um conjunto de dados com $n$ instâncias e $m$ atributos. Nessa tabela uma linha $i$ refere-se ao $i$-ésimo exemplo ou instância ($i = 1, 2, \ldots, n$) e as entradas de coluna $x_{i,j}$ referem-se ao valor do $j$-ésimo atributo $X_j$ ($j = 1, 2, \ldots, m$) da instância $i$.

| $a_1$ | $a_2$ | $\cdots$ | $a_m$ | *classe* |
|---|---|---|---|---|
| $x_{1,1}$ | $x_{1,2}$ | $\cdots$ | $x_{1,m}$ | $y_1$ |
| $x_{2,1}$ | $x_{2,2}$ | $\cdots$ | $x_{2,m}$ | $y_2$ |
| $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $x_{n,1}$ | $x_{n,2}$ | $\cdots$ | $x_{n,m}$ | $y_n$ |

Tabela 1: Formato Atributo-Valor (ou Planilha) para Dados

Pode ser observado que instâncias são tuplas $(x_{i,1}, x_{i,2}, \ldots, x_{i,m}, y_i) = (\mathbf{x}_i, y_i)$ também denominadas ($\mathbf{X}$, classe($\mathbf{X}$)) ou $(\mathbf{X}, Y)$ onde o último atributo, $Y$, é o que se deseja predizer dados os outros atributos $\mathbf{X}$ *i.e.*, $Y = f(\mathbf{X})$.

Esse formato é bastante utilizado por sistemas de aprendizado proposicional (Baranauskas e Monard, 2000b). A seguir é descrito um método para converter esse formato para uma representação mais próxima da utilizada por sistemas de Programação Lógica Indutiva.

# 3 Método MB – Morik e Brockhausen

Em princípio, todo problema que possa ser resolvido com um sistema de aprendizado proposicional pode ser resolvido também com um sistema de PLI (Cameron-Jones e Quinlan, 1993). Para isso, exemplos em formato atributo-valor podem ser considerados como fatos relacionados a um único predicado `exemplo/n`. Esse predicado se torna então o conceito procurado e faz parte da cabeça de cláusulas induzidas por um sistema de aprendizado de máquina relacional.

| *#id* | $a_1$ | $a_2$ | $\cdots$ | $a_m$ | *classe* |
|---|---|---|---|---|---|
| 1 | $x_{1,1}$ | $x_{1,2}$ | $\cdots$ | $x_{1,m}$ | $+$ |
| 2 | $x_{2,1}$ | $x_{2,2}$ | $\cdots$ | $x_{2,m}$ | $-$ |
| $\vdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $i$ | $x_{i,1}$ | $x_{i,2}$ | $\cdots$ | $x_{i,m}$ | $+$ |
| $\vdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ | $\cdots$ |
| $n$ | $x_{n,1}$ | $x_{n,2}$ | $\cdots$ | $x_{n,m}$ | $-$ |

Tabela 2: Formato Atributo-Valor para Exemplos e Contra-exemplos de um Conceito

Por exemplo, considerando os exemplos da Tabela 2, semelhante à Tabela 1, para exemplos (classe $+$) e contra-exemplos (classe $-$) de um conceito, o predicado `exemplo/n` seria representado da seguinte forma:

`exemplo(`$x_{1,1}$, $x_{1,2}$, $\ldots$, $x_{1,m}$`).`

$\vdots$

`exemplo(`$x_{i,1}$, $x_{i,2}$, $\ldots$, $x_{i,m}$`).`

para os exemplos positivos e

`exemplo(`$x_{2,1}$, $x_{2,2}$, ..., $x_{2,m}$`).`

⋮

`exemplo(`$x_{n,1}$, $x_{n,2}$, ..., $x_{n,m}$`).`

para os exemplos negativos.

Cláusulas obtidas utilizando esse predicado como cabeça provavelmente conterão variáveis universalmente quantificadas, pois alguns atributos podem assumir qualquer valor sem afetar a hipótese sendo induzida para o conceito procurado. Ou seja, o valor desses atributos pode não influir na descrição do conceito. Quando essas variáveis não estão restringidas pelo corpo da cláusula, esta se torna muito geral. Mapeando cada atributo da relação a um predicado separado, aumenta-se o número de predicados e, ao mesmo tempo, evita-se a inclusão de variáveis universalmente quantificadas e irrestritas (Morik e Brockhausen, 1997). Ou seja, o conhecimento de fundo é enriquecido ao mesmo tempo em que se elimina uma parte do espaço de hipóteses a ser pesquisado. Com isso, obtém-se uma representação direta de um exemplo em formato atributo-valor, através de uma representação relacional para os valores dos atributos de um exemplo, e ainda de uma forma que pode evitar variáveis universalmente quantificadas.

Na sua forma mais simples, e utilizando o nome dos atributos como os nomes das relações respectivas, os exemplos na Tabela 2 poderiam ser representados da seguinte forma:

`ex(1). a1(`$1, x_{1,1}$`). a2(`$1, x_{1,2}$`). ...am(`$1, x_{1,m}$`).`
`ex(2). a1(`$2, x_{2,1}$`). a2(`$2, x_{2,2}$`). ...am(`$2, x_{2,m}$`).`

⋮

`ex(`$i$`). a1(`$i, x_{i,1}$`). a2(`$i, x_{i,2}$`). ...am(`$i, x_{i,m}$`).`

⋮

`ex(`$n$`). a1(`$n, x_{n,1}$`). a2(`$n, x_{n,2}$`). ...am(`$n, x_{n,m}$`).`

onde `ex(1). ...`, `ex(`$i$`).` são exemplos positivos e `ex(2). ...`, `ex(`$n$`).` são exemplos negativos. O conhecimento representado através dos predicados `a1/2`, `a2/2`, ..., `am/2`, torna-se simplesmente o conhecimento de fundo.

Tendo em vista essas duas representações, uma baseada no formato proposicional (atributo-valor) e outra relacional, é possível imaginar um processo de conversão de dados para as duas formas. Neste trabalho consideramos esse processo especificamente voltado para dois sistemas de aprendizado proposicional e dois sistemas de Programação Lógica Indutiva. Os dois sistemas de aprendizado proposicional considerados, $\mathcal{CN}2$ e $\mathcal{C}4.5$, induzem regras de decisão e árvores de decisão, respectivamente. Maiores detalhes sobre esses dois sistemas podem ser encontrados em (Baranauskas e Monard, 2000a). Os dois sistemas de PLI, Foil e Progol, são brevemente descritos na seção seguinte.

# 4 Sistemas de PLI Contemplados

Os sistemas de PLI contemplados neste trabalho, FOIL e PROGOL, possuem algumas características em comum, como o fato de fazerem uma busca "top-down" e induzirem regras ou cláusulas descritas em lógica de primeira ordem. Diferem principalmente quanto à forma de inclusão de conhecimento prévio do domínio. Ambos sistemas têm sido aplicados com sucesso em diversos domínios de aplicação. Segue uma breve descrição de cada um desses sistemas.

## 4.1 Foil

É um sistema que lê especificações extensionais de um conjunto de relações e produz hipóteses compostas de cláusulas de Horn para uma ou mais delas (Quinlan, 1990). O sistema aprende um predicado por vez, de uma forma não-interativa e não-incremental, fazendo uma busca "top-down", *i.e.* do geral para o específico, no espaço de hipóteses. Seu conhecimento prévio deve ser sempre na forma de listas de fatos.

FOIL aproveita algumas idéias do paradigma de aprendizado baseado em atributo-valor, utilizando um método de cobertura semelhante ao de AQ (Michalski, 1983) e uma heurística baseada em informação semelhante ao de ID3 (Quinlan, 1986). Além disso, herda a idéia de busca "top-down" em grafos de refinamento do MIS (Shapiro, 1983). Seu critério de aceitação baseia-se em ganho de informação e MDL (Minimum Description Length), procurando encontrar cláusulas com comprimento mínimo de descrição, que possam substituir os exemplos sendo considerados.

## 4.2 Progol

PROGOL aprende múltiplos predicados, de forma não-interativa, utilizando uma estratégia de busca mista. Uma facilidade importante, não disponível em FOIL, é que ele também permite a inclusão de conhecimento prévio descrito de forma intensional. Inicialmente PROGOL procura uma hipótese menos geral para servir de limite inferior do reticulado de hipóteses e faz uma busca do geral ao específico, ou seja "top-down", através de um grafo de refinamento das cláusulas (Muggleton, 1995). PROGOL também pode ser usado para aprender somente com exemplos positivos. Seu critério de aceitação se baseia na compressão dos exemplos obtida por cada cláusula.

# 5 Ferramentas para o Processo de Conversão

A Figura 1 ilustra o processo, por nós implementado, de conversão dos dados utilizados pelos sistemas de aprendizado $\mathcal{CN}2$, $\mathcal{C}4.5$, FOIL e PROGOL.

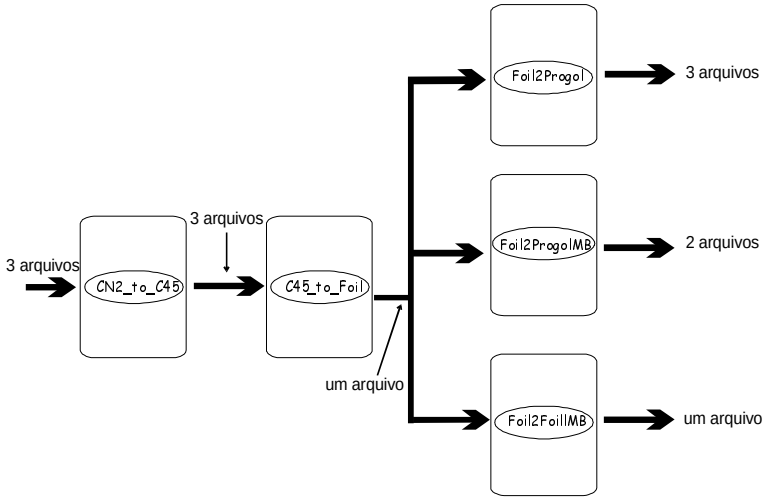


Figura 1: Uma visão geral do processo de conversão

Inicialmente os dados encontram-se em três arquivos na sintaxe requerida por $\mathcal{CN}2$. A ferramenta denominada cn2_to_c45 é responsável pela transformação desses dados para a sintaxe de $\mathcal{C}4.5$, colocando-os em outros três arquivos. Em seguida, a ferramenta c45_to_foil transforma esses dados para a sintaxe que FOIL admite, gerando um único arquivo de dados. O conteúdo desse arquivo pode ser transformado para a sintaxe utilizada por PROGOL, através da ferramenta foil2progol. Pode ainda ser transformado para uma sintaxe de entrada voltada para aprendizado relacional através do "método MB". A ferramenta foil2progolmb é responsável pela transformação para a sintaxe de PROGOL, enquanto foil2foilmb realiza a transformação para a sintaxe de FOIL. A seguir serão descritas cada uma dessas transformações.

A fim de exemplificar melhor o processo de conversão e as ferramentas implementadas, será utilizado um sub-conjunto de um conjunto de dados originalmente proposto por Quinlan (Quinlan, 1993), contendo medidas de condições climáticas para um período de 15 dias.

Cada instância (Tabela 3) é composta dos seguintes atributos:

- *outlook*: assume um dos valores discretos "sunny", "overcast" ou "rain", para os casos da previsão do tempo ser ensolarado, nublado ou chuvoso, respectivamente;
- *temperature*: um valor numérico indicando a temperatura em graus Celsius;
- *humidity*: um valor numérico que indica a porcentagem de umidade no ar;
- *windy*: assume um dos valores discretos "yes" ou "no", indicando se está ventando ou não no dia em questão.

| No. da Inst. | *outlook* | *Temperature* | *Humidity* | *Windy* | *Travel?* |
|---|---|---|---|---|---|
| $T_1$ | sunny | 25 | 72 | yes | go |
| $T_2$ | sunny | 28 | 91 | yes | dont_go |
| $T_3$ | sunny | 22 | 70 | no | go |
| $T_4$ | sunny | 23 | 95 | no | dont_go |
| $T_5$ | sunny | 30 | 85 | no | dont_go |
| $T_6$ | overcast | 23 | 90 | yes | go |
| $T_7$ | overcast | 29 | 78 | no | go |
| $T_8$ | overcast | 19 | 65 | yes | dont_go |
| $T_9$ | overcast | 26 | 75 | no | go |
| $T_{10}$ | overcast | 20 | 87 | yes | go |
| $T_{11}$ | rain | 22 | 95 | no | go |
| $T_{12}$ | rain | 19 | 70 | yes | dont_go |
| $T_{13}$ | rain | 23 | 80 | yes | dont_go |
| $T_{14}$ | rain | 25 | 81 | no | go |
| $T_{15}$ | rain | 21 | 80 | no | go |

Tabela 3: Condições Climáticas para uma Viagem

Cada dia (ou instância) é rotulado como "go" se o tempo estiver bom para se fazer uma viagem à fazenda, ou "dont_go", se o tempo não estiver bom – coluna "*Travel?*" na Tabela 3.

## 5.1 De $\mathcal{CN}2$ para $\mathcal{C}4.5$

O processo de conversão começa pela transformação dos arquivos utilizados por $\mathcal{CN}2$ para os arquivos na sintaxe utilizada por $\mathcal{C}4.5$. O utilitário que realiza esse passo inicial é chamado cn2_to_c45 (Figura 2).

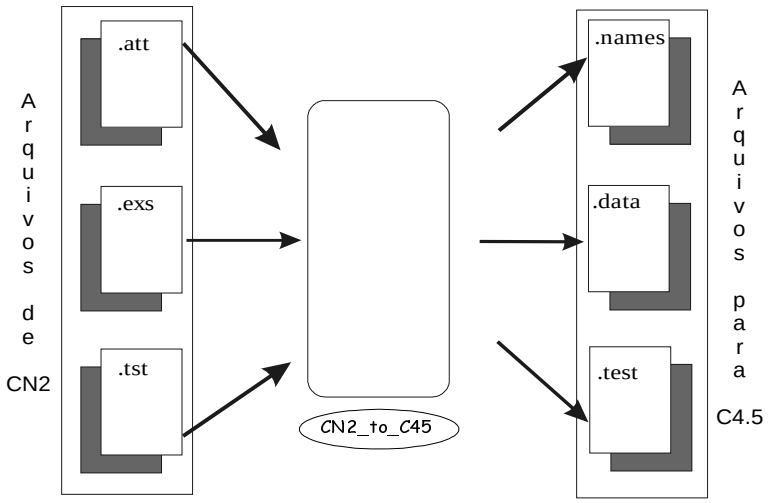


Figura 2: De $\mathcal{CN}2$ para $\mathcal{C}4.5$

A entrada para cn2_to_c45 consiste dos três arquivos padrão utilizados por $\mathcal{CN}2$, cujo conteúdo é descrito a seguir.

1. `voyage.att` (Figura 3) contendo os nomes das classes positiva (`go`) e negativa (`dont_go`) e dos atributos que compõem os exemplos, enumerando seus possíveis valores quando os atributos são discretos. Sua primeira linha deve ser `** ATTRIBUTE FILE **`;

2. `voyage.exs` (Figura 4) contendo os exemplos de treinamento utilizados por $\mathcal{CN}2$ para induzir a hipótese. Sua primeira linha deve ser `** EXAMPLE FILE **`;

3. `voyage.tst` (Figura 5) contendo os exemplos de teste para avaliar a precisão da hipótese induzida. Sua primeira linha também deve ser `** EXAMPLE FILE **`.

É importante lembrar que o símbolo de comentário em $\mathcal{CN}2$ é o símbolo `%`. Assim, no arquivo exemplo `voyage.att` há duas linhas de comentário que indicam a posição das informações sobre atributos e sobre as classes.

```
** ATTRIBUTE FILE **
% Features
outlook:     sunny overcast rain;
temperature: (INT)
humidity:    (FLOAT)
windy:       yes no;

% Classes
class: go dont_go;
```

Figura 3: Arquivo $\mathcal{CN}2$ `voyage.att`

```
** EXAMPLE FILE **
sunny    25 72 yes go
sunny    23 95 no  dont_go
sunny    30 85 no  dont_go
overcast 23 90 yes go
overcast 29 78 no  go
overcast 26 75 no  go
rain     22 95 no  go
rain     19 70 yes dont_go
rain     23 80 yes dont_go
rain     21 80 no  go
```

Figura 4: Arquivo $\mathcal{CN}2$ `voyage.exs`

Após a execução de cn2_to_c45, cada um desses arquivos é transformado, respectivamente, para os três arquivos padrão de entrada utilizados por $\mathcal{C}4.5$:

```
** EXAMPLE FILE **
sunny    28 91 yes dont_go
sunny    22 70 no  go
overcast 19 65 yes go
overcast 20 87 yes dont_go
rain     25 81 no  go
```

Figura 5: Arquivo $\mathcal{CN}2$ `voyage.tst`

1. `voyage.names` (Figura 6) contendo as descrições dos tipos de dados dos atributos e das classes positiva (`go`) e negativa (`dont_go`);
2. `voyage.data` (Figura 7) contendo os exemplos de treinamento;
3. `voyage.test` (Figura 8) contendo os exemplos de teste.

Deve ser observado que os tipos de dados numéricos em $\mathcal{CN}2$, `(INT)` e `(FLOAT)`, são representados por um único tipo de dados, `continuous`, no arquivo de descrição de atributos do $\mathcal{C}4.5$. O símbolo de comentário, representado pelo símbolo `%` na sintaxe do $\mathcal{CN}2$, é equivalente ao símbolo `|` no $\mathcal{C}4.5$. Tudo o que estiver escrito a partir desse símbolo até o final de uma linha é removido no processo de conversão realizado pela ferramenta cn2_to_c45.

```
go, dont_go.

outlook:     sunny, overcast, rain.
temperature: continuous.
humidity:    continuous.
windy:       yes, no.
```

Figura 6: Arquivo $\mathcal{C}4.5$ `voyage.names`

```
sunny,   25,72,yes,go
sunny,   23,95,no, dont_go
sunny,   30,85,no, dont_go
overcast,23,90,yes,go
overcast,29,78,no, go
overcast,26,75,no, go
rain,    22,95,no, go
rain,    19,70,yes,dont_go
rain,    23,80,yes,dont_go
rain,    21,80,no, go
```

Figura 7: Arquivo $\mathcal{C}4.5$ `voyage.data`

```
sunny,   28,91,yes,dont_go
sunny,   22,70,no, go
overcast,19,65,yes,go
overcast,20,87,yes,dont_go
rain,    25,81,no, go
```

Figura 8: Arquivo $\mathcal{C}$4.5 `voyage.test`

## 5.2 De $\mathcal{C}$4.5 para Foil "Proposicional"

O processo de conversão dos dados na sintaxe de $\mathcal{C}$4.5 para a de Foil, com os dados em um formato por nós denominado "proposicional", é realizado pelo utilitário c45_to_foil (Figura 9).

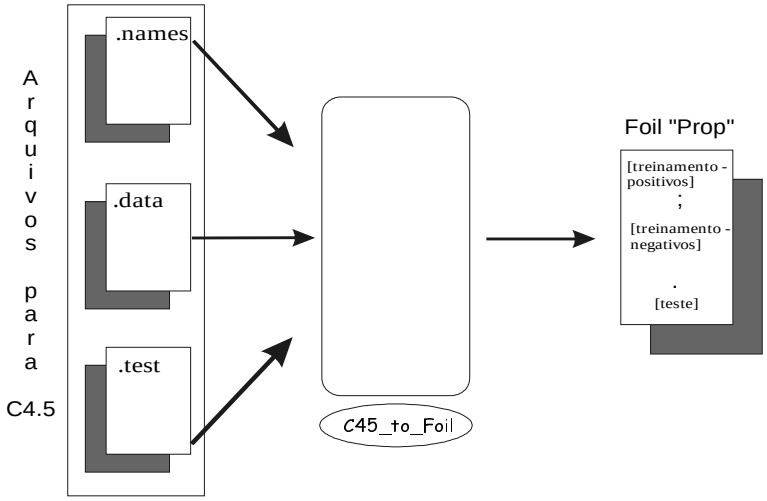


Figura 9: De $\mathcal{C}$4.5 para Foil "Proposicional"

Esse utilitário foi implementado por Quinlan e Cameron-Jones (autores de Foil) e acompanha a distribuição da versão 6.4 de Foil. Antes de efetuar a conversão dos três arquivos na sintaxe de $\mathcal{C}$4.5, algumas informações, descritas a seguir, devem ser acrescentadas manualmente ao arquivo `voyage.names`. Cada linha contendo informação sobre um atributo deve indicar o nome do tipo de dado utilizado pelo atributo. Isto deve ser acrescentado pelo usuário, como se fosse um comentário em $\mathcal{C}$4.5, da seguinte forma:

(*informação do atributo em* $\mathcal{C}$4.5) `| type: nome_tipo`

onde `nome_tipo` é o nome do tipo desse atributo. Um nome distinto deve ser dado a cada tipo de dados distinto, pois será prefixado a cada valor possível do tipo pelo utilitário implementado por Quinlan e Cameron-Jones. Esse nome pode ser o próprio nome do atributo que irá fazer uso

do tipo, ou pode ser uma abreviação. Para o conjunto de dados `voyage` utilizado como exemplo, o arquivo `.names` acrescido dessas informações é mostrado na Figura 10. Deve ser destacado que os atributos `temperature` e `humidity` foram "ligados" pelo tipo `real`, já que ambos atributos possuem o mesmo tipo de dados. Dois nomes diferentes, `o` e `w`, foram utilizados para os atributos `outlook` e `windy`, pois eles assumem valores diferentes. No caso de assumirem os mesmos valores, é conveniente que esses nomes sejam idênticos, já que fornecem uma informação importante aos sistemas de PLI..

```
go, dont_go.

outlook:     sunny, overcast, rain. | type: o
temperature: continuous. | type: real
humidity:    continuous. | type: real
windy:       yes, no. | type: w
```

Figura 10: Arquivo $\mathcal{C}$4.5 `voyage.names` modificado para uso com c45_to_foil

Com estas modificações, os três arquivos na sintaxe de $\mathcal{C}$4.5 podem ser fornecidos como entrada ao c45_to_foil. A saída de c45_to_foil é um arquivo cujo nome é o próprio nome do conjunto de dados concatenado com a extensão `.d`, *i.e.* para o exemplo de viagem o nome é `voyage.d` (Figura 11).

```
#o: *osunny, *oovercast, *orain.
real: continuous.
#w: *wyes, *wno.

is_go(o,real,real,w)
osunny,25,72,wyes
oovercast,23,90,wyes
oovercast,29,78,wno
oovercast,26,75,wno
orain,22,95,wno
orain,21,80,wno
;
osunny,23,95,wno
osunny,30,85,wno
orain,19,70,wyes
orain,23,80,wyes
.

is_go
osunny,28,91,wyes:-
osunny,22,70,wno:+
oovercast,19,65,wyes:+
oovercast,20,87,wyes:-
orain,25,81,wno:+
.
```

Figura 11: Arquivo FOIL "Proposicional" `voyage.d`

No início desse arquivo encontram-se descritos os tipos de dados que compõem os atributos dos

exemplos. Para tipos de dados com valores discretos, o símbolo `#` precede o nome do tipo. Cada valor possível do tipo é então listado, sendo prefixado pelo símbolo `*`. O tipo pré-definido `continuous` indica um tipo de dados numérico, podendo assumir qualquer valor do conjunto dos números reais, assim como na sintaxe de $\mathcal{C}4.5$.

O predicado meta – `is_go(o, real, real, w)` na Figura 11 – que irá representar o conceito a ser aprendido é descrito em seguida. O nome do predicado é composto pelo nome do conceito procurado (no exemplo considerado, `go`) acrescentado de `is_`. Assim, no exemplo o nome do predicado meta é `is_go`. Nesse predicado encontra-se também descrito o tipo de dado de cada um dos seus argumentos.

Depois da descrição dos tipos de dados e do predicado meta, encontram-se os exemplos positivos do conceito. Esses exemplos positivos são separados dos exemplos negativos (contra-exemplos) pelo símbolo "`;`" numa única linha. O símbolo "`.`" sozinho numa única linha indica o final dos exemplos de treinamento a serem utilizados para induzir o conceito. Em seguida, encontram-se os exemplos de teste. Cada um desses exemplos de teste termina com a sequência "`:+`" ou "`:-`", indicando se o exemplo é positivo ou negativo, respectivamente.

Finalmente, o arquivo `voyage.d` gerado pelo c45_to_foil é o arquivo de entrada para as outras três ferramentas por nós implementados, *i.e.* foil2progol, foil2progolmb e foil2foilmb (Figura 1, página 6), descritas nas seções subsequentes.

## 5.3 De Foil "Proposicional" para Progol "Proposicional"

A ferramenta foil2progol (Figura 12) gera três arquivos de entrada para PROGOL, todos eles com extensão `.pl`, com os dados no formato por nós denominado "proposicional".

A fim de identificar cada um desses arquivos, foil2progol usa o nome do arquivo de dados (`voyage` no exemplo considerado) e concatena o nome, respectivamente, com `_training`, `_test` e `_extra_bg_test`. Assim, os três arquivos gerados por foil2progol são:

1. `voyage_training.pl` (Figura 13);
2. `voyage_test.pl` (Figura 14);
3. `voyage_extra_bg_test.pl` (Figura 15).

O arquivo gerado `voyage_training.pl` (Figura 13) contém parâmetros configuráveis[1], entre outras coisas. No PROGOL, esses parâmetros são setados através do predicado pré-definido `set/2`.

[1]Deve ser observado que tanto os parâmetros configuráveis quanto outras informações que devem ser explicitamente definidas para PROGOL, através de declarações específicas, encontram-se implicitamente definidas em FOIL.

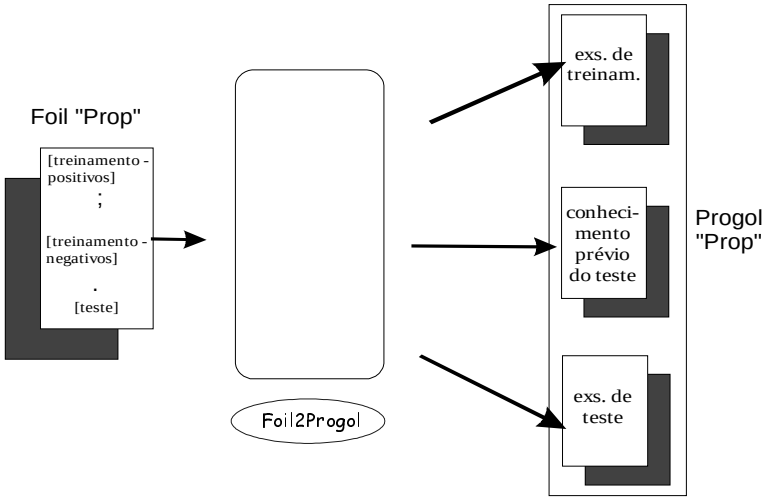


Figura 12: De FOIL "Proposicional" para PROGOL "Proposicional"

foil2progol configura o parâmetro `noise` na primeira linha do arquivo utilizando esse predicado, de modo que cláusulas possam ser encontradas que cubram até 20% de exemplos negativos. Isso é necessário para manter a paridade com o FOIL, que é ajustado por default para procurar cláusulas com no máximo 20% de erro.

Depois de configurar esse parâmetro, declarações de modo dos predicados utilizados são escritas. Essas declarações indicam quais predicados farão parte da cabeça das cláusulas induzidas e quais farão parte do corpo. Para isso, se utiliza um predicado pré-definido `modeh/2` no caso da cabeça e outro, `modeb/2`, para o corpo. Ambos predicados possuem os mesmos argumentos. O primeiro argumento indica o "recall", que limita o número de instanciações alternativas que podem ser consideradas para o predicado declarado no segundo argumento. O "recall"[2] pode variar de 1 até infinito, sendo este último indicado por "`*`". Na prática, "`*`" implica no máximo 100 instanciações. O segundo argumento é uma declaração de predicado que mostra o nome do predicado e os tipos de cada um de seus argumentos. Os argumentos de entrada são precedidos pelo símbolo "`+`" e os de saída pelo símbolo "`-`". Quando o argumento em questão deve assumir um valor constante, a declaração do tipo do argumento deve ser precedido pelo símbolo "`#`". Maiores informações podem ser encontradas em (Roberts, 1997) e (Muggleton e Firth, 1999).

A declaração da cabeça das cláusulas é escrita da seguinte forma: `:- modeh(1, example(+ex))?`. Ou seja, a cabeça será formada por um predicado `example/1` com um argumento de entrada do tipo `ex` e "recall" igual a 1. O argumento desse predicado é um identificador utilizado para marcar os exemplos de forma única. Isso se tornou necessário pois PROGOL, ao contrário de FOIL, não

[2]Observar que o parâmetro "recall" não é o mesmo recall utilizado para definir precisão

```
:- set(noise, 20)?
:- modeh(1, example(+ex))?

:- modeb(1, is_go(+ex, -o, -real, -real, -w))?

:- modeb(1,(+any = #any))?

:- modeb(1,(#real =< +real))?
:- modeb(1,(+real =< #real))?

% Data Types and Values
o(osunny). o(oovercast). o(orain).

w(wyes). w(wno).

real(X) :-
         number(X).

ex(D) :-
         name(D, [_|X]), name(Num, X), int(Num),
         Num >= 0, Num =< 14, !.

% Positive Examples
example(e0). is_go(e0,osunny,25,72,wyes).
example(e1). is_go(e1,oovercast,23,90,wyes).
example(e2). is_go(e2,oovercast,29,78,wno).
example(e3). is_go(e3,oovercast,26,75,wno).
example(e4). is_go(e4,orain,22,95,wno).
example(e5). is_go(e5,orain,21,80,wno).

% Negative Examples
:- example(e6). is_go(e6,osunny,23,95,wno).
:- example(e7). is_go(e7,osunny,30,85,wno).
:- example(e8). is_go(e8,orain,19,70,wyes).
:- example(e9). is_go(e9,orain,23,80,wyes).
```

Figura 13: Arquivo PROGOL "Proposicional" `voyage_training.pl`

consegue induzir o conceito se ele estiver definido por um predicado que contém somente os atributos de cada exemplo. Em outras palavras, PROGOL necessita de conhecimento na forma de conhecimento de fundo para induzir o conceito. A fim de contornar este problema, foil2progol cria o predicado `example/1`, e declara-o como a cabeça das cláusulas a serem procuradas (*i.e.* o conceito a ser induzido) através de uma declaração de modo `modeh/2`. O conhecimento de fundo é definido de forma similar ao predicado meta do arquivo FOIL contendo as instâncias dos valores dos atributos de cada exemplo, além de um argumento adicional (primeiro argumento) que identifica o exemplo, como mostra a primeira declaração `modeb/2` na Figura 13.

Depois de inserir essas declarações, definem-se algumas relações já conhecidas porém de modo especial para melhorar o processo de indução. A primeira é a declaração da relação `=/2`. Através da declaração `:- modeb(1,(+any = #any))`, argumentos de qualquer tipo declarados como va-

riáveis, podem ser substituídas por valores constantes, caso sejam encontrados durante a indução. Para o caso de haver tipos de dados numéricos, outras duas declarações de modo `modeb/2` são incluídas. Ambas as declarações tratam da relação `=</2` de modo a lidar com intervalos numéricos com limites superiores e inferiores. Para declarar o limite inferior o argumento da esquerda na relação `=<` deve ser uma constante, e o da direita uma variável. No caso do limite superior, o argumento da direita é uma constante e o da esquerda uma variável.

Em seguida a essas declarações de modo, os tipos de dados utilizados são descritos. Os tipos de dados discretos são mostrados como fatos de um predicado cujo nome é o próprio tipo. No exemplo considerado (Figura 13), os tipos de dados discretos `o` e `w` são descritos, respectivamente, como {`o(osunny). o(oovercast). o(orain).`} e {`w(wyes). w(wno).`}. Quando há valores numéricos, inclui-se uma descrição intensional do predicado `real/1` como sendo equivalente ao predicado pré-definido do PROGOL, `number/1`. O tipo de dado que expressa o argumento do identificador do exemplo, `ex`, é declarado através do predicado `ex/1` como sendo composto por uma sequência de letras (no caso, a letra `e`) e terminando com um número no intervalo de 0 até $n-1$, onde $n$ é o número total de exemplos de treinamento e teste.

Depois das declarações de tipos de dados, os exemplos de treinamento são escritos, com seus identificadores e os valores dos atributos do respectivo exemplo. Exemplos negativos são indicados pelos símbolos "`:- `", precedendo o identificador. Por exemplo, o primeiro exemplo positivo de treinamento no arquivo de FOIL (Figura 11, página 11), escrito como `is_go(osunny,25,72,wyes)`, é transformado em dois fatos: {`example(e0). is_go(e0,osunny,25,72,wyes).`}, e assim sucessivamente. Da mesma forma, o primeiro exemplo negativo de treinamento de FOIL, escrito como {`is_go(e6,osunny,23,95,wno)`} pode ser transformado para uma declaração com os dois fatos {`:- example(e6). is_go(e6,osunny,23,95,wno).`}, e analogamente para os outros exemplos negativos.

```
% Positive Examples
example(e10).
example(e11).
example(e12).

% Negative Examples
:- example(e13).
:- example(e14).
```

Figura 14: Arquivo PROGOL "Proposicional" `voyage_test.pl`

O segundo arquivo, `voyage_test.pl` (Figura 14), contém os exemplos de teste, declarados através do predicado `example/1` e indicados como positivos ou negativos da mesma forma que no primeiro arquivo. Os valores dos atributos correspondentes a esses exemplos são colocados como conhecimento de fundo no terceiro arquivo gerado, `voyage_extra_bg_test.pl` (Figura 15).

```
% Background for Positive Test Examples
is_go(e10,osunny,22,70,wno).
is_go(e11,oovercast,19,65,wyes).
is_go(e12,orain,25,81,wno).

% Background for Negative Test Examples
is_go(e13,osunny,28,91,wyes).
is_go(e14,oovercast,20,87,wyes).
```

Figura 15: Arquivo PROGOL "Proposicional" `voyage_extra_bg_test.pl`

## 5.4 De Foil "Proposicional" para Progol MB

A ferramenta foil2progolmb (Figura 16) gera dois arquivos de entrada para PROGOL, através do método MB, ambos com extensão `.pl`.

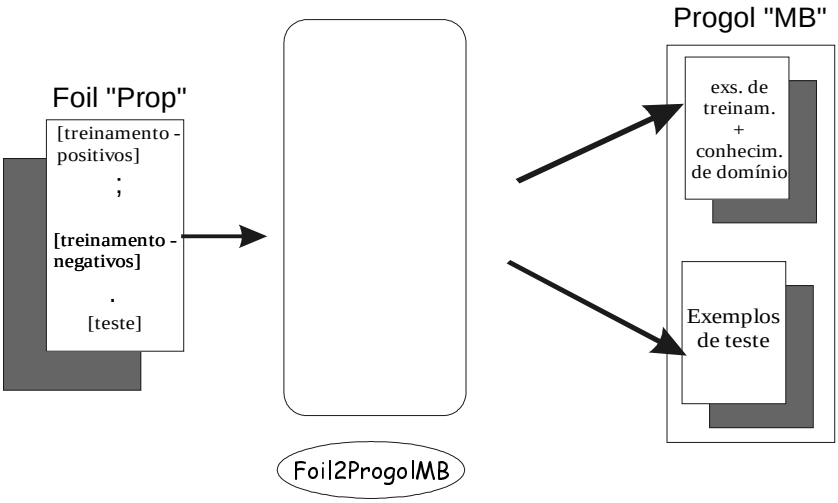


Figura 16: De FOIL "Proposicional" para PROGOL MB

Esses arquivos são criados a partir do arquivo de FOIL em formato "proposicional" gerado pela ferramenta c45_to_foil. Assim como na Subseção 5.3, a fim de identificar cada um desses dois arquivos, foil2progolmb usa o nome do arquivo de dados (`voyage` no exemplo considerado) e concatena, respectivamente, `training_mb` e `test_mb` a ele para formar o nome dos arquivos. A terminação `_mb` distingue esses arquivos daqueles criados pela ferramenta foil2progol, indicando que foram criados com base na proposta de Morik e Brockhausen explicada na Seção 3. Os dois arquivos gerados são:

1. `voyage_training_mb.pl` (Figura 17); e
2. `voyage_test_mb.pl` (Figura 18).

```
:- set(noise, 20)?
:- modeh(1, example(+ex))?

:- modeb(1, a0(+ex, -o))?
:- modeb(1, a1(+ex, -real))?
:- modeb(1, a2(+ex, -real))?
:- modeb(1, a3(+ex, -w))?

:- modeb(1,(+any = #any))?
:- modeb(1,(#real =< +real))?
:- modeb(1,(+real =< #real))?

% Data Types and Values
o(osunny). o(oovercast). o(orain).
w(wyes). w(wno).

real(X) :-
          number(X).

ex(D) :-
        name(D, [_|X]), name(Num, X), int(Num),
        Num >= 0, Num =< 14, !.

% Positive Training Examples
example(e0). a0(e0, osunny). a1(e0, 25). a2(e0, 72). a3(e0, wyes).
example(e1). a0(e1, oovercast). a1(e1, 23). a2(e1, 90). a3(e1, wyes).
example(e2). a0(e2, oovercast). a1(e2, 29). a2(e2, 78). a3(e2, wno).
example(e3). a0(e3, oovercast). a1(e3, 26). a2(e3, 75). a3(e3, wno).
example(e4). a0(e4, orain). a1(e4, 22). a2(e4, 95). a3(e4, wno).
example(e5). a0(e5, orain). a1(e5, 21). a2(e5, 80). a3(e5, wno).

% Negative Training Examples
:- example(e6). a0(e6, osunny). a1(e6, 23). a2(e6, 95). a3(e6, wno).
:- example(e7). a0(e7, osunny). a1(e7, 30). a2(e7, 85). a3(e7, wno).
:- example(e8). a0(e8, orain). a1(e8, 19). a2(e8, 70). a3(e8, wyes).
:- example(e9). a0(e9, orain). a1(e9, 23). a2(e9, 80). a3(e9, wyes).

% Background Knowledge for Positive Test Examples
a0(e10, osunny). a1(e10, 22). a2(e10, 70). a3(e10, wno).
a0(e11, oovercast).  a1(e11, 19). a2(e11, 65). a3(e11, wyes).
a0(e12, orain). a1(e12, 25). a2(e12, 81). a3(e12, wno).

% Background Knowledge for Negative Test Examples
a0(e13, osunny). a1(e13, 28). a2(e13, 91). a3(e13, wyes).
a0(e14, oovercast). a1(e14, 20). a2(e14, 87). a3(e14, wyes).
```

Figura 17: Arquivo PROGOL MB voyage_training_mb.pl

O arquivo gerado `voyage_training_mb.pl` (Figura 17) contém os mesmos parâmetros configuráveis e declarações de modo que foram inseridos no arquivo `voyage_training.pl` com a ferramenta foil2progol, visto na Seção 5.3. A cabeça das cláusulas a serem buscadas é declarada através do predicado pré-definido `modeh/2` e os predicados que podem formar o corpo das cláusulas são declaradas através do predicado pré-definido `modeb/2`. No entanto, onde foil2progol declara apenas um predicado como conhecimento de fundo, foil2progolmb declara vários.

Para isso, o predicado meta descrito no arquivo "proposicional" na sintaxe do FOIL é decomposto em $n$ predicados binários, um para cada atributo que compõe o exemplo. Esses predicados são numerados de a0 até a$n-1$ e declarações de modo de corpo são inseridas para cada um desses novos predicados através do predicado pré-definido `modeb/2`. Os argumentos desses predicados são um identificador de exemplo, criado para cada exemplo, e o valor do atributo representado pelo predicado para o exemplo identificado.

Depois de inserir essas declarações, os tipos de dados utilizados também são declarados como da mesma forma que na ferramenta foil2progol. Os tipos de dados discretos são mostrados como fatos de um predicado cujo nome é o próprio tipo. No exemplo considerado (Figura 17), os tipos de dados discretos `o` e `w` são descritos, respectivamente, como {`o(osunny). o(oovercast). o(orain).`} e {`w(wyes). w(wno).`}. Quando há valores numéricos, a ferramenta foil2progolmb inclui uma descrição intensional do predicado `real/1` como sendo equivalente ao predicado pré-definido do PROGOL, `number/1`. O tipo de dado que expressa o argumento do identificador do exemplo, `ex`, é declarado através do predicado `ex/1` como sendo composto por uma sequência de letras (no caso, a letra '`e`') e um número no intervalo de 0 até $n-1$, onde $n$ é o número total de exemplos de treinamento e teste.

Os exemplos de treinamento são então escritos no arquivo `voyage_training_mb.pl`. Para cada exemplo escreve-se um fato para o identificador do exemplo. Em seguida, utilizando os predicados de atributo definidos na parte de declaração de modos, indica-se o valor para cada atributo. Cada exemplo positivo e negativo é descrito dessa forma. Os exemplos negativos são indicados por '`:-`' antes dos fatos que descrevem seus identificadores. Por exemplo, o primeiro exemplo positivo de treinamento contido no arquivo de FOIL "proposicional" (Figura 11, página 11), e equivalente a `is_go(osunny,25,72,wyes)`, seria transformado para {`example(e0). a0(e0,osunny). a1(e0,25). a2(e0,72). a3(e0,wyes).`}, e assim sucessivamente. De forma semelhante, o primeiro exemplo negativo de treinamento de FOIL "proposicional", equivalente a `is_go(e6,osunny,23,95,wno)` seria acrescentado de '`:-`' e transformado para {`:- example(e6). a0(e6,osunny). a0(e6,23). a0(e6,95). a0(e6,wno).`}, e analogamente para os outros exemplos negativos.

Para manter a representação semelhante àquela da ferramenta foil2foilmb descrita na Subseção 5.5, os valores dos atributos para os exemplos de teste são incluídos juntos com os dos exemplos

de treinamento. Sendo assim, o segundo arquivo, `voyage_test_mb.pl` (Figura 18), contém os identificadores para os exemplos de teste, declarados como fatos do predicado `example/1`. Esse arquivo só pode conter instâncias do predicado de identificação, indicados como positivos ou negativos da mesma forma que no primeiro arquivo, `voyage_training_mb.pl` (Figura 17).

```
% Positive Test Examples
example(e10).
example(e11).
example(e12).

% Negative Test Examples
:- example(e13).
:- example(e14).
```

Figura 18: Arquivo PROGOL MB `voyage_test_mb.pl`

## 5.5 De Foil "Proposicional" para Foil MB

A ferramenta foil2foilmb (Figura 19) gera somente um arquivo, com os dados no formato MB, cuja extensão é `.d`.

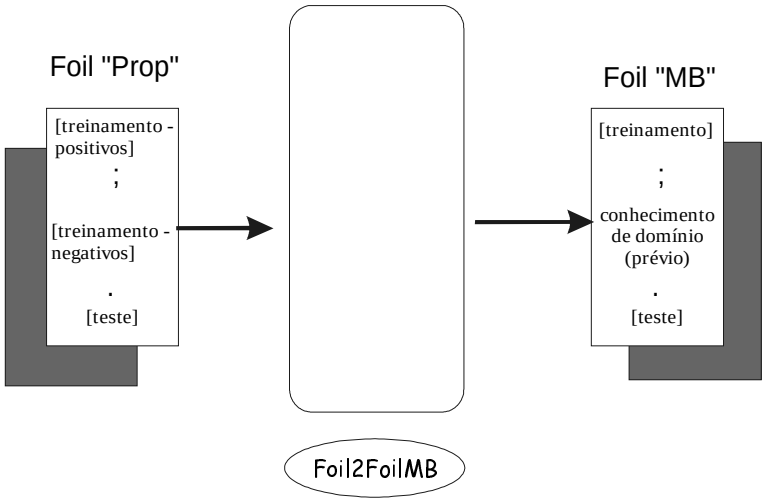


Figura 19: De FOIL "Proposicional" para FOIL MB

O nome do arquivo é composto pelo nome do arquivo de dados e o sufixo `_mb`, para indicar que foi criado com base na proposta de Morik e Brockhausen vista na Seção 3. Assim, o nome do arquivo para o exemplo corrente é `voyage_mb.d`, como pode ser visto na Tabela 4.

```
#o: *osunny, *oovercast, *orain.
real: continuous.
#w: *wyes, *wno.
#example: *e0, *e1, *e2, *e3, *e4, *e5, *e6, *e7, *e8, *e9, *e11, *e12,
*e14, *e10, *e13.

is_go(example)
e0
e1
e2
e3
e4
e5
;
e6
e7
e8
e9
.
*a0(example, o)
e0, osunny
e1, oovercast
e2, oovercast
e3, oovercast
e4, orain
e5, orain
e6, osunny
e7, osunny
e8, orain
e9, orain
e11, osunny
e12, oovercast
e14, orain
e10, osunny
e13, oovercast
.
*a1(example, real)
e0, 25
e1, 23
e2, 29
e3, 26
e4, 22
e5, 21
e6, 23
e7, 30
```

*continua na próxima página*

*continuação da página anterior*

```
e8, 19
e9, 23
e11, 22
e12, 19
e14, 25
e10, 28
e13, 20
.
*a2(example, real)
e0, 72
e1, 90
e2, 78
e3, 75
e4, 95
e5, 80
e6, 95
e7, 85
e8, 70
e9, 80
e11, 70
e12, 65
e14, 81
e10, 91
e13, 87
.
*a3(example, w)
e0, wyes
e1, wyes
e2, wno
e3, wno
e4, wno
e5, wno
e6, wno
e7, wno
e8, wyes
e9, wyes
e11, wno
e12, wyes
e14, wno
e10, wyes
e13, wyes
.

is_go
e11:+
```

*continua na próxima página*

*continuação da página anterior*

```
e12:+
e14:+
e10:-
e13:-
.
```

Tabela 4: Arquivo PROGOL MB `voyage_mb.d`

---

No início do arquivo, são colocados os tipos de dados previamente declarados no arquivo FOIL "proposicional". Além desses, uma lista de identificadores é gerada representando os valores discretos do tipo de dados `ex`, para os identificadores de exemplos.

Depois dos tipos de dados, é descrito o predicado meta cujo nome é o mesmo do predicado meta do arquivo de FOIL "proposicional" (para o conjunto de dados considerado, `is_go`) e contém um único argumento que é o identificador do exemplo. Esse predicado será usado na cabeça das cláusulas encontradas.

Para cada um dos argumentos do predicado meta no arquivo de FOIL "proposicional" são criados novos predicados binários. A declaração de cada um desses predicados é precedido de um `*` para indicar que não constitui um predicado meta e sim conhecimento de fundo que poderá ser incluído no corpo das cláusulas induzidas. O primeiro argumento é sempre o identificador do exemplo e o segundo é o valor, para aquele exemplo, do atributo representado pelo predicado binário. Assim, para o primeiro atributo é criado o predicado `a0` com os argumentos de tipo `ex` e `o`, ou seja, `*a0(ex, o)`, e assim por diante. O predicado novo para um determinado atributo é descrito e em seguida são listados os fatos sobre os valores daquele atributo para os exemplos. Um '`.`' sozinho em uma linha indica o fim da lista de fatos para o atributo. Esse processo se repete até que todos os atributos tenham sido listados.

Como FOIL não possui uma interface para ler os exemplos de teste separadamente da geração da hipótese, facilidade esta disponível em PROGOL, os valores dos atributos para os exemplos de teste devem ser colocados juntos com os de treinamento de modo que sejam carregados juntos na memória. Finalmente, é novamente escrito o nome do predicado meta (`is_go`) para indicar que os exemplos de teste serão listados. Somente os identificadores dos exemplos de teste são colocados, um em cada linha. A sequência "`:+`" ou "`:-`" no final da linha indica que o exemplo é positivo ou negativo, respectivamente, em relação ao conceito procurado. Um '`.`' sozinho em uma linha indica o fim da lista de exemplos de teste e do arquivo de dados.

# 6 Considerações Finais

Em aprendizado de máquina indutivo, é necessário ter linguagens de descrição para descrever os dados e os conceitos aprendidos através dos sistemas de aprendizado utilizados para induzir os conceitos. Essas linguagens podem ser divididas em dois tipos: *baseadas em atributos* ou proposicionais e *relacionais*.

Sistemas de aprendizado baseados em linguagens proposicionais possuem uma série de técnicas já bem estabelecidos que permitem adquirir conhecimento de uma forma rápida e concisa. No entanto, não conseguem aprender relações intrínsecas aos dados. Para fazer isso, alguns sistemas de aprendizado fazem uso de linguagens relacionais, baseadas em lógica de primeira ordem, as quais permitem aprender essas relações incluindo conhecimento de fundo sobre o conceito sendo induzido.

Em princípio, qualquer problema que possa ser resolvido utilizando AM proposicional também pode ser resolvido com AM relacional. Para tanto, é necessário transformar os exemplos no formato atributo-valor para o formato relacional. O processo de conversão descrito neste trabalho possibilita essa transformação dos exemplos através de uma representação direta do atributo-valor bem como através do método MB.

Todas as ferramentas por nós criadas para o processo de conversão foram implementadas utilizando a linguagem de programação PERL (Schwartz e Christiansen, 1997). Decidiu-se utilizar PERL pelas facilidades que essa linguagem apresenta para extrair e transformar texto contido em arquivos de texto, através do uso de expressões regulares. Com isso, a conversão dos dados pôde ser realizada de uma forma simples. O código-fonte de cada uma das ferramentas é apresentado no Apêndice A.

# Referências


Baranauskas, J. A. e Monard, M. C. (2000a). An Unified Overview of Six Supervised Symbolic Machine Learning Inducers. Technical Report 103, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_103.ps.zip.

Baranauskas, J. A. e Monard, M. C. (2000b). Reviewing some Machine Learning Concepts and Methods. Technical Report 102, ICMC – USP, São Carlos, SP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/Rt_102.ps.

Cameron-Jones, R. M. e Quinlan, J. R. (1993). First Order Learning, Zeroth Order Data. Technical Report 468, The University of Sydney. 15 pg.

Clark, P. e Boswell, R. (1991). Rule Induction with $\mathcal{CN}2$: Some Recent Improvements. In Kodratoff, Y., editor, *Proceedings of the 5th European Conference (EWSL 91)*, pages 151–163. Springer Verlag.

Michalski, R. S. (1983). A Theory and Methodology of Inductive Learning. In Michalski, R. S., Carbonell, J. G., e Mitchell, T. M., editores, *Machine Learning: An Artificial Intelligence Approach*, volume I, pages 83–184, Palo Alto, CA. Tioga.

Morik, K. e Brockhausen, P. (1997). A Multistrategy Approach to Relational Knowledge Discovery in Databases. *Machine Learning*, 27:287–312.

Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing*, 13:245–286.

Muggleton, S. e Firth, J. (1999). *CProgol4.4: a tutorial introduction*. University of York MLG Group. ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/progol4.4/.

Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, 1:81–106.

Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266.

Quinlan, J. R. (1993). *C4.5 Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, California.

Roberts, S. (1997). *An Introduction to Progol*. ML-Group, University of York. ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/progol4.4/.

Schwartz, R. L. e Christiansen, T. (1997). *Learning Perl, 2nd Edition*. O'Reilly and Associates.

Shapiro, E. Y. (1983). *Algorithmic Program Debugging*. The MIT Press.

# A Código-Fonte das Ferramentas

Neste Apêndice, é listado o código-fonte, escrito em PERL, das ferramentas por nós implementadas no processo de conversão de dados.

## A.1 foil2progol

```
#!/usr/bin/perl
#-*-Perl-*-
#
# foil2progol -- going for the gold: FOIL data file to Progol data files
#                converter
# Copyright (C) 1999, 2000 Chandler Caulkins <chandler@icmc.sc.usp.br>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

# Keywords: FOIL PROGOL format data conversion
# $Id: foil2progol, v 0.01 2000/01/20 chandler

##########################################################################

$foil2progol_version='0.01';
$author_name= scalar(getpwuid($<));
$date=scalar(localtime(time));

($program_name = $0) =~ s@.*/@@;
@SAVED_ARGV=@ARGV;

# name of the FOIL dataset file being processed
$_ = $SAVED_ARGV[0];
# strips the extension from the filename
s/\..+//;
```

```
$dataset_name = $_;

require('getopts.pl');
Getopts('hv:');

#######################################################################
### display version if requested.
&show_version if ($opt_v);

### display usage if requested.
&show_usage if ($opt_h);

# flags internos
$numeric_values = 0;
$period_before = 0;
$semicolon_before = 0;

#######################################################################
### read input.

while (defined($line = <>)) {
    chop($line);

    ### ignore empty lines, as well as any declarations of numeric types.
    if (($line !~ /^$/) && ($line !~ /continuous/)) {
        $_ = $line;

        ### remove leading and trailing blanks
        s/^\s+//;
        s/\s+$//;

        ### "uncapitalize" any capital letters
        ### (this is one way it can be done in PERL)
        tr/A-Z/a-z/;

        ### definition of datatypes and their values
        ### (which are then placed in the array "datatypes")
        if (/:.+\./) {
            s/([\*\#\.])//g;
            s/:/,/;
            push @datatypes, $_;
        }

        ### extraction of predicate definition
        elsif (/\(.+\)/) {
            s/\)//;
```

```
    ### quicklist is used to separate the predicate name
    ### from the predicate arguments
    @quicklist = split(/\(/);
    $head_name = $quicklist[0];
    @head_args = split(/,/, $quicklist[1]);

    $head_declaration = $head_name."(\+ex";
    for $i (0 .. $#head_args) {
        $head_declaration .= ", \-" . $head_args[$i];
    }
    $head_declaration .= ")";
}

elsif (/^\.$/) {
   $period_before = 1;
   $semicolon_before = 0;
}

elsif (/^\;$/) {
   $semicolon_before = 1;
}

### all other lines should be attribute-values of the examples
### (for some reason empty lines were slipping thru the
###  condition at the very beginning)
else {
    # when a semicolon has been found, the beginning of the
    # negative examples set has arrived...
    if ($semicolon_before){
       push @train_negative, $_;
    }
    # the first time a period is found, the test examples
    # can be saved...
    elsif ($period_before){
       if (/:-$/){
          s/:-//;
          push @test_negative, $_;
       }
       if (/:\+$/){
          s/:\+//;
          push @test_positive, $_;
       }
    }
    # after all these conditions, the positive training examples
    # can (finally) be saved
    else {
       push @train_positive, $_;
```

```
            }
        }
    }

    if ($line =~ /continuous/) {
        $numeric_values = 1;
    }

}

# Creating and opening files for training...
$filename = $dataset_name.'_training.pl';
$fileopen = open (TRAININGFILE, ">$filename");

# ... test examples ...
$filename = $dataset_name.'_test.pl';
$fileopen = open (TESTFILE, ">$filename");

# ... and now, extra background for the test files!
$filenamebg = $dataset_name.'_extra_bg_test.pl';
$fileopen = open (TESTBGFILE, ">$filenamebg");

# With the information all extracted, a standard
# header can be created for the Prolog training
# data file.
&print_header($numeric_values);

### Next, valid values for discrete datatypes are
### placed in the training datafile
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Data Types and Values \n\n";
for $i (0 .. $#datatypes) {
    @typeinfo = split(/,\s/, $datatypes[$i]);

    for $j (1 .. $#typeinfo){
        print TRAININGFILE "$typeinfo[0]($typeinfo[$j])\. ";
    }
    print TRAININGFILE "\n\n";
}
if ( $numeric_values ) {
    print TRAININGFILE "real(X) :- \n";
    print TRAININGFILE "\t number(X)\.\n\n";
}

print TRAININGFILE "ex(D) :- \n";
print TRAININGFILE "\t name(D, [_|X]), name(Num, X), int(Num), \n";
$n_examples = $#train_positive + 1 + $#train_negative + 1 +
```

```
  $#test_positive + 1 + $#test_negative;
print TRAININGFILE "\t Num >= 0, Num =< $n_examples, !\.\n\n";

### The positive examples are written to the training datafile ...
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Positive Examples \n\n";
for $i (0 .. $#train_positive){
    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Positive Training Example \#$i\n\n";
    print TRAININGFILE "example(e$i).\n";
    print TRAININGFILE "$head_name\(e$i,$train_positive[$i]\).\n";
}

print TRAININGFILE "\n";

### ... then the negative examples are written ...
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Negative Examples \n\n";
for $i (0 .. $#train_negative){
    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Negative Training Example \#$i\n\n";
    $id = $i + $#train_positive+1;
    print TRAININGFILE ":- example(e$id).\n";
    print TRAININGFILE "$head_name\(e$id,$train_negative[$i]\).\n";
}

print TRAININGFILE "\n";

### ... and finally the test examples are written to the test
### examples file; first the positive examples ...
print TESTFILE "%%%%%%%%%%%%%%%%%%%%\n";
print TESTFILE "% Positive Examples \n\n";
print TESTBGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TESTBGFILE "% Background for Positive Test Examples \n\n";
for $i (0 .. $#test_positive){
    print TESTFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TESTFILE "% Positive Test Example \#$i\n\n";
    $id = $i + $#train_positive + 1 + $#train_negative + 1;
    print TESTFILE "example(e$id).\n";
    print TESTBGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TESTBGFILE "% BG for Positive Test Example \#$i\n\n";
    print TESTBGFILE "$head_name(e$id,$test_positive[$i]).\n";
}

print TESTFILE "\n\n";
print TESTBGFILE "\n\n";
```

```
# ... and then the negative ones.
print TESTFILE "%%%%%%%%%%%%%%%%%%%%\n";
print TESTFILE "% Negative Examples \n\n";
print TESTBGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TESTBGFILE "% Background for Negative Test Examples \n\n";
print TESTFILE "%%%%%%%%%%%%%%%%%%%%%%\n";
print TESTFILE "% Negative Examples \n\n";
for $i (0 .. $#test_negative){
    print TESTFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TESTFILE "% Negative Test Example \#$i\n\n";
    $id = $i + $#train_positive + 1 + $#train_negative + 1 + $#test_positive + 1;
    print TESTFILE ":- example(e$id).\n";
    print TESTBGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TESTBGFILE "% BG for Negative Test Example \#$i\n\n";
    print TESTBGFILE "$head_name(e$id,$test_negative[$i]).\n";
}

### Last but not least, the files are closed ...
close(TRAININGFILE);
close(TESTFILE);
close(TESTBGFILE);

### ... and the program exits successfully (or so we pray)!
exit 0;

#######################################################################
### create standard header for Prolog training data file.

sub print_header
{

    local ($has_numbers) = @_;

    # comments for posterity
    print TRAININGFILE '%%%%%%%%%%%%%%%%%%%
```

```
    # for numeric values
    if ($has_numbers > 0){
        # Generates 1 lower bound for input variable.
        print TRAININGFILE ":- modeb(1,(#real =< +real))?\n";
        # Generates 1 upper bound for input variable.
        print TRAININGFILE ":- modeb(1,(+real =< #real))?\n\n";

    }

}

######################################################################
### print program usage and exit.

sub show_usage
{

    print <<EOF;

Usage: $program_name [-hv] [file]

       -h             display this help and exit.
       -v             display version number and exit.

EOF

    exit 0;
}

######################################################################
### print program version and exit.

sub show_version
{
    printf("%s %s\n",$program_name,$foil2progol_version);
    exit 0;
}

### foil2progol ends here.
```

## A.2 foil2progolmb

```
#! /usr/bin/perl
#-*-Perl-*-
#
# foil2progolmb.perl -- for transforming a FOIL data file to a Progol
# format with the "representation" given in [MorikBrockhausen 1997]
#
# Copyright (C) 1999 Chandler Caulkins <chandler@icmc.sc.usp.br>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

# Keywords: FOIL PROGOL format data conversion
# $Id: foil2progolmb,v 0.01 2000/01/20 chandler

#######################################################################

$foil2progolmb_version='0.01';
$author_name= scalar(getpwuid($<));
$date=scalar(localtime(time));

($program_name = $0) =~ s@.*/@@;
@SAVED_ARGV=@ARGV;

# name of the FOIL dataset file being processed
$_ = $SAVED_ARGV[0];
# strips the extension from the filename
s/\..+//;
$dataset_name = $_;

require('getopts.pl');
Getopts('hv:');

#######################################################################
```

```
### display version if requested.
&show_version if ($opt_v);

### display usage if requested.
&show_usage if ($opt_h);

# flags internos
$numeric_values = 0;
$period_before = 0;
$semicolon_before = 0;
$id = 0;

#######################################################################
### read input.

while (defined($line = <>)) {
    chop($line);

    ### ignore empty lines, as well as any declarations of numeric types.
    if (($line !~ /^$/) && ($line !~ /continuous/)) {
        $_ = $line;

        ### remove leading and trailing blanks
        s/^\s+//;
        s/\s+$//;

        ### "uncapitalize" any capital letters
        ### (this is one way it can be done in PERL)
        tr/A-Z/a-z/;

        ### definition of datatypes and their values
        ### (which are then placed in the array "datatypes")
        if (/:.+\./) {
            s/([\*\#\.])//g;
            s/:/,/;
            push @datatypes, $_;
        }

        ### extraction of predicate definition
        elsif (/\(.+\)/) {
            s/\)//;

            ### quicklist is used to separate the predicate name
            ### from the predicate arguments
            @quicklist = split(/\(/);
            $head_name = $quicklist[0];
            @head_args = split(/,/, $quicklist[1]);
```

```
        $head_declaration = "example(+ex)";
        for $i (0 .. $#head_args) {
            push @mode_declarations,"a$i(\+ex, \-$head_args[$i])";
        }
    }

    elsif (/^\.$/) {
       $period_before = 1;
       $semicolon_before = 0;
    }

    elsif (/^\;$/) {
       $semicolon_before = 1;
    }

    ### all other lines should be attribute-values of examples
    ### (for some reason empty lines were slipping thru the
    ###  condition at the very beginning)
    else {
        # when a semicolon has been found, the beginning of the
        # negative examples set has arrived...
        if ($semicolon_before){
           push @train_negative, $_;
        }
        # the first time a period is found, the test examples
        # can be saved...
        elsif ($period_before){
           if (/:-$/){
              s/:-//;
              push @test_negative, $_;
           }
           if (/:\+$/){
              s/:\+//;
              push @test_positive, $_;
           }
        }
        # after all these conditions, the positive training examples
        # can (finally) be saved
        else {
           push @train_positive, $_;
        }
    }
}

if ($line =~ /continuous/) {
    $numeric_values++;
```

```
    }

}

# Creating and opening files for training...
$filename = $dataset_name.'_mb_training.pl';
$fileopen = open (TRAININGFILE, ">$filename");

# ... and test examples
$filename = $dataset_name.'_mb_test.pl';
$fileopen = open (TESTFILE, ">$filename");

# With the information already extracted, a standard
# header can be created for the Prolog training
# data file.
&print_header($numeric_values);

### Next, valid values for discrete datatypes are
### placed in the training datafile
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Data Types and Values \n\n";
for $i (0 .. $#datatypes) {
    @typeinfo = split(/,\s/, $datatypes[$i]);

    for $j (1 .. $#typeinfo){
        print TRAININGFILE "$typeinfo[0]($typeinfo[$j])\. ";
    }
    print TRAININGFILE "\n\n";
}

if ( $numeric_values ) {
    print TRAININGFILE "real(X) :- \n";
    print TRAININGFILE "\t number(X)\.\n\n";
}

print TRAININGFILE "ex(D) :- \n";
print TRAININGFILE "\t name(D, [_|X]), name(Num, X), int(Num), \n";
$n_examples = $#train_positive + 1 + $#train_negative + 1 +
  $#test_positive + 1 + $#test_negative;
print TRAININGFILE "\t Num >= 0, Num =< $n_examples, !\.\n\n";

### The positive examples are written to the training datafile ...
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Positive Training Examples \n\n";
for $i (0 .. $#train_positive){
    @example = split(/,/, $train_positive[$i]);
```

```
    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Positive Training Example \#$i\n";
    print TRAININGFILE "example(e$i).\n\n";
    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Background Knowledge\n";
    for $j (0 .. $#example){
        print TRAININGFILE "a$j(e$i, $example[$j]).\n";
    }
    print TRAININGFILE "\n";
}

print TRAININGFILE "\n";

### ... then the negative examples are written ...
print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Negative Training Examples \n\n";
for $i (0 .. $#train_negative){
    @example = split(/,/, $train_negative[$i]);

    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Negative Training Example \#$i\n";
    $id = $i + $#train_positive+1;
    print TRAININGFILE ":- example(e$id).\n\n";
    print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TRAININGFILE "% Background Knowledge\n";
    for $j (0 .. $#example){
        print TRAININGFILE "a$j(e$id, $example[$j]).\n";
    }
    print TRAININGFILE "\n";
}

### ... and finally the test examples are written to the test
### examples file; first the positive examples ...
print TESTFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TESTFILE "% Positive Test Examples \n\n";

print TRAININGFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
print TRAININGFILE "% Background Knowledge for Positive Test Examples \n\n";

for $i (0 .. $#test_positive){
    @example = split(/,/, $test_positive[$i]);

    $id = $i + $#train_positive + 1 + $#train_negative + 1;

    print TESTFILE "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n";
    print TESTFILE "% Positive Test Example \#$i\n\n";
    print TESTFILE "example(e$id).\n";
```

```
    print TRAININGFILE "%%%
```

```
sub print_header
{

    local ($has_numbers) = @_;
    local ($n_examples) = 0;

    # comments for posterity
    print TRAININGFILE '%%%
```

```
EOF

    exit 0;
}

#######################################################################
### print program version and exit.

sub show_version
{
    printf("%s %s\n",$program_name,$foil2progolmb_version);
    exit 0;
}

### foil2progol ends here.
```

## A.3 foil2foilmb

```
#! /usr/local/bin/perl
#-*-Perl-*-
#
# foil2foilmb.perl -- for transforming a FOIL data file to a FOIL
# format with the the "representation" given in [MorikBrockhausen 1997]
#
# Copyright (C) 1999, 2000 Chandler Caulkins <chandler@icmc.sc.usp.br>
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

# Keywords: FOIL PROGOL format data conversion
# $Id: foil2foilmb,v 0.01 2000/01/20 chandler

#######################################################################

$foil2foilmb_version='0.01';

($program_name = $0) =~ s@.*/@@;
@SAVED_ARGV=@ARGV;

# name of the FOIL dataset file being processed
$_ = $SAVED_ARGV[0];
# strips the extension from the filename
s/\..+//;
$dataset_name = $_;

require('getopts.pl');
Getopts('hv:');

#######################################################################
### display version if requested.
&show_version if ($opt_v);
```

```
### display usage if requested.
&show_usage if ($opt_h);

# flags internos
$period_before = 0;
$semicolon_before = 0;
$id = 0;

#######################################################################
### read input.

# Creating and opening file for training...
$filename = $dataset_name.'.mb.d';
$fileopen = open (TRAININGFILE, ">$filename");

if ($fileopen) {
while (defined($line = <>)) {
    chop($line);

    ### ignore empty lines, as well as any declarations of numeric types.
    if ($line !~ /^$/) {
        $_ = $line;

        ### remove leading and trailing blanks
        s/^\s+//;
        s/\s+$//;

        ### "uncapitalize" any capital letters
        ### (this is one way it can be done in PERL)
        tr/A-Z/a-z/;

        ### definition of datatypes and their values
        ### (which are then placed in the array "datatypes")
        if (/.+:.+\./) {
            print TRAININGFILE "$_\n";
        }

        ### extraction of predicate definition
        elsif (/\(.+\)/) {
            s/\)//;

            ### quicklist is used to separate the predicate name
            ### from the predicate arguments
            @quicklist = split(/\(/);
            $head_name = $quicklist[0];
            @head_args = split(/,/, $quicklist[1]);
```

```
            $head_declaration = $head_name."(example)";
            for $i (0 .. $#head_args) {
                push @mode_declarations,"a$i(example, $head_args[$i])";
            }
        }

        elsif (/^\;$/) {
           $semicolon_before = 1;
        }

        elsif (/^\.$/) {
           $period_before = 1;
           $semicolon_before = 0;
        }

        ### all other lines should be attribute-values of examples
        ### (for some reason empty lines were slipping thru the
        ###  condition at the very beginning)
        else {
            # when a semicolon has been found, the beginning of the
            # negative examples set has arrived...
            if ($semicolon_before){
               # @example = split($_);
               push @train_negative, "e".$id++.",$_";
            }
            # the first time a period is found, the test examples
            # can be saved...
            elsif ($period_before){
               if (/:-$/){
                  s/:-//;
                  push @test_negative, "e".$id++.",$_";
               }
               if (/:\+$/){
                  s/:\+//;
                  push @test_positive, "e".$id++.",$_";
               }
            }
            # after all these conditions, the positive training examples
            # can "finally" be saved
            else {
               push @train_positive, "e".$id++.",$_";
            }
        }
    }
}
```

```
print TRAININGFILE "\#example: ";

for $i (0 .. $#train_positive){
    @example = split(/,/, $train_positive[$i]);
    print TRAININGFILE "\*$example[$0], ";
}

for $i (0 .. $#train_negative){
    @example = split(/,/, $train_negative[$i]);
    print TRAININGFILE "\*$example[$0], ";
}

for $i (0 .. $#test_positive){
    @example = split(/,/, $test_positive[$i]);
    print TRAININGFILE "\*$example[$0], ";
}

for $i (0 .. $#test_negative-1){
    @example = split(/,/, $test_negative[$i]);
    print TRAININGFILE "\*$example[$0], ";
}

@example = split(/,/, $test_negative[$#test_negative]);
print TRAININGFILE "\*$example[$0].\n";

print TRAININGFILE "\n$head_declaration\n";

### The positive examples are written to the training datafile ...
for $i (0 .. $#train_positive){
    @example = split(/,/, $train_positive[$i]);
    print TRAININGFILE "$example[$0]\n";
}

print TRAININGFILE "\;\n";

### ... then the negative examples are written ...
for $i (0 .. $#train_negative){
    @example = split(/,/, $train_negative[$i]);
    print TRAININGFILE "$example[$0]\n";
}

print TRAININGFILE "\.\n";

# Now for the background knowledge...

$no_of_attributes = $#example;
for $i (1 .. $no_of_attributes) {
```

```
    print TRAININGFILE "\*$mode_declarations[($i-1)]\n";
    for $tr (0 .. $#train_positive) {
        @example = split(/,/, $train_positive[$tr]);
        print TRAININGFILE "$example[0], $example[$i]\n";
    }
    for $tr (0 .. $#train_negative) {
        @example = split(/,/, $train_negative[$tr]);
        print TRAININGFILE "$example[0], $example[$i]\n";
    }
    for $te (0 .. $#test_positive) {
        @example = split(/,/, $test_positive[$te]);
        print TRAININGFILE "$example[0], $example[$i]\n";
    }
    for $te (0 .. $#test_negative) {
        @example = split(/,/, $test_negative[$te]);
        print TRAININGFILE "$example[0], $example[$i]\n";
    }
    print TRAININGFILE "\.\n";
}

print TRAININGFILE "\n$head_name\n";

### ... and finally the test examples are written;
### first the positive examples ...
for $i (0 .. $#test_positive){
    @example = split(/,/, $test_positive[$i]);
    print TRAININGFILE "$example[0]\:\+\n";
}

# ... and then the negative ones.
for $i (0 .. $#test_negative){
    @example = split(/,/, $test_negative[$i]);
    print TRAININGFILE "$example[0]\:\-\n";
}

print TRAININGFILE ".\n";

### Lastly, the file is closed ...
close(TRAININGFILE);

### ... and the program exits successfully!
exit 0;

} # end of "if $fileopen"

#######################################################################
### print program usage and exit.
```

```
sub show_usage
{

    print <<EOF;

Usage: $program_name [-hv] [file]

       -h             display this help and exit.
       -v             display version number and exit.

EOF

    exit 0;
}

#######################################################################
### print program version and exit.

sub show_version
{
    printf("%s %s\n",$program_name,$foil2foilmb_version);
    exit 0;
}

### foil2foilmb ends here.
```