UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

ISSN 0103-2569

**Descrição da Implementação do Módulo Conversor Kaeru para Transformar Dados no Formato Atributo-Valor para o Formato Relacional do Sistema de Programação Lógica Indutiva Aleph**


**Mariza Ferro**
**Maria Carolina Monard**


**Nº 234**

RELATÓRIOS TÉCNICOS

**São Carlos – SP**
**Jul./2004**

# Descrição da Implementação do Módulo Conversor *Kaeru* para Transformar Dados no Formato Atributo-Valor para o Formato Relacional do Sistema de Programação Lógica Indutiva Aleph *

**Mariza Ferro**
**Maria Carolina Monard**

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação e Estatística
Laboratório de Inteligência Computacional
Caixa Postal 668, 13560-970 - São Carlos, SP, Brasil
e-mail: {mariza, mcmonard}@icmc.usp.br

---

**Resumo**

A Programação Lógica Indutiva (PLI) é uma sub área de Aprendizado de Máquina (AM) que pode ser definida como a intersecção de AM indutivo e programação lógica. Para aproveitar a expressividade desses sistemas relacionais quando as bases de dados estão num formato originalmente proposicional (atributo-valor), é necessário transformar essa descrição proposicional em uma descrição relacional equivalente. Surge assim a necessidade de automatizar tanto quanto possível esse processo de transformação. Esse é o objetivo do módulo conversor *Kaeru*, apresentado neste trabalho, que é um dos módulos do ambiente PLI em desenvolvimento no Laboratório de Inteligência Computacional (LABIC). O módulo conversor *Kaeru* automatiza o processo de transformação de bases de dados proposicionais para o formato relacional para ser utilizado por sistemas de PLI. Além disso, o módulo *Kaeru* cria os arquivos necessários para executar o sistema de PLI Aleph.

*Palavras-Chave:* Aprendizado de Máquina, Programação Lógica Indutiva.

---

**Julho 2004**

*Trabalho realizado com auxílio da CAPES

# Sumário

1 Introdução 1

2 O Projeto DISCOVER 3
2.1 A Biblioteca de Classes Discover Object Library - DOL . . . . . . . . . . . 4
2.2 O Ambiente Computacional SNIFFER . . . . . . . . . . . . . . . . . . . . . 5
2.3 A Sintaxe Padrão para Conjuntos de Dados no Formato Atributo-Valor . 5
2.4 Tipos de Dados Implementados . . . . . . . . . . . . . . . . . . . . . . . . 7

3 O Módulo Conversor *Kaeru* 9
3.1 Arquitetura do Módulo Conversor *Kaeru* . . . . . . . . . . . . . . . . . . . 9
3.2 O Sistema Aleph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2.1 Funcionamento Básico . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2.2 Declarações de modo . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.2.3 Tipos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.2.4 Declaração dos Determinations . . . . . . . . . . . . . . . . . . . . 14
3.2.5 Exemplos Positivos e Negativos . . . . . . . . . . . . . . . . . . . . 14
3.2.6 Parâmetros . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.2.7 Características Importantes . . . . . . . . . . . . . . . . . . . . . . 15

4 Descrição do Módulo Conversor *Kaeru* 16
4.1 Formato dos Arquivos de Entrada . . . . . . . . . . . . . . . . . . . . . . . 16
4.1.1 Arquivo `.param` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
4.1.2 Arquivo `.bk` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2 Formato do Arquivo de Saída `.b` . . . . . . . . . . . . . . . . . . . . . . . 21

5 Conclusão 25

A Parâmetros do Aleph 26

Referências 33

# Lista de Figuras

1 Interação entre filtros, sintaxes e bibliotecas, (Chiara and Monard, 2003) 4
2 Interação do módulo conversor *Kaeru*, Aleph, a biblioteca DOL e o Ambiente SNIFFER PLI . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3 Arquivos de entrada e saída do módulo conversor *Kaeru* . . . . . . . . . 16

# Lista de Tabelas

1 Subconjunto de dados `voyage` . . . . . . . . . . . . . . . . . . . . . . . . 6
2 Exemplo de arquivo de declaração de atributos: `voyage.names` . . . . . . 7
3 Exemplo de arquivo de declaração de dados: `voyage.data` . . . . . . . . 7
4 Exemplo de arquivo de declaração de parâmetros do Aleph: `voyage.param` 17
5 Exemplo de arquivo de declaração de parâmetros do módulo conversor *Kaeru*: `voyage.bk` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

# Lista de Abreviaturas

**AM** Aprendizado de Máquina

**IA** Inteligência Artificial

**LABIC** Laboratório de Inteligência Computacional

**LPO** Linguagem de Primeira Ordem

**PLI** Programação Lógica Indutiva

# 1 Introdução

Atualmente há um grande interesse na extração de conhecimento de grandes volumes de dados armazenados nas bases de dados de empresas, hospitais e áreas tais como a biologia. Esse é o objetivo do processo de mineração de dados que utiliza, entre outros, sistemas de Aprendizado de Máquina (AM) para descobrir padrões e conhecimentos úteis dos dados.

Entre as diferentes dimensões que distinguem os sistemas de aprendizado, uma das mais significativas é a descrição de objetos (exemplos) e conceitos (hipóteses) induzidos. O formato atributo-valor, ou proposicional, é a linguagem de descrição de objetos mais freqüentemente utilizada em AM, embora sua baixa capacidade de expressão impeça a representação de objetos estruturados, bem como a relação entre objetos ou entre seus componentes. Assim, aspectos relevantes dos objetos, que de alguma maneira poderiam caracterizar o conceito que está sendo aprendido, não podem ser representados utilizando o formato atributo-valor. Essas limitações comprometem a aplicabilidade de algoritmos proposicionais em domínios nos quais a estrutura interna dos objetos de estudo é de grande importância, tais como química, linguagem natural e medicina, entre outros.

Outra linguagem de representação que pode ser utilizada na descrição de objetos e conceitos é a linguagem relacional. Os sistemas de aprendizado que utilizam esse tipo de linguagem de representação são chamados de sistemas de aprendizado relacional; nesses sistemas objetos podem ser descritos em termos de seus componentes e relações entre esses componentes. No aprendizado relacional a linguagem utilizada para descrever exemplos, conhecimento do domínio e descrições dos conceitos é, tipicamente, um subconjunto da Linguagem de Primeira Ordem (LPO) (Lavrač and Džeroski, 1994). O uso de LPO como linguagem de representação em sistemas de aprendizado permite que relações ou predicados possam ser induzidos. Esses sistemas possuem uma alta expressividade para representar conceitos e a importante habilidade de representar conhecimento do domínio. Entretanto, a maior expressividade da linguagem faz com que o espaço dos conceitos passíveis de serem aprendidos seja aumentado. Além da alta expressividade e uso de conhecimento do domínio, os sistemas de aprendizado relacional têm a vantagem de expressar seu conhecimento de uma forma diretamente intelegível aos humanos, característica muito importante quando o objetivo é a extração e descoberta de conhecimento (Muggleton, 1999).

Os sistemas de aprendizado que induzem hipóteses na forma de programas lógicos por meio do uso de conhecimento do domínio e uma linguagem de descrição baseada em LPO,

ou relacional, são chamados sistemas de Programação Lógica Indutiva (PLI). Por meio de pesquisas em PLI busca-se superar as limitações da representação proposicional. A área de pesquisa em PLI tem sido fortemente influenciada pela teoria de aprendizado computacional e, recentemente, pela descoberta de conhecimento em bases de dados, o que tem conduzido para o desenvolvimento de novas técnicas de mineração de dados relacional (Lavrač and Flach, 2001).

Entretanto, para aproveitar a expressividade desses sistemas relacionais quando os exemplos estão descritos no formato atributo-valor, é necessário transformar essa descrição proposicional para uma descrição relacional equivalente. Surge assim a necessidade de automatizar tanto quanto possível esse processo de transformação. Esse é o objetivo do módulo conversor *Kaeru* [1] apresentado neste trabalho. Em outras palavras o módulo conversor *Kaeru*, implementado na linguagem *Perl* (Wall et al., 1996), tem como objetivo auxiliar no processo de transformação de bases de dados proposicionais para o formato relacional, automatizando e padronizando esse processo de transformação. Após realizada a transformação para o formato relacional, a base de dados pode ser utilizada por sistemas de PLI. Além de transformar bases de dados proposicionais para o formato relacional o módulo conversor *Kaeru* cria os arquivos necessários para executar o sistema de PLI Aleph. Esse sistema foi escolhido pois ele permite simular diversos sistemas de PLI existentes, facilitando assim as pesquisas nessa área. O módulo conversor *Kaeru* está integrado ao projeto DISCOVER (Baranauskas and Batista, 2000), em desenvolvimento no Laboratório de Inteligência Computacional (LABIC). O projeto DISCOVER é um ambiente computacional, para descoberta de conhecimento em bases de dados, no qual estão integrados algoritmos de aprendizado proposicional implementados pela comunidade, bem como ferramentas específicas desenvolvidas por pesquisadores do LABIC, as quais oferecem funcionalidades voltadas para o aprendizado de máquina proposicional, mineração de dados e mineração de textos. Assim, o módulo *Kaeru* tem como objetivo permitir a incorporação de algoritmos de PLI no ambiente DISCOVER.

Este trabalho está organizado da seguinte forma: na Seção 2 são apresentados o projeto DISCOVER, a biblioteca de classes DOL e o ambiente SNIFFER implementados no DISCOVER, além da sintaxe padrão utilizada nesse projeto para representar conjuntos de dados no formato atributo-valor; na Seção 3 são explicados a arquitetura e funcionamento do módulo conversor *Kaeru*, como ele interage com o DISCOVER e com o sistema de PLI Aleph; na Seção 4 é apresentado o formato adotado para os arquivos de entrada e saída do módulo conversor *Kaeru*; por fim, na Seção 5, são apresentadas as conclusões deste trabalho.

[1] kaeru é uma palavra de origem japonesa que significa converter ou transformar

# 2 O Projeto DISCOVER

No Laboratório de Inteligência Computacional (LABIC) [2] são desenvolvidos diversos trabalhos de pesquisa cooperativos. Porém, por falhas de comunicação e documentação ocorriam algumas sobreposições de implementações. Esses fatores levaram alguns pesquisadores do LABIC a propor a implementação do DISCOVER (Baranauskas and Batista, 2000). O objetivo principal do projeto DISCOVER é reduzir problemas de implementação e integração de ferramentas e algoritmos para apoiar todas as etapas do processo de descoberta de conhecimento, oferecendo funcionalidades voltadas para aprendizado de máquina, mineração de dados e mineração de textos. O projeto DISCOVER pode ser entendido como um conjunto de métodos para manipular dados e conhecimento por meio de sintaxes padrão para a representação de dados e do conhecimento induzido por diferentes algoritmos de aprendizado, bem como por bibliotecas que oferecem um conjunto de funcionalidades básicas para tais manipulações.

O DISCOVER já conta com vários ambientes, tais como o ambiente Discover Learning Environment - DLE, composto pela biblioteca de classes Discover Object Library - DOL, e o ambiente para gerenciamento de experimentos SNIFFER, que foram desenvolvidos para as diferentes tarefas de pré-processamento de dados no formato atributo-valor e análise dos classificadores induzidos por algoritmos de aprendizado utilizados no processo de Mineração de Dados (Batista and Monard, 2003). Novas funcionalidades estão sendo especificadas, principalmente para representação de regras de regressão (Dosualdo, 2003), regras de associação (Melanda, 2002) e o módulo para PLI, apresentado neste trabalho. Um *framework* para a integração dos componentes do ambiente DISCOVER foi proposto por Prati (2003) utilizando *software patterns*, no qual os componentes são integrados por meio de uma linguagem baseada em XML.

Na Figura 1 é mostrado, de uma forma simplificada, como os filtros, sintaxes e bibliotecas interagem uns com os outros para o caso de algoritmos simbólicos proposicionais de aprendizado supervisionado. Os dados, no formato atributo-valor da sintaxe padrão do DISCOVER, podem ser submetidos a diversos processos de pré-processamento já implementados. Após, esses dados são convertidos para a sintaxe padrão do algoritmo de aprendizado que o usuário deseja executar. O conhecimento (classificador) simbólico induzido por esse algoritmo de aprendizado, é posteriormente convertido para a sintaxe padrão da linguagem de representação de conceitos proposicionais do DISCOVER. Esse conhecimento pode ser posteriormente submetido a diversos processos de pós-processamento de conhecimento já implementados no DISCOVER. As linhas tracejadas na Figura 1 mostram os

[2] http://labic.icmc.sc.usp.br

processos que utilizam a biblioteca de classes DOL e rotinas implementadas no ambiente DLE do DISCOVER.

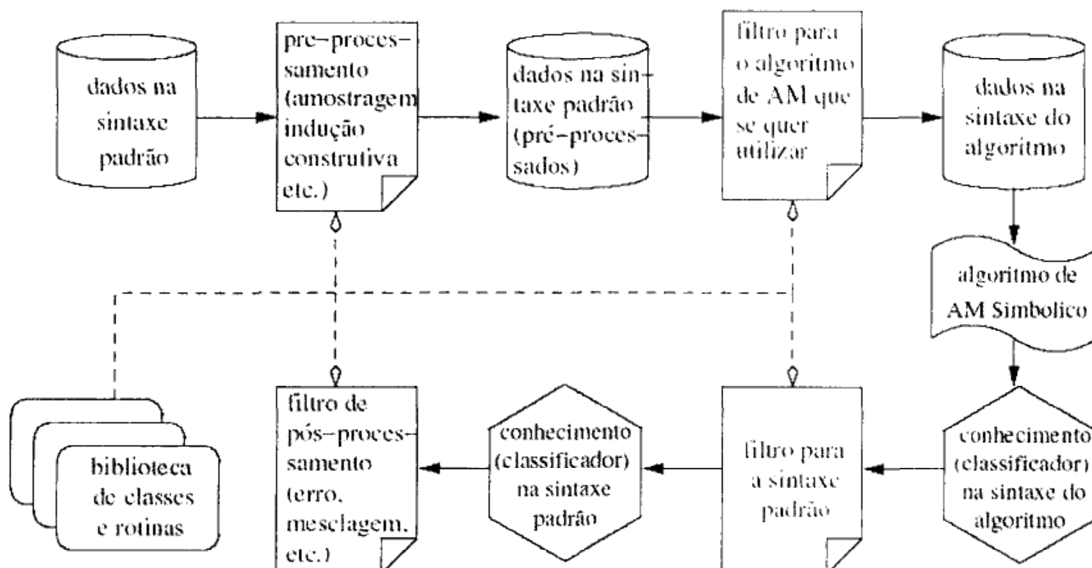


Figura 1: Interação entre filtros, sintaxes e bibliotecas, (Chiara and Monard, 2003)

## 2.1 A Biblioteca de Classes Discover Object Library - DOL

A biblioteca DOL é uma biblioteca orientada a objeto baseada em padrões de projeto (Gamma et al., 1995) . As classes dessa biblioteca implementam as tarefas de manipulação e gerenciamento de dados mais comuns em pré-processamento, tais como gerenciamento de diferentes sintaxes de arquivos de dados e atributos, amostragens, métodos de *resampling*, estatísticas descritivas, normalizações de dados, entre outros (Batista and Monard, 2003). O objetivo da DOL é dar suporte à criação de novos métodos de pré-processamento de dados e ser uma biblioteca de métodos de pré-processamento.

O conjunto de métodos da DOL permite ao desenvolvedor uma forma simples de ter acesso aos dados. Os dados podem estar tanto disponíveis em arquivos texto como em tabelas de banco de dados relacionais. Os dados são carregados em uma estrutura e uma grande quantidade de métodos fornece acesso aos dados, possibilitando sua manipulação. Em seguida, eles podem ser novamente armazenados em arquivos texto, em diferentes sintaxes, ou carregados em uma tabela de banco de dados relacional. Entre as principais funcionalidades da biblioteca DOL pode-se citar: manipulação de atributos e dados, integração com diversos sistemas de aprendizado de máquina, integração com

sistemas gerenciadores de banco de dados, filtros de exemplos, estatísticas descritivas e correlações, além de diversos métodos de *resampling*. A biblioteca DOL foi desenvolvida em uma arquitetura modular na qual cada módulo é constituído de uma ou mais classes que realizam um conjunto bem definido de tarefas. A biblioteca possui um módulo central chamado Core, o qual carrega um conjunto de dados em uma estrutura e disponibiliza, atualmente, mais de 60 métodos capazes de consultar e manipular essa estrutura. O módulo Core é o único que precisa obrigatoriamente ser carregado por uma aplicação que utiliza a biblioteca. Já os demais módulos são carregados apenas quando for necessário.

## 2.2 O Ambiente Computacional SNIFFER

O ambiente computacional SNIFFER é um ambiente de gerenciamento de avaliações e comparações experimentais de algoritmos de AM. O ambiente computacional SNIFFER automatiza a avaliação experimental, e está atualmente integrado com diversos sistemas de Aprendizado de Máquina tais como $\mathcal{C}4.5$ (Quinlan, 1988), $\mathcal{C}4.5$ Rules (Quinlan, 1987), ID3 (Quinlan, 1986), $\mathcal{CN}2$ (Clark and Boswell, 1991) e NewId (Boswell, 1990). Dessa forma, é possível comparar o desempenho de novos métodos de pré-processamento de dados aplicados a diversos sistemas de aprendizado.

O ambiente SNIFFER complementa a biblioteca DOL pois permite que métodos de pré-processamento de dados desenvolvidos sejam avaliados e comparados experimentalmente de uma forma rápida e segura (Batista and Monard, 2003).

## 2.3 A Sintaxe Padrão para Conjuntos de Dados no Formato Atributo-Valor

No desenvolvimento de pesquisas na área de extração de conhecimento, realizar uma investigação que envolve extrair conhecimento de vários conjuntos de dados utilizando diversos sistemas de aprendizado é, normalmente, muito trabalhoso. Isso porque, sistemas de aprendizado usam sintaxes diferentes para representar os dados (Batista, 2001) e atributos nos respectivos arquivos de entrada. Por esse motivo, no DISCOVER, foi decidido adotar uma sintaxe padrão para representação de dados no formato atributo-valor e representação do conhecimento extraído por algoritmos de AM simbólicos proposicionais (Prati et al., 2001), bem como implementar bibliotecas que oferecem funcionalidades sobre essas sintaxes padrão (Batista and Monard, 2003). A partir dessa sintaxe padrão, a qual foi dada o nome de DSX (*Discover Dataset Sintax*), é possível utilizar a biblioteca de classes

DOL, para converter um arquivo de dados para a sintaxe utilizada por diversos sistemas de aprendizado.

A sintaxe padrão de dados no formato atributo-valor utiliza arquivos do tipo texto para declarar os atributos, seus respectivos tipos e os valores que esses atributos assumem para um determinado conjunto de exemplos. Dado um conjunto de exemplos, um arquivo com extensão `.names` é utilizado para declarar os atributos, enquanto os valores que esses atributos assumem para esse conjunto de exemplos são declarados em um arquivo com a extensão `.data`. Os dois arquivos devem possuir o mesmo nome, se diferenciando apenas pela extensão. Opcionalmente, também podem ser usados arquivos de dados com as extensões `.test`, com casos rotulados de teste para medir o erro de classificação, `.validation` com casos rotulados para validação de modelos e `.cases` com casos não rotulados para serem rotulados por um classificador.

A fim de ilustrar, considere o subconjunto de seis exemplos do conjunto de dados `voyage` (Quinlan, 1988), mostrados na Tabela 1

| Exemplo | outlook | temperature | humidity | windy | class |
|---|---|---|---|---|---|
| 1 | sunny | 25 | 72 | yes | go |
| 2 | sunny | 28 | 91 | yes | dont_go |
| 3 | overcast | 23 | 90 | yes | go |
| 4 | overcast | 29 | 78 | no | go |
| 5 | rain | 22 | 95 | no | go |
| 6 | rain | 19 | 70 | yes | dont_go |

Tabela 1: Subconjunto de dados `voyage`

Cada exemplo encontra-se rotulado com a classe `go` ou `dont_go`, e é descrito pelos seguintes atributos:

- `outlook`: que pode assumir os valores `sunny`, `overcast` ou `rain`;
- `temperature`: que pode assumir um valor numérico;
- `humidity`: que pode assumir um valor numérico;
- `windy`: que pode assumir os valores `yes` ou `no`.

Na Tabela 2 na página oposta é apresentado o arquivo de declaração de atributos `voyage.names` na sintaxe DSX para o conjunto de dados artificial `voyage`.

Na Tabela 3 na próxima página é apresentado o arquivo de declaração de dados correspondente aos exemplos descritos na Tabela 1. Nesse arquivo são declarados, na mesma

```
class.          | Class Attribute

| Attributes
outlook:        nominal (sunny, overcast, rain).
temperature:    integer.
humidity:       integer.
windy:          nominal (yes, no).
class:          nominal (go, dont_go).
```

Tabela 2: Exemplo de arquivo de declaração de atributos: `voyage.names`

```
sunny, 25, 72, yes, go
sunny, 28, 91, yes, dont_go
overcast, 23, 90, yes, go
overcast, 29, 78, no, go
rain, 22, 95, no, go
rain, 19, 70, yes, dont_go
```

Tabela 3: Exemplo de arquivo de declaração de dados: `voyage.data`

ordem dos atributos no arquivo `.names`, os valores que esses atributos assumem para um determinado exemplo.

## 2.4 Tipos de Dados Implementados

Na DOL foram definidos alguns tipos de dados que podem ser associados aos identificadores de atributos. Nesta seção é apresentada apenas uma breve descrição e exemplo desses tipos de dados. Maiores detalhes podem ser encontrados em (Batista and Monard, 2003).

**Nominal** O tipo nominal é utilizado para declarar atributos que podem assumir um grupo restrito de valores.
`Atrib1: nominal(circulo,quadrado).`

**Enumerated** O tipo enumerated é bastante semelhante ao tipo nominal. A principal diferença é que no tipo enumerated é possível definir uma ordem que os valores

podem assumir.

`Atrib2: enumerated(pequeno,médio,grande).`

**Integer** O tipo integer é utilizado para declarar um atributo que assume valores inteiros.

`Atrib3: integer.`

**Real** O tipo real é semelhante ao integer exceto pelo real armazenar números com ou sem parte fracionária.

`Atrib4: real.`

**String** Um atributo do tipo string pode assumir como valores seqüencias de caracteres de tamanho indefinido e com quaisquer caracteres, inclusive (\n).

`Atrib5: string.`

**Date** Este tipo de dado é capaz de armazenar datas no formato aaaa/mm/dd (ano, mês e dia).

`Atrib6: date.`

**Time** Este tipo de dado permite declarar atributos que podem conter um horário e devem estar no formato hh:mm:ss (horas, minutos e segundos).

`Atrib7: time.`

# 3 O Módulo Conversor *Kaeru*

No processo de extração de conhecimento de base de dados, caso esses dados estejam no formato atributo-valor, é possível utilizar, entre outros, sistemas de aprendizado simbólico proposicionais para extrair conhecimento dos dados. Entretanto, a linguagem de descrição de hipóteses desses sistemas não permite descobrir relações implícitas nos dados, ou seja, conhecimento relacional.

O primeiro passo para tentar descobrir conhecimento relacional utilizando dados que estão no formato atributo-valor, é transformar esses dados para o formato relacional. Após essa transformação, esses dados podem ser utilizados por algoritmos de PLI para induzir conhecimento relacional. O objetivo do módulo conversor *Kaeru*, descrito neste trabalho, é automatizar a transformação de dados no formato atributo-valor na sintaxe padrão do DISCOVER, descrita na Seção 2.3 na página 5, para o formato relacional. Além disso, foi também decidido construir as diretivas necessárias para utilizar o sistema Aleph. Esse sistema, descrito em maiores detalhes na Seção 3.2, permite simular diversos algoritmos de PLI, facilitando assim o trabalho de pesquisa na área.

## 3.1 Arquitetura do Módulo Conversor *Kaeru*

O módulo conversor *Kaeru* foi implementado para ser integrado no DISCOVER, seguindo os mesmos padrões de implementação. O módulo utiliza as facilidades da biblioteca de classes DOL e o ambiente para gerenciamento de experimentos SNIFFER PLI. O SNIFFER PLI, atualmente em desenvolvimento, é semelhante ao ambiente SNIFFER — Seção 2.2 na página 5, mas para gerenciamento de experimentos relacionados, exclusivamente, a algoritmos de PLI. Ele está integrado ao sistema Aleph e acrescido de funcionalidades específicas para manipular dados relacionais. A Figura 2 na página seguinte mostra como esses ambientes estão integrados.

A entrada do módulo conversor *Kaeru* consiste dos arquivos `.names` e `.data` no formato DSX — Seção 2.3, os quais são inicialmente manipulados e verificados pelos métodos da biblioteca DOL. Além desses dois arquivos, também são necessários mais dois arquivos de entrada com extensão `.param` e `.bk`. O arquivo com extensão `.param` contém a declaração dos parâmetros a serem utilizados pelo Aleph, enquanto que o arquivo com extensão `.bk` contém informações, em um formato padrão descrito na Seção 4 na página 16, a serem utilizadas pelo módulo conversor *Kaeru*. Os quatro arquivos de entrada devem ter o mesmo nome, se diferenciando apenas pela sua extensão.

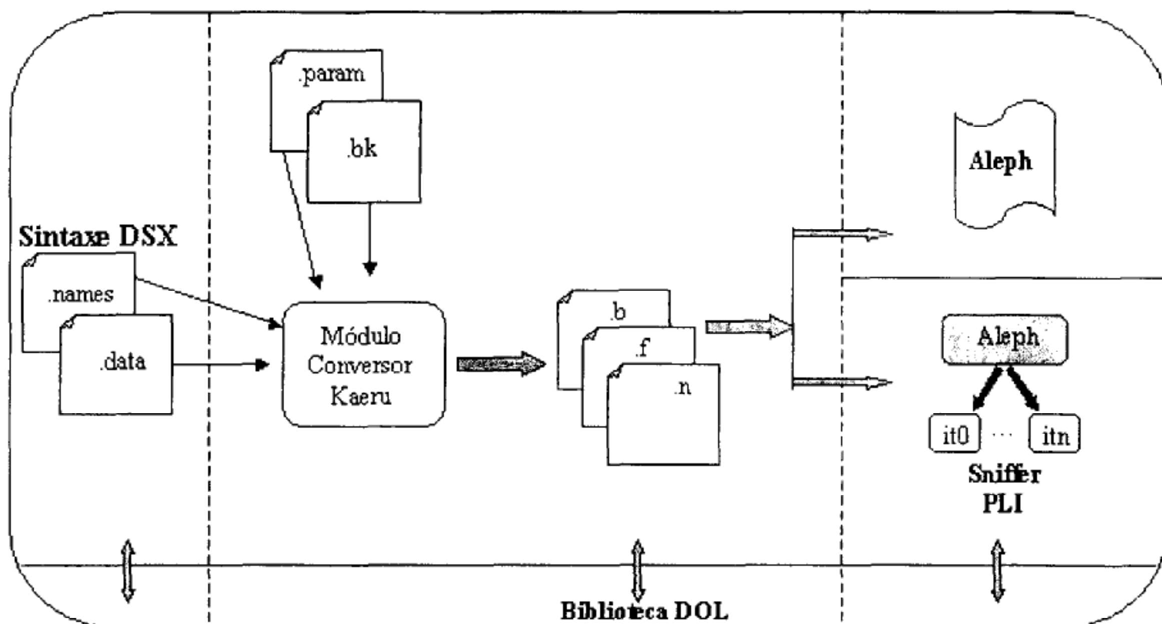


Figura 2: Interação do módulo conversor *Kaeru*, Aleph, a biblioteca DOL e o Ambiente SNIFFER PLI

Deve ser observado que antes de fazer a conversão, é verificada a existência, no arquivo `.names`, de um atributo especial denominado `att_id`, cujo valor identifica cada um dos exemplos no arquivo `.data`. Caso esse atributo não exista, ele é inserido internamente pelo conversor, tanto no arquivo `.names` quanto no arquivo `.data`[3].

Utilizando a informação desses quatro arquivos de entrada, são construídos os três arquivos necessários para a execução do sistema Aleph. Esses três arquivos, com extensão `.b`, `.f` e `.n`, contém as seguintes informações:

- `.b` conhecimento do domínio;
- `.f` exemplos positivos;
- `.n` exemplos negativos.

Após a conversão automática dos arquivos, o usuário pode fazer uma intervenção manual nesses arquivos, pois, eles são gerados com uma configuração padrão para o sistema Aleph. Caso o usuário precisar realizar execuções utilizando outras configurações do Aleph, ele pode modificar tais parâmetros, ou mesmo adicionar mais conhecimento do domínio específico para o domínio da sua aplicação.

[3]Nesse caso o valor do atributo `att_id` do primeiro exemplo é 0, do segundo exemplo é 1 e assim por diante.

O usuário pode simplesmente utilizar o sistema Aleph para induzir conhecimento (hipótese) dos dados ou pode, também, utilizar o Ambiente SNIFFER PLI para medir o erro da hipótese induzida usando *k-fold cross-validation*, ou criando conjuntos específicos de treinamento e teste para executar o Aleph.

## 3.2 O Sistema Aleph

O sistema Aleph (***A** **L**earning **E**ngine for **P**roposing **H**ypotheses*) (Srinivasan, 2000) foi desenvolvido com o intuito de ser um protótipo para explorar idéias de PLI. Atualmente Aleph emula funcionalidades de vários outros sistemas de PLI, tais como: Progol (Muggleton, 1995), FOIL (Quinlan, 1990), FORS (Karalič and Bratko, 1997), Indlog (Camacho, 1994), MIDOS (Wrobel, 1997), Tilde (Blockeel, 1997) e WARMR (Dehaspe and De Raedt, 1997).

O Aleph possui uma poderosa linguagem de representação que permite representar expressões complexas e incorporar novo conhecimento do domínio com facilidade. Além dessas características comuns aos sistemas de PLI, o Aleph possui outras características muito interessantes tais como: permitir escolher qual a ordem de geração das regras (geral para o específico ou específico para o geral); mudar a função de avaliação (entropia, precisão, Laplace entre outras) e a ordem de busca (*Hill-Climbing*, *Branch-and-Bound* entre outras). Essas características do sistema Aleph, além de ser um *software* de uso livre, motivaram a escolha desse sistema para integrar ao DISCOVER.

### 3.2.1 Funcionamento Básico

O Aleph segue um procedimento muito simples, que pode ser descrito pelos seguintes quatro passos:

1. Seleciona um exemplo para ser generalizado. Se não existirem mais exemplos pára;
2. Constrói a cláusula mais específica, que implique no exemplo selecionado no passo anterior e que esteja de acordo com as restrições da linguagem. Normalmente é uma cláusula com muitos literais chamada de cláusula *bottom*, construída com resolução inversa. Esse passo é chamado de saturação;
3. Busca a cláusula mais geral que a cláusula *bottom*. O padrão é do tipo *Branch-and-Bound*. Esse passo é chamado de redução;

4. Adiciona a melhor cláusula a teoria e todos os exemplos redundantes são removidos e retorna ao passo 1.

O Aleph utiliza a informação contida nos seguintes três arquivos para construir uma teoria:

- `file.b`: contém a informação sobre o conhecimento do domínio (intensional e extensional), as restrições de busca e linguagem (declarações de modo, determinations), restrições de tipos e parâmetros;
- `file.f`: contém os exemplos positivos (somente fatos *ground*);
- `file.n`: contém os exemplos negativos (somente fatos *ground*). Pode não existir, já que Aleph pode realizar aprendizado utilizando somente exemplos positivos.

### 3.2.2 Declarações de modo

As declarações de modo no arquivo `file.b`, descrevem as relações (predicados) entre objetos de dados e tipos desses dados. Essas declarações permitem também informar ao Aleph se a relação pode ser utilizada na cabeça (declarações `modeh`) ou no corpo (declarações `modeb`) das regras geradas. Na declaração de modo também são descritos o tipo dos argumentos de cada predicado e as possíveis instanciações desses argumentos. As declarações tem o seguinte formato:

```
mode(Numero_Chamadas,Modo).
```

O `Numero_Chamadas`, ou como é mais conhecido, o *recall*, determina o limite do número de *instanciações* alternativas de um predicado. Uma instanciação do predicado é uma substituição de tipos para cada variável ou constante. O *recall* pode ser qualquer número positivo $n \geq 1$ ou '*'.

Se é conhecido que há somente um certo número de soluções para uma particular instanciação é possível informar isso pelo *recall*. Por exemplo, para uma declaração do predicado `progenitor_de(P,F)` poderia ser dado um *recall* igual a 2, pois todas as pessoas terão no máximo dois progenitores, enquanto que o *recall* seria 4 para uma declaração do predicado `avo(A,N)`. O *recall* '*' é utilizado quando não há limites para o número de soluções para uma instanciação. Por exemplo, em princípio, não há limites para o número de ancestrais que uma pessoa pode ter. A segunda parte da declaração do `mode`, denominada de Modo, indica o formato do predicado que será utilizado e tem o seguinte formato:

```
predicado(Tipo_Argumento1,Tipo_Argumento2,...,Tipo_Argumenton).
```

Por exemplo, para o aprendizado da relação `tio_de(T,S)` com conhecimento do domínio descrito pelas relações `progenitor_de(P,F)` e `irmao_de(I1,I2)` as declarações de modo podem ser:

```
:- modeh(1,tio_de(+pessoa,+pessoa)).
:- modeb(*,progenitor_de(-pessoa,+pessoa)).
:- modeb(*,progenitor_de(+pessoa,-pessoa)).
:- modeb(*,irmao_de(+pessoa,-pessoa)).
```

A declaração `modeh` indica o predicado que irá compor a cabeça das regras. No exemplo, `modeh` informa que a cabeça das regras deve ser `tio_de(T,S)`, onde `T` e `S` são do tipo `pessoa`. O símbolo '+' que aparece antes do tipo indica que esse argumento do predicado é uma *variável*. Assim, a cabeça pode ter a forma `tio_de(T,S)`, mas não `tio_de(ana,maria)`. O símbolo '-' indica uma variável de saída e o símbolo # indica que esse argumento poderia ser uma constante.

A segunda declaração, `modeb`, indica que as regras geradas podem ter no corpo o predicado `progenitor_de(P,F)`, no qual `P` e `F` são do tipo `pessoa`. A primeira declaração `modeb` pode ser usada para adicionar `progenitor_de/2` no corpo das regras e introduzir um ou mais `progenitor/2` de um filho (observar que `Numero_Chamadas` está instânciado com '*'). Similarmente, a segunda declaração `modeb` permite que `progenitor_de/2` seja usado no corpo para encontrar um ou mais filhos de um progenitor, e a última declaração pode ser usada no corpo para encontrar um ou mais irmãos de um indivíduo.

### 3.2.3 Tipos

Os tipos devem ser especificados para cada argumento dos predicados utilizados na construção de uma hipótese. Para o Aleph os tipos são só nomes e a declaração de tipos nada mais é que um conjunto de fatos. Por exemplo, a descrição de objetos do tipo `pessoa` poderia ser

```
pessoa(jane).
pessoa(henrique).
pessoa(sueli).
pessoa(junior).
```

### 3.2.4 Declaração dos Determinations

O Aleph utiliza a declaração `determination`, para declarar os predicados que podem ser usados para construir uma hipótese. Essa declaração tem o formato:

```
determination(Pred_Alvo/Aridadea,Pred_Corpo/Aridadec).
```

O primeiro argumento é o nome do predicado alvo e sua aridade, isto é, o predicado que irá aparecer na cabeça da regra induzida. O segundo argumento consiste do nome e aridade de um predicado que pode aparecer no corpo da cláusula. Por exemplo, para o aprendizado da relação `tio_de(T,S)`, uma declaração seria:

```
:-determination(tio_de/2,progenitor_de/2).
```

Tipicamente, são feitas muitas declarações para um predicado alvo, as quais correspondem aos predicados relevantes para construção de uma cláusula (regra). Caso nenhuma declaração para `determination` seja apresentada, o Aleph não constrói nenhuma cláusula.

### 3.2.5 Exemplos Positivos e Negativos

Exemplos positivos e negativos do conceito a ser aprendido devem ser apresentados na forma extensional (fatos) nos arquivos com extensão `.f` e `.n` respectivamente. Por exemplo, para o aprendizado do conceito `tio_de(T,S)` poderiamos ter a declaração dos seguintes exemplos positivos no arquivo com extensão `.f`:

```
tio_de(jane,henrique).
tio_de(sueli,henrique).
...
```

E os seguintes exemplos negativos no arquivo com extensão `.n`:

```
tio_de(julia,junior).
tio_de(julia,henrique).
...
```

### 3.2.6 Parâmetros

O Aleph permite ao usuário determinar uma variedade de restrições no espaço das possíveis hipóteses a serem aprendidas, bem como na busca realizada nesse espaço de possíveis hipóteses, pela definição de novos valores dos parâmetros disponíveis. O predicado `set/2` permite ao usuário definir o valor do parâmetro `Parametro`

```
set(Parametro,Valor)
```

O valor corrente de um determinado parâmetro pode ser obtido usando

```
setting(Parametro,Valor)
```

enquanto que

```
noset(Parametro)
```

retorna o valor desse parâmetro para seu valor padrão. No Apêndice A são descritos alguns dos parâmetros disponíveis no Aleph .

### 3.2.7 Características Importantes

O Aleph possui algumas características importantes que permitem a modificação de cada um dos passos apresentados na Seção 3.2.1 na página 11, tais como:

- Seleção de exemplos: ao invés de selecionar apenas um exemplo inicial para ser generalizado, é possível escolher mais que um. Caso sejam selecionados $n$ exemplos iniciais; Para cada um deles é criada uma cláusula *bottom*; depois do passo de redução a melhor entre as $n$ reduções é adicionada à teoria;

- Construção da cláusula mais específica: especifica o estágio em que a cláusula *bottom* é construída;

- Busca: a busca por cláusulas pode ser alterada para diferentes estratégias de busca, funções de avaliação e operadores de refinamento;

- Remoção de redundantes: exemplos cobertos podem ser retidos para dar uma melhor estimativa das cláusulas obtidas.

# 4 Descrição do Módulo Conversor *Kaeru*

Na Figura 3 são apresentados os quatro arquivos de entrada utilizados pelo módulo *Kaeru*, com extensão `.names`, `.data`, `.param` e `.bk`, necessários para realizar a conversão de um conjunto de exemplos no formato atributo-valor para a representação relacional utilizada pelo Aleph.

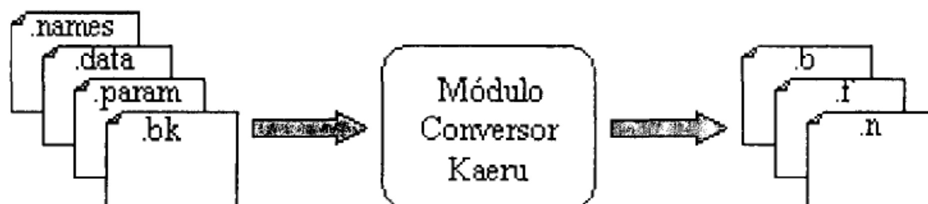


Figura 3: Arquivos de entrada e saída do módulo conversor *Kaeru*

Após a conversão, são gerados os três arquivos de saída `.b`, `.f` e `.n`, necessários para executar o Aleph — Seção 3.2 na página 11. Como mencionado anteriormente todos os arquivos, tanto de entrada quanto de saída, são arquivos texto com o mesmo nome, se diferenciando apenas pela sua extensão. Os arquivos de entrada `.names` e `.data`, que descrevem os exemplos no formato atributo-valor, devem estar no formato DSX do DISCOVER, descrito na Seção 2 na página 3. O formato utilizado para definir a informação contida nos outros dois arquivos é apresentado a seguir.

## 4.1 Formato dos Arquivos de Entrada

O arquivo de entrada com extensão `.param` é utilizado para declaração dos parâmetros do sistema Aleph, e o arquivo com extensão `.bk` é utilizado para declaração dos parâmetros utilizados pelo próprio módulo conversor *Kaeru*. Ambos arquivos são descritos com detalhes a seguir.

### 4.1.1 Arquivo `.param`

Neste arquivo podem ser declarados qualquer um dos parâmetros do conjunto de parâmetros do Aleph, na mesma sintaxe usada pelo Aleph, como ilustrado na Tabela 4 na página oposta.

Cada declaração se inicia com o sinal `:-`, seguido pelo comando `set`, o parâmetro que se deseja modificar e um ponto no fim da declaração. Cada declaração deve ser feita em uma

```
:- set(minpos,3).
:- set(clauselength,5).
```

Tabela 4: Exemplo de arquivo de declaração de parâmetros do Aleph: `voyage.param`

linha do arquivo. Caso nenhuma declaração seja feita, o Aleph será executado com todos os valores padrão dos seus parâmetros. Por exemplo, na Tabela 4, a primeira declaração indica que o limite inferior do número de exemplos positivos que devem ser cobertos por uma cláusula deve ser igual a três. A segunda declaração indica que o limite superior do número de literais em uma cláusula é cinco.

Neste arquivo podem ser declarados qualquer número de parâmetros do Aleph, os quais serão integrados ao arquivo `.b`, descrito na Seção 4.2 na página 21.

### 4.1.2 Arquivo `.bk`

Este arquivo contém declarações utilizadas pelo módulo conversor *Kaeru*, as quais são necessárias para a criação do arquivo `.b`. Na Tabela 5 é apresentado um exemplo desse arquivo para o conjunto de dados `voyage` (Quinlan, 1988) descrito na Seção 2 na página 3.

```
head{}
body{}
positive{go}
negative{dont_go}
```

Tabela 5: Exemplo de arquivo de declaração de parâmetros do módulo conversor *Kaeru*: `voyage.bk`

A primeira declaração define o predicado a ser aprendido — predicado alvo. A segunda declaração define os predicados que podem participar do corpo do predicado alvo. A terceira declaração define as classes positivas dos exemplos e a quarta as classes negativas. Observar que essas declarações não tem ponto no fim.

Para gerar os arquivos de saída `.b`, `.f` e `.n`, é necessário que todos esses parâmetros sejam declarados nesse arquivo. Segue uma descrição detalhada desses parâmetros.

#### Declaração head

Este é o primeiro parâmetro que deve ser declarado no arquivo `.bk` e define a cabeça do predicado a ser induzido pelo sistema, ou seja, qual é o predicado alvo. No sistema de PLI Aleph, assim como na maioria dos sistemas de PLI, o predicado alvo é definido pelo parâmetro `modeh`. A declaração `head` indica ao conversor qual o predicado alvo do `modeh` para o conjunto de dados apresentado nos arquivos de entrada `.names` e `.data`.

A declararação é feita utilizando-se a palavra `head`, seguida por parâmetros entre colchetes ({}). Caso não seja fornecido nenhum parâmetro, como no exemplo apresentado na Tabela 5, o conversor utiliza como predicado alvo o próprio nome da classe declarado no arquivo `.names`. Utilizando o arquivo `voyage.names` — Tabela 2 na página 7 — no qual o atributo classe é chamado `class`, o conversor cria a seguinte declaração de `modeh` com valor 1 para o *recall*.

```
:-modeh(1,class(+att_id,#class)).
```

Neste caso o predicado padrão terá sempre aridade 2, sendo o primeiro argumento o `+att_id`, que corresponde ao atributo que identifica cada exemplo do conjunto de dados, com o tipo de instanciação ajustado para '+', seguido do atributo que corresponde ao valor da classe, com tipo de instanciação ajustado para '#'.

No caso do usuário definir os parâmetros de `head` a declaração deve ter o seguinte formato:

head{`Recall,Atributo_Cabeça(Inst Argumento`$_1$`,...,Inst Argumento`$_n$`).`}

O usuário deve declarar o valor de *Recall*, seguido pelo nome do predicado alvo. Deve ser observado que o valor de `Atributo_Cabeça`, *i.e.* o nome do predicado alvo, deve ser o nome de qualquer dos atributos declarados no arquivo `.names`. Dentro dos parênteses deve ser declarado os tipos dos argumentos separados por vírgula (',') e a sua instanciação (+, − ou #). Observar que podem ser declarados vários argumentos para o predicado alvo.

Os tipos de cada argumento também devem ter o nome de qualquer dos atributos declarados no arquivo `.names`, e não devem ser iniciados por letras maiúsculas para que o Aleph não os considere nome de variáveis. Por exemplo, se o usuário declarar:

```
head{1,windy(+att_id,#windy).}
```

O conversor cria o seguinte `modeh`:

```
modeh(1,windy(+att_id,#windy)).
```

O predicado alvo pode ter um ou mais argumentos

**Declaração body**

Este é o segundo parâmetro que deve ser declarado no arquivo `.bk`. Ele define os predicados que podem aparecer no corpo das regras induzidas por Aleph. Nos sistemas de PLI, esses predicados são definidos pelo parâmetro `modeb`.

A declaração é feita por meio da palavra `body`, seguida por abre e fecha colchetes ({}). A declaração admite zero, um ou vários parâmetros entre os colchetes. Caso não seja fornecido nenhum parâmetro para `body`, como no exemplo apresentado na Tabela 5 na página 17, o conversor cria, por *default*, tantos `modeb` quanto atributos (exceto o atributo `classe` e `att_id`) tenha o conjunto de exemplos considerado (arquivo `.names`), com valor '*' de *recall*. O nome de cada um desses atributos é usado para nomear um predicado de aridade 2. Para o exemplo considerado — Tabela 2 na página 7 — o conversor cria os seguintes quatro `modeb`:

```
:- modeb(*,outlook(+att_id,#outlook)).
:- modeb(*,temperature(+att_id,#temperature)).
:- modeb(*,humidity(+att_id,#humidity)).
:- modeb(*,windy(+att_id,#windy)).
```

O primeiro argumento é o atributo `att_id`, que corresponde ao atributo de identificação dos exemplos no conjunto de dados, com seu tipo de instanciação ajustado para '+'. O segundo argumento é o mesmo do nome do predicado e seu tipo de instanciação ajustado para '#'.

No caso do usuário definir os parâmetros de `body`, a declaração deve ter o seguinte formato:

body{$\texttt{Atributo}_i$_$\texttt{Atributo}_j$,...,$\texttt{Atributo}_k$_$\texttt{Atributo}_n$}

Cada $\texttt{Atributo}_i$ deve ser idêntico ao nome de um dos atributos declarados no arquivo `.names`. A concatenação desses nomes, utilizando o simbolo '_' para concatenar, corresponde ao nome do predicado. Cada um dos nomes dos atributos que participam do nome do predicado, na mesma ordem, definem os dos argumentos (tipos) desse predicado. O usuário pode definir qualquer número de predicados com

qualquer número de argumentos. Por exemplo, se o usuário declarar

```
body{outlook_temperature,outlook_humidity,outlook_humidity_temperature}
```

o conversor cria os seguintes três predicados

```
outlook_temperature(+att_id,#outlook,#temperature).
outlook_humidity(+att_id,#outlook,#humidity).
outlook_humidity_temperature(+att_id,#outlook,#humidity,#temperature).
```

Observar que o conversor sempre insere o atributo `att_id` como primeiro argumento de todos os predicados, com o objetivo de construir a representação relacional dos exemplos equivalentes a representação desses exemplos no formato atributo-valor.

**Declaração positives e negatives**

Os parâmetros `positive` e `negative` — Tabela 5 na página 17 — são, respectivamente, o terceiro e o quarto parâmetros declarados no arquivo `.bk`. O primeiro é para declarar a classe positiva dos exemplos e o segundo a classe negativa. Essa informação é usada pelo conversor para construir, respectivamente, o arquivo de exemplos positivos `.f` e negativos `.n`.

A declaração é feita utilizando-se, respectivamente, a palavra `positive` ou `negative` seguida por parâmetros entre colchetes, como mostrado a seguir:

positive{Valor_Classe_Positiva$_1$,...,Valor_Classe_Positiva$_j$}
negative{Valor_Classe_Negativa$_1$,...,Valor_Classe_Negativa$_k$}

As declarações permitem 0 ou mais parâmetros.

Para definir quais os valores que podem ser declarados para os exemplos positivos e negativos, basta conhecer qual é a cabeça definida para os predicados. Quando o usuário optar pela criação padrão do `modeh`, ou seja, se `head` não tem nenhum parâmetro, a cabeça será sempre o atributo classe, definido no arquivo `.names`.

Utilizando o mesmo exemplo do conjunto de dados `voyage`, o qual está representado na Tabela 2 na página 7, temos: `class: nominal (go, dont_go).`

E a criação padrão do `modeh` será:

```
:- modeh(1,class(+att_id,#class)).
```

Assim, pode-se definir como exemplos positivos e/ou negativos os valores go ou dont_go, como no exemplo apresentado na Tabela 5 na página 17.

Podem ocorrer quatro combinações diferentes para a definição dos parâmetros positive e negative. São possíveis as seguintes declarações:

- positive{} e negative{} - Neste caso em que ambas as declarações não possuem parâmetros, o usuário está definindo que o sistema não deve inserir nenhum exemplo nos arquivos de exemplos positivos (.f) e negativos (.n). Assim, o sistema apenas cria os arquivos .f e .n, inserindo no início dos arquivos, respectivamente, os comentários:

  %%%Positive Examples%%% e
  %%%Negative Examples%%%

- positive{pos} e negative{} - No caso em que há uma, ou várias, declarações em positive e zero em negative, o sistema irá criar o arquivo de exemplos positivos com todos os exemplos que no arquivo .data assumem o valor pos para o atributo definido como cabeça.

  O arquivo de exemplos negativos será criado com todos os exemplos que no arquivo .data assumirem valores diferentes daqueles definidos em positive{}, ou seja, todos os exemplos, exceto os positivos, serão negativos;

- positive{pos} e negative{neg} - No caso em que tanto positive como negative possuem um ou vários parâmetros, o usuário está especificando ao sistema quais os valores que os exemplos positivos e negativos terão, sendo que, se o atributo cabeça possuir outro valor , que não esteja declarado em positive ou em negative, os exemplos com esses valores não farão parte do conjunto de exemplos, ou seja, serão ignorados pelo sistema para a criação automática dos arquivos de exemplos positivos e negativos;

- positive{} e negative{POSONLY} - Quando a declaração de positive possuir zero parâmetros e negative for declarado com o parâmetro POSONLY, o sistema assume que todos os exemplos são positivos e apenas o arquivo de exemplos positivos é criado.

## 4.2 Formato do Arquivo de Saída .b

Entre os três arquivos de saída gerados pelo módulo conversor *Kaeru*, encontram-se os arquivos para exemplos positivos (.f) e negativos (.n) — Figura 3 na página 16 —,

cujo formato foi descrito na Seção 3.2.5 na página 14. O terceiro arquivo `.b` contém os parâmetros que direcionam a execução do Aleph. Os dados que estão no arquivo de conhecimento do domínio (`.b`), também são gerados automaticamente pelo módulo *Kaeru*, utilizando as informações que estão nos arquivos de entrada apresentados na Seção 4.1 na página 16.

A sintaxe dos dados segue a sintaxe do sistema de PLI Aleph (Srinivasan, 2000), descrito na Seção 3.2 na página 11, sendo que todas as informações necessárias para que ele possa construir os predicados encontra-se nesse arquivo. Essa informação pode ser dividida em algumas partes. São elas:

**Parâmetros do Aleph** São as declarações dos parâmetros do Aleph feitas no arquivo de entrada `.param`;

**Declaração dos Modos** A segunda parte que é criada pelo módulo *Kaeru* são as declarações dos modos que descrevem os predicados que serão utilizados pelo Aleph. Primeiro é feita a criação dos `modeh`, e para isso o sistema utiliza as informações declaradas no parâmetro `head` do arquivo `.bk`; em seguida o sistema cria todos os `modeb`, utilizando a informação do parâmetro `body` também do arquivo `.bk`;

**Determinations** A terceira parte é a criação das declarações dos determinations. Esse parâmetro descreve para o Aleph todos os predicados que ele poderá utilizar no processo de indução. Os determinations são criados utilizando a informação dos parâmetros `head` e `body` do arquivo `.bk`. Como descrito na Seção 3.2 na página 11, o determination tem a seguinte sintaxe:

```
:- determination(Head/Arityh,Body/Arityb).
```

Para criar a primeira parte, que descreve no determination qual será a cabeça dos predicados e sua aridade, o sistema utiliza a informação do parâmetro `head`, enquanto que a segunda parte, que descreve qual o corpo que o predicado poderá ter e sua aridade, é retirada do parâmetro `body`;

**Declaração dos Tipos** A declaração dos tipos é feita pelo módulo *Kaeru* utilizando-se as informações disponíveis no arquivo `.names`. O sistema converte a declaração dos tipos que estão na sintaxe DSX para o formato aceito pela sintaxe do sistema Aleph. Serão declarados automaticamente pelo sistema todos os atributos que foram utilizados nos predicados e seus respectivos tipos. Por exemplo, dada a seguinte declaração no arquivo `.names`:

```
outlook: nominal (sunny, overcast, rain).
temperature: integer.
```

O sistema declara no arquivo `.b`:

```
outlook(sunny).
outlook(overcast).
outlook(rain).
temperature(integer).
```

É importante lembrar que a declaração dos tipos, tal como do tipo `integer` no exemplo, não faz com que o Aleph realize nenhuma checagem de tipos. Nos sistemas relacionais, as declarações de tipos são necessárias apenas para limitar a busca realizada pelo sistema;

**Conhecimento do Domínio** A última parte criada pelo módulo $\mathcal{K}aeru$ é a inclusão do conhecimento do domínio, que é necessário para que o sistema Aleph possa induzir conhecimento utilizando bases de dados originalmente proposicionais. Para isso o módulo $\mathcal{K}aeru$ cria, para cada atributo fornecido no arquivo `.names` (exceto para o atributo classe e `att_id`) uma cláusula Prolog. Por exemplo, considerando o arquivo `voyage.names` — Tabela 2 na página 7 — o módulo $\mathcal{K}aeru$ sempre cria as seguintes cláusulas:

```
outlook(Att_id,Outlook):-
        example(Att_id,Outlook,_,_,_,_), Outlook \= '?'.
temperature(Att_id,Temperature):-
        example(Att_id,_,Temperature,_,_,_), Temperature \= '?'.
humidity(Att_id,Humidity):-
        example(Att_id,_,_,Humidity,_,_), Humidity \= '?'.
windy(Att_id,Windy):-
        example(Att_id,_,_,_,Windy,_), Windy \= '?'.
```

O símbolo ? é utilizado na sintaxe DSX para representar um atributo com valor desconhecido. Assim, atributos com valor desconhecido não são considerados. Os predicados `Example/Aridade` tem sua aridade definida pelo número de atributos (incluindo a classe) fornecidos no arquivo `.names`. O módulo $\mathcal{K}aeru$ cria também, para cada exemplo fornecido no arquivo `.data` um fato `Example`. Considerando o arquivo `.data` na Tabela 3 na página 7, o módulo $\mathcal{K}aeru$ sempre cria os seguintes seis fatos:

```
example(0,sunny, 25, 72, yes, go).
example(1,sunny, 28, 91, yes, dont_go).
example(2,overcast, 23, 90, yes, go).
example(3,overcast, 29, 78, no, go).
example(4,rain, 22, 95, no, go).
example(5,rain, 19, 70, yes, dont_go).
```

Considerando as declarações de body apresentadas na página 20, em que o usuário declarou diferentes predicados, então o módulo conversor *Kaeru* cria, adicionalmente, as seguintes cláusulas:

```
outlook_temperature(Att_id,Outlook,Temperature):-
       example(Att_id,Outlook,Temperature,_,_,_),
       Outlook \= '?', Temperature \= '?'.
outlook_humidity(Att_id,Outlook,Humidity):-
       example(Att_id,Outlook,_,Humidity,_,_),
       Outlook \= '?', Humidity \= '?'.
outlook_humidity_temperature(Att_id,Outlook,Humidity,Temperature):-
       example(Att_id,Outlook,Temperature,Humidity,_,_),
       Outlook \= '?', Temperature \= '?', Humidity \= '?'.
```

# 5 Conclusão

Neste trabalho foi dada uma visão geral do módulo *PLI* que está sendo desenvolvido para ser integrado ao ambiente computacional DISCOVER. Esse módulo permitirá incluir no DISCOVER algoritmos de aprendizado relacional. O módulo PLI consiste do módulo conversor $\mathcal{K}aeru$ e do SNIFFER PLI. O módulo conversor $\mathcal{K}aeru$, descrito em detalhes neste trabalho, tem como objetivo auxiliar no processo de transformação de bases de dados no formato atributo-valor, descritas na sintaxe DSX do DISCOVER, para o formato relacional utilizado por sistemas de PLI. Além disso, o módulo conversor $\mathcal{K}aeru$ cria as declarações necessárias para executar o sistema de PLI Aleph.

Uma das vantagens do Aleph é que ele permite simular diversos sistemas de PLI existentes, o que facilita as pesquisas nessa área. Assim o módulo conversor $\mathcal{K}aeru$ facilitará os trabalhos de pesquisa relacionados a extrair conhecimento de dados no formato atributo-valor utilizando tanto algoritmos de aprendizado proposicional quanto relacional.

O segundo módulo do módulo PLI do DISCOVER, SNIFFER PLI, em desenvolvimento, tem como objetivo medir o erro de hipóteses induzidas utilizando o Aleph, usando *k-fold cross validation*, ou usando conjuntos de treinamento e teste definidos pelo usuário.

# A Parâmetros do Aleph

`set(best,+V)`

V é uma lista relacionada a uma cláusula obtida de uma execução anterior contendo, no mínimo, o número de exemplos positivos cobertos, o número de negativos cobertos e o tamanho da cláusula encontrada em uma busca anterior. Útil quando realizando buscas iterativamente;

`set(cache_clauselength,+V)`

V é um inteiro positivo (padrão é 3). Ajusta o limite superior do tamanho de cláusulas que podem ser consideradas para uso futuro;

`set(caching,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então cláusulas e suas coberturas são armazenadas para uso futuro. Somente as cláusulas com tamanho menor ou igual ao que foi ajustado em `cache_clauselength` são armazenadas;

`set(check_redundant,+V)`

V é `true` ou `false` (padrão é *false*). Especifica se uma chamada a `redundant/2` deve ser feita para verificar por literais redundantes em uma cláusula;

`set(check_useless,+V)`

V é `true` ou `false` (padrão é `false`). Se `true`, remove literais na cláusula *bottom* que não contribuem para estabelecer ligações de variáveis;

`set(clauses,+V)`

V é uma lista de classes a serem preditas quando utilizando aprendizado com árvores (*Tree Learning*);

`set(clauselength,+V)`

V é um inteiro positivo (padrão é 4). Ajusta o limite superior do número de literais em uma cláusula aceitável;

`set(clauselength_distribution,+V)`

V é uma lista da forma `[p1-1,p2-2,...]` em que `pi` representa a probabilidade de selecionar uma cláusula com $i$ literais. Utilizada pelos métodos aleatórios de busca;

`set(clauses,+V)`

V é um inteiro positivo. Ajusta o limite superior do número de cláusulas em uma teoria quando realizando *theory_level search*;

`set(condition,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então os exemplos gerados randomicamente são obtidos após o condicionamento do gerador estocástico com os exemplos positivos;

`set(confidence,+V)`

V é um número real no intervalo de 0.0 até 1.0 (padrão é 0.95). Determina a confiança para poda de regras no aprendizado com árvores;

`set(construct_bottom,+V)`

V é `saturation`, `reduction` ou `false` (padrão é `saturation`). Especifica o estágio em que a cláusula *bottom* deve ser construída;

`set(depth,+V)`

V é um inteiro positivo (padrão é 10). Ajusta o limite superior da profundidade de prova (*proof depth*);

`set(explore,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então força a busca a continuar até que todos os elementos restantes no espaço de busca são definitivamente piores que o melhor elemento atual (normalmente a busca pararia quando todos os elementos restantes não fossem melhores que o melhor elemento). Este é um critério fraco. Toda poda interna é ajustada para `off`;

`set(evalfn,+V)`

V é `coverage`, `compression`, `posonly`, `pbayes`, `accuracy`, `laplace`, `auto_m`, `mestimate`, `entropy`, `gini`, `sd`, `wracc` ou `user` (padrão é `coverage`). Especifica qual a função de avaliação que será usada na busca;

`set(good,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então armazena código Prolog das "boas"cláusulas encontradas na busca. Uma cláusula boa é qualquer cláusula com utilidade acima da especificada em `minscore`. Se em `goodfile` foi definido algum nome de arquivo então esse código é armazenado nesse arquivo;

`set(goodfile,+V)`

V é um átomo Prolog. Define o nome do arquivo para armazenar as boas cláusulas encontradas nas buscas;

`set(gsamplesize,+V)`

V é um inteiro positivo (padrão é 100). Define o tamanho do conjunto de exemplos randomicamente gerado no aprendizado utilizando somente exemplos positivos;

`set(i,+V)`

V é um inteiro positivo (padrão é 2). Ajusta o limite superior para a profundidade das novas variáveis;

`set(language,+V)`

V é um inteiro $\geq 1$ ou `inf` (padrão é `inf`). Especifica o número de ocorrências de um predicado em qualquer cláusula;

`set(lazy_on_contradiction,+V)`

V é `true` ou `false` (padrão é *false*). Especifica se a prova de teorema pode

prosseguir se uma restrição é violada;

`set(lazy_on_cost,+V)`

V é `true` ou `false` (padrão é *false*). Especifica se a função de custo definida pelo usuário requer que a cobertura de cláusulas seja avaliada. Isto normalmente é definido internamente e não pelo usuário;

`set(lazy_negs,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então a prova de teorema com exemplos negativos pára quando o limite ajustado em `noise` ou `minacc` é violado;

`set(lookahead,+V)`

V é um inteiro positivo. Ajusta um valor para o operador automático de refinamento;

`set(m,+V)`

V é um número real. Ajusta um valor do *m-estimate*;

`set(minacc,+V)`

V é um número real entre 0.0 e 1.0 (padrão é 0.0). Ajusta o limite inferior da precisão mínima de uma cláusula aceitável;

`set(mingain,+V)`

V é um número real (padrão é 0.5). Especifica o ganho mínimo esperado para dividir uma folha quando construindo árvores;

`set(minpos,+V)`

V é um inteiro positivo (padrão é 1). Ajusta o limite inferior do número de exemplos positivos que devem ser cobertos por uma cláusula. Se a melhor cláusula cobre um número menor de exemplos positivos, ela não é adicionada a teoria atual;

`set(minposfrac,+V)`

V é um número real entre 0.0 e 1.0 (padrão é 0.0). Ajusta o limite inferior de exemplos positivos cobertos por uma cláusula aceitável como sendo a fração dos exemplos positivos cobertos pela cabeça dessa cláusula. Se a melhor cláusula tem taxa inferior a este número então ela não é adicionada a teoria atual;

`set(minscore,+V)`

V é um número real ou `-inf` (padrão é `-inf`). Ajusta o limite inferior da utilidade de uma cláusula aceitável. Na construção de cláusulas, se a melhor cláusula tem utilidade abaixo deste número, então ela não é adicionada a teoria atual;

`set(moves,+V)`

V é um inteiro $\geq 0$. Ajusta o limite superior do número de movimentos permitidos quando realizando uma busca randomica local.

`set(newvars,+V)`

V é um inteiro positivo ou `inf`(padrão é `inf`). Ajusta o limite superior do número

de variáveis que podem ser introduzidas no corpo de uma cláusula;

`set(nodes,+V)`

V é um inteiro positivo (padrão é 5000). Ajusta o limite superior do número de nós a serem explorados quando realizando a busca por uma cláusula aceitável;

`set(noise,+V)`

V é um inteiro $\geq 0$ (padrão é 0). Ajusta o limite superior do número de exemplos negativos que podem ser cobertos por uma cláusula aceitável;

`set(openlist,+V)`

V é um inteiro $\geq 0$ ou `inf`(padrão é `inf`). Ajusta o limite superior da busca em largura para ser usada em uma busca gulosa (*greedy search*);

`set(optimise_clauses,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, realiza *query optimisations* descrita por V.S. Costa, A. Srinivasan, e R.C. Camacho em *A note on two simple transformations for improving the efficiency of an ILP system.*

`set(portray_hypothesis,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, executa a meta `portray(hypothesis)`, que deve ser definida pelo usuário;

`set(portray_literals,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, executa a meta `portray(literals)` no qual `Literals` é algum literal. Essa meta deve ser definida pelo usuário;

`set(portray_search,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, executa a meta `portray(search)` que deve ser definida pelo usuário;

`set(print,+V)`

V é um inteiro positivo (padrão é 4). Define o limite superior do número máximo de literais mostrados no *trace*;

`set(proof_strategy,+V)`

V é `restrict_sld` ou `sld` (padrão é `restrict_sld`). Se `restrict_sld`, então os exemplos cobertos foram determinados forçando a cláusula induzida a ser a primeira cláusula na prova de resolução SLD. Se `sld`, então essa restrição não é forçada;

`set(prooftime,+V)`

V é um inteiro positivo ou `inf` (padrão é `inf`). Ajusta o limite superior de tempo (em segundos) para testar se um exemplo é coberto;

`set(prune_tree,+V)`

V é `true` ou `false` (padrão é *false*). Determina se as regras construídas utilizando o aprendizado de árvore devem ser podadas utilizando poda pessimista;

`set(record,+V)`

V é `true` ou `false` (padrão é *false*). Se `true`, então as linhas de execução do Aleph (*trace*) são escritas em um arquivo; O nome do arquivo é dado por `recordfile`.

`set(recordfile,+V)`

V é um átomo Prolog. Define o nome do arquivo no qual serão gravadas as linhas de execução (*trace*) do Aleph;

`set(refine,+V)`

V é `user`, `auto` ou `false` (padrão é `false`). Especifica qual o tipo de operador de refinamento. Se `user`, então o usuário deve especificar o operador de refinamento com `refine\2`;

`set(rls_type,+V)`

V é `gsat`, `wsat`, `rrr` ou `anneal`. Especifica qual o método randomico de busca a ser utilizado;

`set(rulefile,+V)`

V é um átomo Prolog. Define o nome do arquivo no qual serão armazenadas as cláusulas encontradas em uma teoria (usado por `write_rules/0`);

`set(samplesize,+V)`

V é um inteiro $\geq 0$ (padrão é 0). Define o número de exemplos selecionados randomicamente pelos comandos `induce` ou `induce_cover`;

`set(scs_percentile,+V)`

V é um número na faixa de (0,100] (usualmente um número inteiro). Define que qualquer cláusula no topo do V-percentile de cláusulas são consideradas "boas" quando realizando seleção estocástica de cláusulas;

`set(scs_prob,+V)`

V é um número na faixa de (0, 1.0]. Define a probabilidade mínima de se obter uma "boa" cláusula quando realizando seleção estocástica de cláusula;

`set(scs_sample,+V)`

V é um número inteiro positivo que determina o número de cláusulas selecionadas randomicamente no espaço de hipóteses na busca clausal;

`set(search,+V)`

V é `bf`, `df`, `heuristic`, `ibs`, `ils`, `rls`, `scs`, `id`, `ic` ou `ar` (padrão é `bf`). Especifica qual a estratégia de busca;

`set(searchtime,+V)`

V é um inteiro $\geq 0$ ou `inf`. Especifica o limite superior de tempo (em segundos) para uma busca;

`set(stage,+V)`

V é `saturation`, `reduction` ou `command` (padrão é `command`). Define o estágio da execução. Isso normalmente é definido internamente e não pelo usuário;

`set(store_bottom,+V)`

V é `true` ou `false` (padrão é *false*). Armazena a cláusula *bottom* construída para um exemplo para futura reutilização;

`set(temperature,+V)`

V é um número real diferente de 0. Ajusta a `temperature` para busca quando utilizando *annealing*. Necessário que a `search` esteja definida como `rls` e `rls_type` para setar `anneal`;

`set(test_pos,+V)`

V é um átomo Prolog ou uma lista de átomos. Define o arquivo ou a lista de arquivos que contém os exemplos positivos para testar;

`set(test_neg,+V)`

V é um átomo Prolog ou uma lista de átomos. Define o arquivo ou a lista de arquivos que contém os exemplos negativos para testar;

`set(train_pos,+V)`

V é um átomo Prolog ou uma lista de átomos. Define o arquivo ou a lista de arquivos que contém os exemplos positivos para treinamento;

`set(train_neg,+V)`

V é um átomo Prolog ou uma lista de átomos. Define o arquivo ou a lista de arquivos que contém os exemplos negativos para treinamento;

`set(tree_type,+V)`

V é `classification`, `class_probability` ou `regression`;

`set(tries,+V)`

V é um inteiro positivo. Define o número máximo de reinicios permitido para os métodos randomicos de busca;

`set(typeoverlap,+V)`

V é um número real no intervalo $(0, 1.0]$. Usado por `induce_modes/0` para determinar se a um par de diferentes tipos pode ser dado o mesmo nome;

`set(verbosity,+V)`

V é um inteiro $\geq 0$ (padrão é 1). Define o nível de detalhes de execução do Aleph;

`set(version,+V)`

V é a versão atual do Aleph; é definido internamente;

`set(+P,+V)`

Ajusta qualquer parâmetro `P` definido pelo usuário para o valor `V`. Este parâmetro é útil para anexar anotações aos experimentos. Por exemplo `set(experiment,'Execução 1 com conhecimento B0')`.

O Aleph possui um número muito grande de parâmetros que podem ser ajustados. A

relação de todos esses parâmetros e os comandos disponíveis podem ser encontrados no manual do Aleph (Srinivasan, 2000), que pode ser obtido em:

www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/index.html

# Referências

Baranauskas, J. A. and G. E. A. P. A. Batista (2000). O Projeto DISCOVER: Idéias Iniciais. Comunicação pessoal. 2, 3

Batista, G. E. A. P. A. and M. C. Monard (2003). Descrição da Arquitetura e do Projeto do Ambiente Computacional DISCOVER LEARNING ENVIRONMENT — DLE. Technical Report 187, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_187.pdf. 3, 4, 5, 7

Batista, G. E. A. P. A. (2001). Sintaxe padrão do arquivo de exemplos do projeto DISCOVER. http://www.icmc.sc.usp.br/~gbatista/Discover/SintaxePadraoFinal.htm. 5

Blockeel, H. (1997, July). Tilde - top down induction of logical decision trees. Technical report, K.U.Leuven Dept. of Computer Science. 11

Boswell, T. (1990, January). Manual for NewId version 4.1. Technical Report TI/P2154/RAB/4/2.3, The Turing Institute. 5

Camacho, R. (1994). Learning stage transition rules with Indlog. Volume 237 of *GMD-Studien*, pp. 273–290. Gesellschaft für Mathematik und Datenverarbeitung MBH. 11

Chiara, R. and M. C. Monard (2003). Projeto e implementação de um filtro para transformar logs de servidores web em arquivos no formato padrão do sistema DISCOVER. Technical Report 183, ICMC-USP. i, 4

Clark, P. and R. Boswell (1991). Rule Induction with CN2: Some Recent Improvements. In Y. Kodratoff (Ed.), *Fifth European Conference (EWSL 91)*, pp. 151–163. Springer-Verlag. 5

Dehaspe, L. and L. De Raedt (1997). Mining association rules in multiple relations. pp. 125–132. Springer-Verlag. 11

Dosualdo, D. G. (2003). Investigação de regressão no processo de mineração de dados. Dissertação de Mestrado, ICMC-USP. 3

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Resusable Object-Oriented Software.* Addison Wesley. 4

Karalič, A. and I. Bratko (1997). First-order regression. *Machine learning.* 11

Lavrač, N. and S. Džeroski (1994). *Inductive Logic Programming: Techniques and Applications.* New York: Ellis Horwood. 1

Lavrač, N. and P. A. Flach (2001). An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic (TOCL) 2*(4), 458–494. 2

Melanda, E. A. (2002). Pós-processamento de conhecimento de regras de associação. Qualificação de Doutorado, ICMC-USP. 3

Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing 13*, 245–286. 11

Muggleton, S. (1999). Inductive logic programming: Issues, results and the challenge of learning language in logic. *Artificial Intelligence 114*(1-2), 283–296. 1

Prati, R. C., J. A. Baranauskas, and M. C. Monard (2001). Uma proposta de unificação da linguagem de representação de conceitos de algoritmos de aprendizado de máquina simbólicos. Technical Report 137, ICMC-USP. ftp://ftp.icmc.sc.usp.br/pub/BIBLIOTECA/rel_tec/RT_137.ps.zip. 5

Prati, R. C. (2003). O *Framework* de Integração do Sistema DISCOVER. Dissertação de Mestrado, ICMC-USP. 3

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning 1*, 81–106. Reprinted in Shavlik and Dietterich (eds.), 1990. Readings in Machine Learning, Morgan Kaufmann Publishers, Inc. 5

Quinlan, J. R. (1987). Generating Production Rules from Decision Trees. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Italy, pp. 304–307. 5

Quinlan, J. R. (1988). *C4.5 Programs for Machine Learning.* CA: Morgan Kaufmann. 5, 6, 17

Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning 5*(3), 239–266. 11

Srinivasan, A. (2000). The aleph manual. Technical report, Oxford University. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph. 11, 22, 32

Wall, L., T. Christiansen, and R. L. Schwartz (1996). *Programming Perl* (2 ed.). O'Reilly & Associates. 2

Wrobel, S. (1997). An algorithm for multi-relational discovery of subgroups. In J. Komorowski and J. Zytkow (Eds.), *Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, Berlin, pp. 78–87. Springer-Verlag. 11