Universidade de São Paulo
Instituto de Física de São Carlos
Departamento de Física e Informática

Ampliando os Limites do Aprendizado
Indutivo de Máquina
através das Abordagens Construtiva e Relacional

Maria do Carmo Nicoletti

Orientação:

Profa. Dr. Maria Carolina Monard

Tese apresentada ao Instituto de Física de São Carlos - USP, como parte dos requisitos exigidos para a obtenção do título de Doutor em Ciências ("Física Aplicada, subárea Física Computacional").

São Carlos
1994

**IFSC UNIVERSIDADE
DE SÃO PAULO**

Instituto de Física e Química de São Carlos

Fone (0162) 72-6222
Fax (0162) 72-2218

Av. Dr. Carlos Botelho, 1465
Caixa Postal 369
CEP 13560.970 - São Carlos - SP
Brasil

MEMBROS DA COMISSÃO JULGADORA DA TESE DE DOUTORADO DE MARIA DO CARMO NICOLETTI APRESENTADA AO
INSTITUTO DE FÍSICA E QUÍMICA DE SÃO CARLOS, DA UNIVERSIDADE DE SÃO PAULO, EM 24/06/1994

COMISSÃO JULGADORA:

Prof. Dra. Maria Carolina Morand

Prof. Dr. Marcio Rillo

Prof. Dra. Maria Cecilia Calani Baranauskas

Prof. Dr. José Fernando Fontanari

Prof. Dr. Alvaro Garcia Neto

para Adriana V. Villaça, Vicky L. Giles e Rachel M. Giles

IFSC — SERVIÇO DE BIBLIOTECA E
DOCUMENTAÇÃO

Agradecimentos

Dr. M.C. Monard, pela dedicação e incentivo durante a orientação desta tese

Dr. N. Lavrač, pelo apoio e constantes contribuições

Dr. D.F.de Aragon, pelas discussões, críticas e sugestões

Dr. J.R. Quinlan e seu grupo de pesquisa, pela acolhida e atenção a mim dispensadas

Dr. M.G.N. Mizukami, pela leitura e comentários sobre esta tese

Dr. C.M.F. de Andrade, Dr. O.L. Linhares, M.B.O Bueno, M.A. Nicoletti, S.C.P.F. Fabbri, R.A.D. Penteado, M.G.B. Rocha, J.A. Coutinho, S.D. Zorzo, S.M. Aluisio, C.L. Pagadigorria, pelo apoio

ao programa PICD/CAPES, através da UFSCar

ao programa ProTeM/CC-SP/CNPq

ao IFQSC/USP, ICMSC/USP e UFSCar

Resumo

Este trabalho investiga Aprendizado Indutivo de Máquina como função das linguagens de descrição, utilizadas para expressar instâncias, conceitos e teoria do domínio. A ampliação do poder de representação do aprendizado proposicional é abordada no contexto de indução construtiva, no domínio de funções booleanas, com a proposta de uma estratégia de composição de atributos denominada *root-fringe*. Avaliações experimentais dessa e de outras estratégias de construção de novos atributos foram conduzidas e os resultados analisados. Dois métodos de poda, para tratamento de ruídos, em aprendizado de árvores de decisão, foram avaliados num ambiente de indução construtiva e os resultados discutidos. Devido à limitação do aprendizado proposicional, foram investigadas formas de ampliação dos limites do aprendizado, através da ampliação do poder representacional das linguagens de descrição. Foi escolhida Programação Lógica Indutiva — PLI — que é um paradigma de aprendizado indutivo que usa restrições de Lógica de Primeira Ordem como linguagens de descrição. O aprendizado em PLI só é factível quando as linguagens utilizadas estão restritas e são fortemente controladas, caso contrário, o aprendizado em PLI se torna indecidível. A pesquisa em PLI se direccionou à formas de restrição das linguagens de descrição da teoria do domínio e de hipóteses. Três algoritmos que “traduzem” a teoria do domínio de sua forma intencional, para extensional, são apresentados. As implementações de dois deles são discutidas. As implementações realizadas deram origem a dois ambientes experimentais de aprendizado: o ambiente proposicional experimental, do qual faz parte o ambiente experimental construtivo, e o ambiente experimental relacional.

Abstract

This work investigates Inductive Machine Learning as a function of the description languages employed to express instances, concepts and domain theory. The enlargement of the representational power of propositional learning methods is approached via constructive induction, in the domain of boolean functions, through the proposal of a bias for composing attributes, namely, the bias *root-fringe*. Experimental evaluation of *root-fringe*, as well as other biases for constructing new attributes were conducted and the results analysed. Two pruning methods for decision trees were evaluated in an environment of constructive induction and the results discussed. Due to the limitations of propositional learning, ways of enlarging the limits of the learning process were investigated through enlarging the representational power of the description languages. It was chosen Inductive Logic Programming — ILP — which is an inductive learning paradigm which uses restrictions of First Order Logic as description languages. Learning using ILP is only feasible when the languages are restricted and are strongly controlled, otherwise, learning in ILP becomes undecidable. Research work in ILP was directed towards restricting domain theory and hypotheses description languages. Three algorithms that “translate” the intensional expression of a domain theory into its extensional expression are presented. The implementation of two of them are discussed. The implementations gave rise to two experimental learning environments: the propositional environment, which includes the constructive environment, and the relational environment.

Conteúdo

1	Introdução	1
1.1	Contextualização	1
1.2	Objetivos e Pesquisa Desenvolvida	2
1.3	Organização da Tese	3
2	Aquisição de Conhecimento	6
2.1	Considerações Iniciais	6
2.2	Sistemas Baseados em Conhecimento	6
2.3	O Processo de Aquisição de Conhecimento	8
2.4	Modalidades de Aquisição de Conhecimento	10
2.5	Considerações Finais	12
3	Aprendizado de Máquina	14
3.1	Considerações Iniciais	14
3.2	Aprendizado e Aprendizado de Máquina	14
3.3	Um Modelo de Aprendizado	16
3.3.1	O Ambiente	17
3.3.2	A Base de Conhecimento	18
3.3.3	O Módulo de Desempenho	19
3.4	Considerações sobre Aprendizado de Máquina	20
3.5	Considerações Finais	21
4	Paradigma Indutivo de Aprendizado de Máquina	22
4.1	Considerações Iniciais	22

4.2	Taxionomia de Aprendizado de Máquina	22
4.3	O Aprendizado Indutivo	23
4.4	Características do Aprendizado Indutivo	24
4.5	Cr�terios de Sucesso	25
4.6	Conceitos B�sicos	26
4.7	Relev�ncia das Linguagens de Descri��o no Aprendizado Indutivo	29
4.8	Considera��es Finais	32
5	Aprendizado de �rvores de Decis�o	33
5.1	Considera��es Iniciais	33
5.2	�rvores de Decis�o como Linguagem de Descri��o de Hip�teses	34
5.3	Algoritmo Geral dos Sistemas da Fam�lia TDIDT	35
5.4	Cr�terios para Avalia��o de �rvores de Decis�o	38
5.4.1	Tipos de Atributos Utilizados em Linguagens Proposicionais . .	39
5.4.2	Cr�terios para a Avalia��o dos Atributos	39
5.5	Tratamento do Conjunto de Treinamento	40
5.6	O Mecanismo de Janela	43
5.7	O Mecanismo de Poda	45
5.8	M�todo de Poda da Redu��o do Erro e seu Variante	46
5.9	Fam�lia TDIDT	48
5.10	Um Ambiente Experimental para a Fam�lia TDIDT	51
5.11	O Ambiente de Aprendizado Proposicional	54
5.12	Considera��es Finais	56
6	Aprendizado Construtivo	57
6.1	Considera��es Iniciais	57
6.2	Amplia��o da Linguagem de Descri��o de Exemplos Atrav�s da Con- stru��o de Features	58
6.2.1	O Uso de Operadores Construtivos	60
6.2.2	Principais Caracter�sticas de um Sistema Construtivo	61
6.3	Aprendizado de Fun��es Booleanas com Constru��o Autom�tica de Fea- tures	63

6.4	Definição do Operador Construtivo	64
6.5	Estratégias para Construção de Features	65
6.6	Descrição dos Algoritmos	67
6.6.1	Algoritmo Iterativo de Construção de Árvore de Decisão com Construção Automática de Features	67
6.6.2	Algoritmo Root	69
6.6.3	Algoritmo Fringe	71
6.6.4	Algoritmo Root-Fringe	73
6.7	Pré-Processamento do Conjunto de Treinamento	75
6.7.1	Minimização de Funções Booleanas	75
6.7.2	Algoritmo para o Pré-Processamento do Conjunto Treinamento .	76
6.8	Ambiente Experimental Construtivo	76
6.9	Considerações Finais	78
7	Avaliação Empírica dos Algoritmos para Construção de Features	79
7.1	Considerações Iniciais	79
7.2	Avaliação das Estratégias de Construção de Features	79
7.2.1	Análise dos Resultados	81
7.2.2	Avaliação dos Biases Root, Fringe e Root-Fringe	89
7.3	Descrição dos Experimentos com Métodos de Poda e Análise dos Resultados	90
7.4	Avaliação da Estratégia de Pré-Processamento do Conjunto de Treinamento	97
7.5	Considerações Finais	99
8	Programação Lógica Indutiva — PLI	100
8.1	Considerações Iniciais	100
8.2	Conceitos Básicos	101
8.3	O Processo de Aprendizado em PLI	102
8.4	Buscas no Espaço de Hipóteses	103
8.4.1	Através de Técnicas de Generalização	105
8.4.2	Através de Técnicas de Especialização	108

8.5	Considerações Finais	108
9	A Teoria do Domínio em PLI	110
9.1	Considerações Iniciais	110
9.2	Restringindo a Teoria do Domínio	110
9.2.1	Conceitos Básicos	111
9.2.2	O Modelo H-easy	112
9.2.3	Cláusulas Generativas	116
9.3	Transformação de Teoria de Domínio Intencional em Extensional	119
9.3.1	Algoritmo para Verificar se um Programa Lógico é Generativo .	120
9.3.2	Algoritmos para a Geração do Fechamento da Derivação Atômica de um Programa Lógico Generativo	124
9.4	O Ambiente de Aprendizado Relacional — PLI	127
9.5	Considerações Finais	129
10	Linguagem de Descrição de Hipóteses em PLI	130
10.1	Considerações Iniciais	130
10.2	Linguagem de Descrição de Hipóteses sem Símbolos Funcionais	131
10.2.1	Profundidade de Variáveis e Cláusulas	131
10.2.2	ij-Determinação	133
10.3	Linguagem de Descrição de Hipóteses com Símbolos Funcionais	134
10.3.1	Termo Determinado	135
10.3.2	ij-Determinação	141
10.3.3	Outra Noção de Profundidade e Nível de Termo	142
10.4	Considerações Finais	144
11	Os Sistemas de PLI <i>GOLEM</i> e <i>FOIL</i>	146
11.1	Considerações Iniciais	146
11.2	O Sistema <i>GOLEM</i>	146
11.2.1	O Algoritmo <i>GOLEM</i>	148
11.2.2	O Aprendizado do Conceito de <i>cena</i> a Partir de Dois Exemplos Positivos	148

11.2.3 O Aprendizado do Conceito de <i>cena</i>	152
11.2.4 O Aprendizado de Recursão	154
11.3 O Sistema FOIL	154
11.3.1 O Algoritmo <i>FOIL</i>	160
11.3.2 O Aprendizado do Conceito <i>cena</i>	161
11.3.3 O Aprendizado de Recursão	161
11.4 Considerações sobre <i>GOLEM</i> e <i>FOIL</i>	163
11.5 Considerações Finais	164
12 Conclusões	166
A Busca como Método de Resolução de Problemas	169
B Entropia	174
C ID3 e o Assistant Professional	177
Bibliografia	185

Lista de Figuras

2.1	Estrutura Básica de um SBC	7
2.2	SBC e SE	8
2.3	SBC e seu Ambiente	9
2.4	Modalidades de Aquisição de Conhecimento [Wielinga-87]	11
3.1	Um Modelo de Sistema de Aprendizado	17
4.1	Seqüência de Exemplos e Contra-exemplos para o Aprendizado do Con- ceito de Arco	25
4.2	Completeza e Consistência de uma Hipótese	28
5.1	Árvore de Classificação de Hipotireoidismo	35
5.2	Estrutura de Classificação Pictoricamente Representada como Árvore	36
5.3	Uma Única Divisão do Intervalo - Entropia = 0.490	42
5.4	Duas Divisões do Intervalo - Entropia = 0.261	43
5.5	Mecanismo de Janela	44
5.6	Subárvore de uma Árvore de Classificação	47
5.7	Família TDIDT	49
5.8	Ambiente Experimental TDIDT	52
5.9	Visão Geral do Ambiente Experimental TDIDT	53
5.10	Ambiente de Aprendizado Proposicional	55
6.1	Árvore de Decisão que Classifica um Robô em <i>amigo</i> ou <i>inimigo</i>	59
6.2	Árvore de Decisão que Classifica um Robô em <i>amigo</i> ou <i>inimigo</i> Usando a Feature <i>mesma-forma</i>	59
6.3	Construção de Feature: um Problema de Busca	61

6.4	Representação da Relação entre o Sistema e o Processo de Construção de Features	63
6.5	Árvore de Decisão da Função $f(x_1, x_2, x_3, x_4) = \sum \min(1, 3, 5, 7)$	65
6.6	Árvore de Decisão da Função $f(x_1, x_2, x_3, x_4) = \sum \min(1, 3, 5, 7)$ com a Feature $f_1 = \text{and}(x_1 = 0, x_4 = 1)$	65
6.7	Biases para a Construção de Features	66
6.8	Construção de Árvore de Decisão com Construção Automática de Features	67
6.9	<i>Root</i> - Primeira Iteração	69
6.10	<i>Root</i> - Segunda Iteração	70
6.11	<i>Root</i> - Terceira e Última Iteração	70
6.12	<i>Fringe</i> - Primeira Iteração	71
6.13	<i>Fringe</i> - Segunda Iteração	72
6.14	<i>Fringe</i> - Terceira Iteração	72
6.15	<i>Fringe</i> - Quarta e Última Iteração	72
6.16	<i>Root-Fringe</i> - Primeira Iteração	74
6.17	<i>Root-Fringe</i> - Segunda Iteração	74
6.18	<i>Root-Fringe</i> - Terceira e Última Iteração	74
6.19	Pré-Processamento do Conjunto de Treinamento	76
6.20	Visão Geral do Ambiente Experimental Construtivo	77
6.21	Módulo Gerador de Exemplos	77
7.1	3-termo 3FND - o: H •: NF ×: NN	84
7.2	4-termo 3FND - o: H •: NF ×: NN	85
7.3	(P%)Precisão - o: 3-termo 3 FND ×: 4-termo 3 FND	86
7.4	(NI)Número de Iterações - o: 3-termo 3FND ×: 4-termo 3FND	86
7.5	3-termo 3FND - o: (FA)Features Distintas na Árvore •: (FC)Features Distintas Construídas	87
7.6	4-termo 3FND - o: (FA)Features Distintas na Árvore •: (FC)Features Distintas Construídas	87
7.7	3-termo 3FND - ×: (Pr)Primitivas Distintas na Árvore •: (FPr)Primitivas na Árvore	88
7.8	4-termo 3FND - ×: (Pr)Primitivas Distintas na Árvore •: (FPr)Primitivas na Árvore	89

7.9	(H)Altura 3-termo 3FND - o: sem poda o: reduced x: variante	91
7.10	(H)Altura 4-termo 3FND - o: sem poda o: reduced x: variante	92
7.11	(NF)Número Folhas 3-termo 3FND - o: sem poda o: reduced x: variante	92
7.12	(NF)Número Folhas 4-termo 3FND - o: sem poda o: reduced x: variante	93
7.13	(P%)Precisão 3-termo 3FND - o: sem poda o: reduced x: variante	94
7.14	(P%)Precisão 4-termo 3FND - o: sem poda o: reduced x: variante	94
9.1	Relação de Família	113
9.2	Ambiente de Aprendizado Relacional — PLI	127
10.1	Ordem Parcial Imposta pela Profundidade de Variáveis em $p(X1, X2, X3)$ $\leftarrow q(X1, X2), s(Y2, X1, Z3), r(X3, Z1, X2, Y1), t(X2, Z3, Z2, Z1)$	135
10.2	Ordem Parcial Imposta pela Profundidade de Variáveis em $p(X1, X2, X3)$ $\leftarrow q(X1, Z2), r(Z2, X3, Y1), s(Y1, Z1, X1), t(X2, Z1, Z2, Y2), u(Z1, Y1, Y2, Z3)$	136
10.3	Profundidade de Termo	143
11.1	Dois Exemplos Positivos de <i>cena</i>	149
11.2	Exemplos Positivos para o Aprendizado de <i>cena</i>	153
11.3	Exemplos Negativos para o Aprendizado de <i>cena</i>	154
11.4	Hierarquia de Animais	156
12.1	Aprendizado e Linguagens	166
A.1	Busca em Largura	171
A.2	Busca em Profundidade	171
A.3	Busca A^*	173
C.1	Árvore Gerada pelo Assistant Professional	183
C.2	Árvore Podada Gerada pelo TDIDT	184

Lista de Tabelas

5.1	Conjunto de Treinamento	35
5.2	Regras de Decisão para Classificação de Hipotireoidismo	36
5.3	Problemas Tratados pelo Assistant-86	49
6.1	Exemplos de Robôs Amigos e Robôs Inimigos	58
6.2	Exemplos de Robôs Amigos e Robôs Inimigos Estendidos com o Novo Atributo <i>mesma_forma</i>	59
6.3	Dezesseis Possíveis Operadores Construídos a Partir de Duas Features Booleanas — Op_1 Corresponde ao <i>and</i> Booleano e Op_7 Corresponde ao <i>or</i>	61
7.1	Características Medidas	81
7.2	3-termo 3FND	82
7.3	4-termo 3FND	83
7.4	Número de Iterações Necessárias para a Construção da Árvore de Decisão	91
7.5	3-termo 3FND	95
7.6	4-termo 3FND	96
7.7	Desempenho de <i>fringe</i> e <i>quine-fringe</i> no Aprendizado de Funções Completas	97
7.8	Desempenho de <i>fringe</i> e <i>quine-fringe</i> no Aprendizado de Funções Booleanas com 4 Atributos, a Partir do Conjunto de Treinamento Incompleto	98
7.9	Desempenho de <i>fringe</i> e <i>quine-fringe</i> no Aprendizado de Funções Booleanas com 5 Atributos, a Partir do Conjunto de Treinamento Incompleto	98
8.1	Lggs entre Pares de Literais Calculadas pelo Sistema <i>GOLEM</i>	106
10.1	Variáveis de Entrada/Saída de Literais em uma Cláusula	132
10.2	Profundidade de Variáveis e Cláusulas	134

11.1 Exemplos Positivos e Teoria do Domínio para <i>cena</i>	149
11.2 Exemplos Positivos e Negativos para o Aprendizado de <i>cena</i>	152
11.3 Teoria de Domínio para o Aprendizado de <i>cena</i>	155
11.4 Aprendizado de <i>cena</i> Usando Diferentes Exemplos Negativos	155
11.5 Exemplos Positivos e Negativos para o Aprendizado de <i>ako</i> pelo <i>GOLEM</i>	156
11.6 Teoria do Domínio para o Aprendizado de <i>ako</i> pelo <i>GOLEM</i>	157
11.7 Exemplos Positivos para o Aprendizado de <i>ako</i> pelo <i>FOIL</i>	161
11.8 Exemplos Negativos para o Aprendizado de <i>ako</i> pelo <i>FOIL</i>	162
11.9 Teoria do Domínio para o Aprendizado de <i>ako</i> pelo <i>FOIL</i>	162
11.10 Conceitos <i>ako</i> Apreendidos nos Experimentos com <i>FOIL</i>	163
11.11 Características dos Sistemas <i>PLI GOLEM</i> e <i>FOIL</i>	164
A.1 Estratégias Mais Comuns de Busca	173
B.1 Atributos para a Previsão de Lucro	175
B.2 Conjunto de Treinamento	175
B.3 Conjunto de Treinamento Agrupado por Valores do Atributo <i>concorrente</i>	175
C.1 Atributos Relacionados ao Desempenho do Programador	177
C.2 Atributos e seus Possíveis Valores	178
C.3 Classes nas quais programadores são classificados	178
C.4 Conjunto de Treinamento	179

Capítulo 1

Introdução

1.1 Contextualização

Aprendizado pode ser caracterizado como um processo multidimensional que, via de regra, ocorre através da aquisição de conhecimento declarativo, do desenvolvimento de habilidades motoras e cognitivas através de instrução e prática, da organização do conhecimento existente em representações mais efetivas, da descoberta de novos fatos e/ou teorias através de observação e experimentação ou, então, através da combinação e/ou composição dessas dimensões [Michalski 93].

Devido, principalmente, à sua natureza multidimensional e interdisciplinar; à existência ou não de conhecimento prévio para o aprendizado de novos conhecimentos, à não obediência a uma cronologia predefinida, a diferentes graus de especialização das informações disponíveis, às diferentes possibilidades — bem como às diferentes granularidades — de expressão do conhecimento aprendido, à sua natureza muitas vezes incremental e acumulativa, aprendizado se evidencia como um processo altamente complexo e ainda não completamente entendido.

Entre os vários paradigmas existentes para aprendizado tratado como uma área de pesquisa em Inteligência Artificial, o chamado *aprendizado indutivo* é o que tem sido mais largamente pesquisado e o que mais tem contribuído efetivamente para a implementação de sistemas de aprendizado de máquina. A partir de exemplos, vários tipos de tarefas podem ser aprendidas como, por exemplo, diagnóstico de doenças, previsão meteorológica, predição do comportamento de novos compostos químicos, predição de propriedades mecânicas de metais com base em algumas de suas propriedades químicas etc. Técnicas de aprendizado de máquina têm sido utilizadas em todas essas tarefas e em muitas outras, principalmente naquelas passíveis de serem enquadradas como aprendizado de conceitos a partir de exemplos.

A aplicação mais óbvia de aprendizado de máquina está na articulação de sistemas de aprendizado de máquina a mecanismos de aquisição de conhecimento, em sistemas baseados em conhecimento.

1.2 Objetivos e Pesquisa Desenvolvida

Para a expressão de qualquer paradigma de aprendizado são necessárias linguagens que descrevam objetos assim como linguagens que descrevam os conceitos aprendidos. Vários formalismos lógicos têm sido usados em sistemas de aprendizado indutivo para a representação de exemplos e conceitos. Em geral, distinguem-se dois tipos de descrição: *descrição baseada em atributos* e *descrição relacional*. Em uma descrição baseada em atributos objetos são descritos em termos de atributos e valores desses atributos. Em uma descrição relacional (também chamada de descrição estrutural) um objeto é descrito em termos de seus componentes e de relações entre eles.

Alguns dos algoritmos/sistemas de aprendizado indutivo que têm sido utilizados com relativo sucesso em um número razoável de aplicações — tais como o ID3 [Quinlan-86] e o AQ [Michalski 83] — usam linguagens baseadas em atributos, também conhecidas como linguagens proposicionais, para a representação de instâncias e conceitos. Na realidade, as linguagens de descrição de conceitos utilizadas por esses sistemas para a expressão de suas hipóteses induzidas, que tipicamente são, respectivamente, árvores de decisão e regras de produção, podem ser tratadas como variantes de linguagens baseadas em atributos.

Apesar do relativo sucesso dos sistemas proposicionais de aprendizado, tais sistemas, no entanto, são fortemente limitados pela linguagem usada na descrição de exemplos e conceitos, bem como pela forma limitada e inexpressiva (quando existente) do uso de teoria do domínio no processo de aprendizado.

A pesquisa desenvolvida nesta tese se direcionou à investigação de formas de ampliação dos limites do aprendizado indutivo, através da ampliação do poder representacional das linguagens de descrição de instâncias e conceitos utilizadas pelo sistema de aprendizado.

Inicialmente foi investigado empiricamente o uso de indução construtiva no aprendizado: o processo construtivo provoca mudanças nos espaços representacionais de instâncias e conceitos, através da criação de novos descritores, ampliando com isso o vocabulário da linguagem utilizada. Foram investigados diversos algoritmos de uma classe específica de indução construtiva, um deles por nós proposto, com o intuito também de encontrar uma abordagem teórica que permitisse entender o processo realizado, a fim de descobrir o melhor algoritmo dessa classe de algoritmos. Entretanto, este problema continua ainda em aberto. Muito embora o uso de indução construtiva amplie o espaço de descritores, o processo continua ainda limitado ao aprendizado daqueles conceitos expressáveis em lógica proposicional. As diversas implementações realizadas fazem parte de um ambiente experimental construtivo, que por sua vez é parte de um ambiente mais abrangente de experimentação com aprendizado proposicional.

A adoção, por sistemas de aprendizado, de linguagens lógicas de primeira ordem como linguagens representacionais, permite que relações — predicados — possam ser induzidas e, conseqüentemente, que o espaço dos conceitos passíveis de serem aprendidos seja aumentado.

A área de pesquisa de Programação Lógica Indutiva utiliza técnicas de programação

lógica em situações de aprendizado indutivo, objetivando a indução de programas lógicos, a partir de exemplos e de teoria do domínio. Sistemas que implementam programação lógica indutiva utilizam restrições de lógica de primeira ordem para expressar instâncias e conceitos, e viabilizam maneiras efetivas de utilização da teoria do domínio na expressão do conceito. Mesmo através do uso restrito e controlado de expressões de lógica de primeira ordem, a tarefa de aprendizado pode se tornar inviável, considerando as dimensões que o espaço de busca pode atingir; pesquisas em Programação Lógica Indutiva se concentram no controle do espaço de busca, processo que pode parcialmente ser viabilizado através do controle da linguagem utilizada.

Abordamos inicialmente Programação Lógica Indutiva através da revisão dos conceitos que subsidiam as principais restrições às linguagens de descrição de instâncias, conceitos e teoria do domínio, procurando algumas vezes um consenso entre as diferentes interpretações dadas a um mesmo conceito assim como sugerindo algumas alterações de maneira a unificá-las.

A abordagem empírica foi realizada através da experimentação com os sistemas *GOLEM* e *FOIL* e evidenciou, além das características típicas e limitações de cada sistema, a relevância que exemplos e teoria do domínio desempenham no processo de aprendizado de conceito que acontece sob o paradigma de programação lógica indutiva. A exigência — por ambos os sistemas — de expressão extensional da teoria do domínio, conduziu à uma proposta nossa de dois algoritmos “tradutores”. A implementação de um deles permite que tanto o *GOLEM* quanto o *FOIL* possam ser utilizados em situações de aprendizado em que a teoria do domínio esteja em forma intencional. As implementações desenvolvidas fazem parte de um ambiente experimental de Programação Lógica Indutiva que denominamos ambiente relacional.

1.3 Organização da Tese

O Capítulo 2 aborda o processo de Aquisição de Conhecimento para o desenvolvimento de Sistemas Baseados em Conhecimento e evidencia Aprendizado de Máquina como uma das maneiras de realizar Aquisição de Conhecimento. O Capítulo 3 discute Aprendizado e Aprendizado de Máquina através de uma proposta simples de modelo de aprendizado.

O Capítulo 4 apresenta uma taxionomia de aprendizado e evidencia o Aprendizado Indutivo como uma das possíveis formas de automatizar a Aquisição de Conhecimento. Examinadas as principais características que distinguem os métodos de aprendizado indutivo e apresenta os conceitos básicos que o subsidiam. Finaliza, discutindo a relevância do poder de expressão das linguagens de descrição de conceitos e de hipóteses, ressaltando a importância do uso da teoria do domínio na construção da expressão do conceito a ser aprendido.

O Capítulo 5 trata do aprendizado indutivo que utiliza linguagens proposicionais para a expressão de instâncias e hipóteses. Neste contexto, uma das mais representativas famílias de algoritmos/sistemas — a família TDIDT *Top Down Induction of Decision*

Trees — que usa árvores de decisão para a expressão das hipóteses induzidas, é caracterizada. O ambiente experimental desenvolvido para a família TDIDT é descrito, seus principais componentes detalhados e por fim, tal ambiente é contextualizado no ambiente de aprendizado proposicional existente.

Face às eventuais limitações quanto à abrangência do processo de aprendizado, provocadas pela adoção de linguagens representacionais com pouco poder de expressão, o Capítulo 6 discute a importância do Aprendizado Construtivo como uma forma de minimizar aquele problema, através da ampliação construtiva do poder de expressão da linguagem de descrição de hipóteses. Aprendizado Construtivo é abordado no domínio de funções booleanas. São apresentados e discutidos três *biases* para a construção automática de novas *features*, durante o aprendizado de funções booleanas. A estratégia de pré-processar o conjunto de treinamento, de maneira a estender o conjunto inicial de atributos é também proposta, com o objetivo de diminuir o número de iterações realizadas por um algoritmo iterativo de construção de *features*. Finalmente, as implementações dos algoritmos discutidos são contextualizadas no ambiente experimental construtivo desenvolvido, parte do ambiente experimental TDIDT, tratado no capítulo anterior.

O Capítulo 7 descreve detalhadamente os experimentos conduzidos relativos a avaliação do aprendizado construtivo, no domínio de funções booleanas. Os experimentos realizados, utilizando as implementações dos três *biases* para construção de *features*, são descritos e seus resultados analisados. Ainda no domínio de funções booleanas, o tratamento de dados incertos e/ou com ruídos é abordado através de experimentos envolvendo dois métodos de *poda* em árvores de decisão: *Redução do Erro* e *Variante*, que são detalhados e cujos resultados obtidos são comentados e analisados. Experimentos relacionados ao pré-processamento do conjunto primitivo de instâncias são descritos e resultados obtidos são analisados.

O Capítulo 8 apresenta os principais conceitos que subsidiam Programação Lógica Indutiva e discute a adoção desse subparadigma de aprendizado indutivo como uma das maneiras de se contornar as limitações impostas por um aprendizado direcionado por linguagens proposicionais, ainda que utilizadas em abordagens construtivas. O aprendizado em Programação Lógica Indutiva é abordado como o processo de busca no espaço de possíveis hipóteses.

O termo teoria do domínio é utilizado, neste trabalho, para nomear o conhecimento prévio existente sobre o domínio do problema. A incorporação desse tipo de conhecimento ao processo de aprendizado quase sempre contribui para uma expressão mais concisa e mais natural do conceito a ser aprendido. O Capítulo 9 investiga formas de utilização da teoria do domínio em sistemas baseados em programação lógica indutiva. Os sistemas PLI, que atualmente têm implementações disponíveis e eficientes (e.g. sistemas *GOLEM* e *FOIL*), exigem que a teoria do domínio esteja disponível *extensionalmente*, isto é, na forma de um conjunto de cláusulas unitárias *ground*. Esse pré-requisito limita, de certa forma, a utilização desses sistemas, uma vez que o seu uso se torna inviável em situações nas quais a teoria do domínio é especificada *intencionalmente*, isto é, na forma de um programa lógico. Neste Capítulo são apresentados três algoritmos que viabilizam a utilização, pelos sistemas PLI, de teoria do domínio

disponível apenas na forma *intencional*. Os principais predicados Prolog que implementam dois dos algoritmos, bem como alguns resultados, são apresentados e discutidos.

O Capítulo 10 discute formas de restringir as linguagens de representação de instâncias, conceitos e teoria do domínio, num ambiente de aprendizado baseado em PLI, de maneira a viabilizar o aprendizado de conceitos expressos em linguagens lógicas de primeira ordem, visando tornar exeqüível o processo de busca. Alguns conceitos utilizados pelo sistema *GOLEM* (e eventualmente pelo *FOIL*) para a restrição de linguagens são apresentados e analisados, buscando um consenso entre várias interpretações existentes. Alterações em alguns deles são sugeridas.

O Capítulo 11 apresenta e discute os dois sistemas PLI mais representativos: *GOLEM* e *FOIL*, respectivamente. Por fim, o Capítulo 12 apresenta as conclusões deste trabalho, evidenciando linhas de pesquisas a serem continuadas.

Capítulo 2

Aquisição de Conhecimento

2.1 Considerações Iniciais

O desenvolvimento de Sistemas Baseados em Conhecimento — SBC — é uma das áreas de Inteligência Artificial — IA — que tem recebido grande investimento em pesquisas e tem tido uma razoável projeção em diversos campos de conhecimento.

A construção de SBCs é um processo longo e demorado, em que uma das atividades mais problemáticas é a relacionada à eliciação do conhecimento do especialista humano (ou de qualquer outra fonte), processo conhecido como Aquisição de Conhecimento — AC. Uma das possíveis formas de agilizar a construção de SBCs é através da investigação/desenvolvimento de métodos mais efetivos de Aquisição de Conhecimento.

O objetivo deste capítulo é o de contextualizar o processo de Aquisição de Conhecimento e evidenciar Aprendizado de Máquina — AM — como uma das possíveis maneiras de, automaticamente, viabilizar este processo.

2.2 Sistemas Baseados em Conhecimento

Sistemas Baseados em Conhecimento podem ser definidos como programas de computador que usam conhecimento representado explicitamente, para resolver problemas. Assim, conhecimento e processo de resolução de problemas são pontos centrais no desenvolvimento de um SBC. É conveniente dividir o desenvolvimento de um SBC em três módulos principais, como mostra a Figura 2.1, na qual

1. *Base de Conhecimento (BC)* — contém o conhecimento específico do domínio da aplicação, o que inclui
 - fatos sobre o domínio
 - regras que descrevem relações no domínio
 - métodos e heurísticas para a resolução de problemas no domínio

2. *Motor de Inferência (MI)* — mecanismo responsável pelo processamento do conhecimento da BC usando, para isso, alguma linha de raciocínio. Implementa as estratégias de inferência e controle. Quando o conhecimento está expresso como regras, as estratégias de controle usadas pelo MI normalmente são

- *Backward Chaining* ou encadeamento para trás
- *Forward Chaining* ou encadeamento para frente

Quando o MI usa a estratégia *Backward Chaining*, ele pesquisa uma lista de hipóteses e procura reunir evidências para viabilizar a conclusão da validade de alguma(s) dela(s). Esta estratégia corresponde à pergunta: *É possível provar as hipóteses a partir dos dados disponíveis?* Se a estratégia de controle for *Forward Chaining*, o MI parte dos dados e, com base nas regras de conhecimento, deduz outras asserções, procurando chegar à solução do problema. Essa estratégia corresponde à pergunta: *O que é possível concluir a partir dos dados disponíveis?*

3. *Interface com o Usuário* — módulo responsável pela comunicação entre o usuário e o sistema. Deve fornecer, também, justificativas e explicações referentes às conclusões obtidas baseadas na BC, bem como do raciocínio utilizado.

O conjunto constituído pelo Motor de Inferência e Interface com o Usuário é, usualmente, chamado de *shell* do SBC [Monard 91]. É conveniente lembrar que os chamados Sistemas Especialistas — SE — constituem uma classe específica dos Sistemas Baseados em Conhecimento. De uma maneira geral, pode-se dizer que os SBCs são sistemas que resolvem problemas usando o conhecimento sobre o domínio do problema, enquanto que os SEs são SBCs que resolvem problemas do mundo real que, quando resolvidos pelo ser humano, requerem considerável habilidade, conhecimento e heurísticas [Rodrigues 93]. A Figura 2.2 [Waterman 86] mostra as características desses sistemas no contexto dos sistemas de Inteligência Artificial. Na literatura muitas vezes os termos SBC e SE são usados indistintamente. O presente trabalho adota essa abordagem.

Figura 2.1: Estrutura Básica de um SBC

A Figura 2.3 [Hayes-Roth 87] mostra um Sistema Baseado em Conhecimento tratado como uma aplicação computacional, com ambiente de desenvolvimento e ambiente de operação distintos. Durante o desenvolvimento e/ou refinamento de um SBC são usadas

Figura 2.2: SBC e SE

ferramentas para aquisição de conhecimento, manutenção da base de conhecimento, validação e projeto de interface, que fazem parte do ambiente de desenvolvimento.

Uma das características mais importantes em SBCs é a capacidade de resolver problemas que, para serem resolvidos, demandam informação especializada relativa a um determinado domínio de conhecimento. Um SBC deve ser capaz, também, de explicar seu comportamento e suas decisões ao usuário, ou seja, *porque* e *como* chegou a uma determinada solução. De uma maneira geral as perguntas *por que* referem-se a qual conhecimento respalda a conclusão; as perguntas *como*, por sua vez, referem-se aos passos de raciocínio seguidos para determinar a solução do problema. Esta característica é especialmente necessária quando o SBC lida com domínios incertos (diagnóstico médico, por exemplo); a explicação pode, de certa forma, aumentar o grau de confiança que o usuário deposita no sistema, ou então, ajudá-lo a encontrar alguma falha no raciocínio do sistema.

Uma outra característica freqüentemente necessária é a habilidade de lidar com incertezas e informações incompletas: a informação a respeito do problema a ser resolvido pode estar incompleta ou ser parcialmente confiável, bem como as relações no domínio do problema podem ser aproximadas. Espera-se, também, que um SBC seja flexível o suficiente para permitir, facilmente, a acomodação de novo conhecimento.

Atualmente, uma tendência que se observa no desenvolvimento de SBCs é a da incorporação de módulos que realizam aquisição automática de conhecimento, através da implementação de métodos de aprendizado de máquina.

2.3 O Processo de Aquisição de Conhecimento

A tarefa de aquisição de conhecimento, na grande maioria das abordagens existentes, consiste basicamente na transferência do conhecimento — informações e formas de condução do raciocínio — do especialista humano à base de conhecimento do SBC envolvendo, essencialmente, extração, análise e interpretação do conhecimento que o

Figura 2.3: SBC e seu Ambiente

especialista humano usa quando resolve problemas. Feigenbaum [Feigenbaum 80] introduziu o termo *engenharia do conhecimento* para definir essa redução do conhecimento específico de um domínio a um conjunto de fatos e regras. Muitas vezes, devido à complexidade de problemas reais, a compilação do conhecimento específico do domínio em um conjunto de fatos e regras é uma tarefa difícil, que exige um grande investimento em tempo e esforço.

Além da extração da informação necessária, o engenheiro do conhecimento deve traduzir o conhecimento do especialista para o esquema formal de representação usado pelo SBC. O engenheiro do conhecimento, via realimentação do especialista, deve repetidamente refinar o sistema até que este atinja um nível de desempenho próximo ao do especialista, quando da resolução do problema.

É importante lembrar que o processo de extração do conhecimento do especialista envolve um intenso questionamento que, em certas situações, pode interferir na sua própria percepção de como elabora o raciocínio. Geralmente um especialista acha difícil detalhar descrições de seu conhecimento e de como o usa; por outro lado, muitas vezes é também difícil para o engenheiro do conhecimento traduzir com exatidão aquele conhecimento, geralmente amplo e multifacetado, em uma linguagem de representação restrita. Devido a essas dificuldades, muitas vezes o conhecimento extraído pode ser inconsistente (como consequência de diferenças individuais entre especialistas), incompleto e impreciso.

2.4 Modalidades de Aquisição de Conhecimento

Existe um vasto conjunto de métodos e ferramentas que facilitam a tarefa de aquisição de conhecimento. A disponibilidade dessa variedade de métodos reflete o fato da aquisição de conhecimento ser um processo multidimensional, podendo ocorrer em diferentes estágios do desenvolvimento de um SBC, e podendo envolver vários tipos de conhecimento. Em [Wielinga 87] os métodos e técnicas para Aquisição de Conhecimento são separados em quatro grupos, visualizados na Figura 2.4, pg. 11 e descritos a seguir.

1. A aquisição do conhecimento é realizada pelo engenheiro de conhecimento; é ele quem extrai, estrutura e formaliza o conhecimento do especialista.
2. O próprio especialista da área de conhecimento cria a BC através do uso de um programa de edição. Esta abordagem é mais limitada do que a anterior. Em determinadas situações, entretanto, seu uso pode ser o mais conveniente, particularmente quando a estrutura da base de conhecimento for pré-definida e facilmente inteligível pelo especialista e o motor de inferência for fixo.
3. O processo de aquisição de conhecimento é feito com o uso de um programa de indução. Este programa é usado para gerar o conhecimento, através da análise de dados de entrada. Para que esta abordagem possa ser cogitada, é necessário, entre outros, que existam dados significativos disponíveis. No domínio de diagnósticos clínicos pode ser altamente conveniente examinar uma grande quantidade de casos, com o objetivo de identificar as características que determinam uma certa doença. O conhecimento gerado desta forma (identificação das características determinantes) pode ser usado, subseqüentemente, para diagnosticar certas doenças com base naquelas características.
4. O processo de aquisição de conhecimento é feito com o uso de um programa de aprendizado. Nesta abordagem o programa consulta o especialista, ou então é alimentado com informações textuais ou exemplos. Gera — ou então refina — o SBC com base nesta interação. Esta abordagem encontra-se a nível de pesquisa e seu potencial ainda não se mostrou adequado para uso prático.

A classificação proposta em [Boose 89], por sua vez, agrupa os métodos e as técnicas para AC em

1. *manualis* — que tipicamente consistem de entrevistas e análise de protocolos
2. *baseadas em computador*
 - (a) *interativas* — que englobam, principalmente, ferramentas que entrevistam o especialista, que fazem análise textual, que extraem e analisam conhecimento de múltiplas fontes separadamente e as combinam para uso
 - (b) *baseadas em aprendizado* — as quais, via de regra, generalizam situações específicas em conceitos

Figura 2.4: Modalidades de Aquisição de Conhecimento [Wielinga-87]

Os SBCs classificados como de *primeira geração* contêm Bases de Conhecimento formadas com informações extraídas através de um processo de articulação direta. Neste processo, o objetivo do engenheiro do conhecimento é extrair o *know-how* do especialista humano do domínio e convertê-lo na representação de conhecimento utilizada pelo sistema. Via de regra, o conhecimento é extraído através de métodos manuais de aquisição (por exemplo entrevista) e é categorizado como *conhecimento de superfície*. Os SBCs de *segunda geração* são caracterizados pelo uso de *conhecimento profundo* (conhecimento associado ao modelo do domínio do problema) e pela aplicação de técnicas de aprendizado de máquina para (pelo menos parcialmente) automatizar o processo de Aquisição de Conhecimento [Lavrač 89].

O que vem sendo reforçado pelo número cada vez maior de utilização de SBCs em aplicações reais, é a necessidade de tais sistemas viabilizarem diferentes níveis de integração. Assim, ênfase tem sido dada à integração

- do sistema de Aquisição de Conhecimento e da shell do SBC [Eshelman 87]
- do sistema de Aquisição de Conhecimento e do sistema de Base de Dados
- do conhecimento advindo da participação de vários especialistas humanos, conhecimento este que deve ser integrado ou em um modelo consistente de domínio ou em um modelo que evidencie as abordagens conflitantes no domínio [Shaw 88]
- das várias estratégias de AM utilizadas
- do sistema de Aprendizado de Máquina ao Sistema de Aquisição de Conhecimento

Como o conhecimento tem natureza inerentemente dinâmica, o processo de aquisição de conhecimento nunca termina: para a maioria dos domínios, não existe um estado final de conhecimento completo. Este fato requer que sistemas que realizam aquisição de conhecimento viabilizem revisões, reestruturações e reorganizações da base de conhecimento, de maneira a possibilitar mudanças.

O processo de Aquisição de Conhecimento não se restringe, unicamente, à fase de construção da Base de Conhecimento. Na realidade, uma tarefa considerada difícil e que demanda um grande investimento em tempo e esforço é a relacionada à manutenção e à consistência da BC. Esta atividade implica a modificação do conhecimento representado, motivada por mudanças no domínio, bem como a detecção e correção de erros existentes no conhecimento que já está representado.

2.5 Considerações Finais

Neste capítulo foi apresentada, inicialmente, a estrutura geral de um SBC e explicitada, de forma pictórica, sua integração com os seus ambientes de desenvolvimento e de operação.

O processo de Aquisição de Conhecimento relacionado à construção/manutenção da Base de Conhecimento foi discutido e algumas das inúmeras taxonomias de métodos

de Aquisição de Conhecimento existentes foram apresentadas. Em todas elas Aprendizado de Máquina é evidenciado como uma das maneiras de automatizar (pelo menos parcialmente) o processo de Aquisição de Conhecimento [Nicoletti 92c].

É conveniente notar, entretanto, que a construção da Base de Conhecimento através do uso exclusivo de procedimentos de AM não é muito aconselhável — engenheiros do conhecimento querem participar do processo de construção da Base, de maneira a poderem adicionar, eliminar ou modificar fatos e regras. Este fato justifica, de acordo com [Morik 93], que técnicas manuais de AC e AM sejam integradas.

Capítulo 3

Aprendizado de Máquina

3.1 Considerações Iniciais

Muito embora quase sempre contextualizada como uma das maneiras de automaticamente adquirir conhecimento para alimentar um SBC, a área de AM é bem mais ampla, no sentido que a pesquisa em AM se dedica também à investigação de formas de condução do raciocínio humano, de formas de organização do conhecimento existente, de diferentes estratégias de raciocínio e suas articulações, assim como à análises teóricas e à exploração do espaço de possíveis métodos. Além de SBCs, AM já tem sido utilizada, entre outros, em sistemas de planejamento, sistemas tutores inteligentes, sistemas sensoriais (e.g. visão, locomoção), robôs autônomos e descoberta de conhecimento em bases de dados.

Neste capítulo é discutido de uma maneira ampla o que se entende por aprendizado e são evidenciados alguns mecanismos, entre os muitos que existem, que habilitam o Aprendizado de Máquina, ou seja, máquinas que incorporam a capacidade de aprendizado.

Um modelo simples de Aprendizado de Máquina (envolvendo quatro unidades funcionais: o Ambiente, a Base de Conhecimento, o Módulo de Desempenho e o Módulo de Aprendizado) é descrito e analisado, com o objetivo de apresentar uma estrutura despojada para o entendimento do processo de Aprendizado de Máquina.

3.2 Aprendizado e Aprendizado de Máquina

Considerando o fato de aprendizado ser um processo altamente complexo, não só por reunir habilidades/conceitos advindos de várias áreas de conhecimento, mas também por envolver diferentes níveis de abstração, não há, em relação a ele, definição única. São encontradas, na literatura, definições que se diferenciam entre si pela ênfase dada a diferentes aspectos do processo de aprendizado.

Em [Simon 83] aprendizado é definido como *o processo através do qual um sistema*

melhora o seu desempenho. Esta definição parte do pressuposto que o sistema está tentando realizar uma tarefa. A melhoria no desempenho pode se dar através da aplicação de novos métodos e conhecimentos, ou então através do melhoramento dos métodos e conhecimentos existentes, de maneira a torná-los mais rápidos e precisos.

Uma abordagem mais limitada de aprendizado, geralmente adotada por pessoas que trabalham especificamente com SBCs, é a de que aprendizado é a aquisição de conhecimento explícito. Muitos SBCs representam conhecimento como um conjunto de regras que necessitam ser adquiridas, organizadas e estendidas. Esta abordagem enfatiza a importância de tornar explícito o conhecimento adquirido, de maneira que ele possa facilmente ser verificado, modificado e explicado.

Aprendizado pode também ser abordado como um processo de aquisição de habilidades — pesquisas que relacionam IA e psicologia cognitiva têm se preocupado em descobrir qual é o conhecimento necessário que promove o desenvolvimento de habilidades, na realização de determinadas tarefas.

Sob uma outra perspectiva, aprendizado pode ser focalizado como o processo relacionado à *construção de teoria, construção de hipóteses e inferência indutiva* [Cohen 82]. Trabalhos desenvolvidos em construção de teoria estão relacionados ao entendimento de como cientistas/pesquisadores desenvolvem teorias que descrevem e explicam determinados fenômenos. Um aspecto fundamental no desenvolvimento de teorias é o relacionado à construção de hipóteses — atividade associada à busca de uma ou mais hipóteses plausíveis, que expliquem um determinado conjunto de fatos no contexto de uma teoria mais geral. Um outro aspecto igualmente relevante é o da inferência indutiva — processo que infere leis gerais a partir de um conjunto específico de exemplos.

Em [Carbonell 83] as várias perspectivas de aprendizado são englobadas na definição que o caracteriza como um processo que envolve a aquisição de novos conhecimentos declarativos, o desenvolvimento de habilidades motoras e cognitivas através de instrução e prática, a inserção e organização do novo conhecimento em estruturas gerais, bem como a descoberta de novos fatos e teorias através de observações e experimentação.

Um sistema inteligente deve, pois, ser capaz de formar conceitos, isto é, classes de entidades unidas por algum princípio. Tal princípio pode ser o fato das entidades terem um uso ou objetivo comum ou, simplesmente, terem características perceptuais similares. Para a utilização do conceito, o sistema deve também desenvolver métodos eficientes de reconhecimento de pertinência de uma dada entidade ao conceito. O estudo e a modelagem de processos pelos quais um sistema adquire, refina e diferencia conceitos são objetivos da área de pesquisa chamada de *Aprendizado de Conceito*, uma subárea de *Aprendizado de Máquina* — AM.

Em *Aprendizado de Conceito*, o termo *conceito* usualmente indica uma classe de equivalência de entidades, que pode ser descrita via um conjunto relativamente limitado de expressões e que deve ser suficiente para a diferenciação entre conceitos. Entidades individuais na classe são chamadas de instâncias do conceito. O fato do conceito ser definido como uma classe de equivalência torna cada instância de uma classe igualmente representativa do conceito em questão; tal definição estabelece, também, os limites da descrição daquele conceito — uma entidade satisfaz ou não satisfaz o conceito.

Pesquisas em AM se direcionam ao estudo dos diferentes mecanismos que habilitam máquinas com a capacidade de aprendizado, o que inclui, entre outros

- exploração das formas alternativas do uso de indução
- investigação da influência, no aprendizado, da representação do conhecimento adotada
- determinação do alcance e limitação dos algoritmos existentes
- experimentação com o volume e nível de abstração da informação inicialmente disponível ao sistema de aprendizado
- tratamento dado a informações incompletas ou incertas
- desenvolvimento de técnicas gerais aplicáveis a diferentes domínios de conhecimento
- formas de viabilizar o aprendizado incremental e multiconceitual
- métodos automáticos e métodos direcionados pelo usuário
- articulação de diferentes estratégias de aprendizado
- utilização da teoria do domínio em processo de aprendizado

Aprendizado de Máquina, da mesma forma que representação do conhecimento e raciocínio, permeia todas as áreas de problemas de IA: resolução de problemas, prova automática de teoremas, raciocínio analógico e não-monotônico, processamento de linguagem natural, reconhecimento de voz, visão, robótica, planejamento, jogos, reconhecimento de padrões, sistemas especialistas etc. Em princípio, progressos em AM podem contribuir para o desenvolvimento de qualquer uma dessas áreas.

3.3 Um Modelo de Aprendizado

Com base na definição de aprendizado proposta por Simon apresentada na Seção 3.2, em [Cohen 82] é proposto um modelo simples de sistema de aprendizado, mostrado na Figura 3.1, onde quadrados representam informação declarativa e retângulos representam procedimentos. As flexas indicam a direção do fluxo de dados através do sistema de aprendizado.

O módulo de aprendizado recebe informação do ambiente e a usa, juntamente com o retorno fornecido pelo módulo de desempenho, no melhoramento da Base de Conhecimento; a maneira como o módulo de aprendizado recebe a informação e modifica o conhecimento da base é conhecida como *estratégia de aprendizagem*.

Se o objetivo do sistema de aprendizado for aquisição de conhecimento, o módulo de desempenho realiza inferências usando o conhecimento refinado da base; se o objetivo

Figura 3.1: Um Modelo de Sistema de Aprendizado

for a melhoria de desempenho, este mesmo módulo executa componentes procedurais (orientação heurística, poda em árvores, etc.). Os resultados obtidos podem, então, ser comparados com os resultados ideais (os produzidos pelo especialista, por exemplo).

Após o término da tarefa, uma realimentação deve acontecer imediatamente, de maneira que o conhecimento possa ser adquirido. A realimentação é feita via *crítica* a qual, após comparação dos resultados do módulo de desempenho, atribui um crédito àquelas partes da BC que tiveram um bom desempenho, de maneira a fortalecê-las; por outro lado, aqueles componentes responsáveis por um desempenho pobre ou incorreto são enfraquecidos através de atribuição de débito. A realimentação na forma de atribuição de crédito/débito é uma dentre as muitas técnicas utilizadas como crítica.

Embora o modelo proposto seja despojado e omita funções importantes, seu uso pode sistematizar a análise dos sistemas de aprendizado de acordo com a maneira como tais sistemas viabilizam as quatro unidades funcionais. Em uma aplicação tanto o ambiente quanto a Base de Conhecimento e o módulo de desempenho, determinam a natureza de um determinado problema de aprendizado e, conseqüentemente, quais funções específicas o módulo de aprendizado deve executar.

3.3.1 O Ambiente

O projeto de um sistema de aprendizado é fortemente afetado pelo tipo de informação disponível ao sistema — particularmente o *nível de abstração* e a *qualidade* dessa informação.

O *nível* da informação refere-se ao grau de generalização da informação, com relação à necessidade do módulo de desempenho. Informação de alto nível é informação abstrata relevante a uma ampla classe de problemas. Informação de baixo nível é informação detalhada relevante a um único problema. A tarefa do módulo de aprendizado pode ser vista como a de nivelar a informação fornecida pelo ambiente, de forma a viabilizar o seu uso pelo módulo de desempenho quando da execução de suas funções.

Assim sendo, quando informações com alto grau de abstração forem fornecidas ao sistema de aprendizado, ele deve completar os detalhes ausentes de maneira que, em situações específicas, o módulo de desempenho possa fazer uso da informação fornecida.

Analogamente, se forem fornecidas informações muito específicas a respeito de como atuar em determinadas situações, o módulo de aprendizado deve generalizar — igno-

rando detalhes irrelevantes — as informações fornecidas em uma regra que pode ser usada para guiar o módulo de desempenho em situações mais gerais.

Desde que seu conhecimento é imperfeito, o módulo de aprendizado não sabe — de antemão — como preencher os detalhes ausentes ou, então, como ignorar detalhes irrelevantes. Conseqüentemente, ele deve adivinhar, ou seja, *formar hipóteses* sobre como nivelar a informação. O sistema deve receber alguma realimentação que o habilite a avaliar e, se necessário, revisar suas hipóteses — esta é a forma pela qual um sistema de aprendizado geralmente aprende: tentativa e erro.

No chamado *aprendizado por exemplos* (Capítulo 4, pg. 22), onde a informação fornecida pelo ambiente é bem específica e detalhada, o módulo de aprendizado aprende construindo hipóteses mais gerais. Já no *aprendizado por explicação* [DeJong 86], a teoria geral do domínio de conhecimento fornecida pelo ambiente é compilada de maneira a se obter hipóteses mais especializadas.

A *qualidade* da informação tem, por sua vez, uma influência significativa na tarefa de aprendizado. Indução, por exemplo, é fortemente dependente da qualidade, relevância e representatividade das instâncias de treinamento fornecidas ao sistema. Aprendizado utilizando métodos indutivos torna-se complexo, por exemplo, quando as instâncias de treinamento têm *ruído* ou, então, são instâncias expressas usando características não relevantes ao conceito a ser aprendido. Um sistema de aprendizado indutivo pode, por exemplo, se alimentado por um conjunto de instâncias não significativas, aprender que *se um indivíduo usa meias verdes e tem relógio digital, então ele tem hepatite*.

3.3.2 A Base de Conhecimento

A informação usada pelo sistema de aprendizado necessita ser representada de alguma forma. A escolha da representação é crítica na determinação da solução. Dois aspectos da Base de Conhecimento afetam o projeto de sistemas de aprendizado: a sua *forma* e o seu *conteúdo*.

Com relação à *forma*, muito embora vários sistemas representacionais têm sido utilizados, tais como regras de produção, gramáticas, funções Lisp, redes semânticas, *frames* etc., uma grande maioria de trabalhos em Aprendizado de Máquina tem adotado uma das duas formas representacionais básicas: vetores de atributos e cálculo de predicado. A expressividade das linguagens de descrição de instâncias e de hipóteses induzidas é essencial para o sucesso de qualquer algoritmo de aprendizado e será abordada mais detalhadamente nos próximos capítulos. Um sistema representacional deve ser sempre analisado com relação a várias características, arroladas a seguir.

- *expressividade de representação* — em qualquer sistema de IA é importante o uso de uma representação na qual o conhecimento possa ser facilmente expresso. Vetores de atributos, por exemplo, são adequados para a descrição de objetos que não têm estrutura interna. Tal representação descreve objetos como um conjunto fixo de atributos, cada um deles podendo assumir um conjunto finito de valores. Para a descrição de situações e de objetos estruturados, o Cálculo de Predicado

é mais adequado.

- *facilidade de inferência* — o custo computacional na realização de inferências deve ser cuidadosamente considerado. Um tipo de inferência necessário em sistemas de aprendizado é a da comparação de duas descrições para determinar a sua equivalência. Este tipo de comparação é feito, facilmente, se a representação adotada for a de vetor de atributos; se as descrições estiverem representadas em Cálculo de Predicado, porém, o custo computacional será bem maior. Uma vez que sistemas de aprendizado realizam buscas em amplos espaços de hipóteses, o custo das comparações pode limitar a extensão dessas buscas.
- *modificabilidade da Base de Conhecimento* — devido à sua natureza, um sistema de aprendizado deve estar sempre modificando partes de sua Base de Conhecimento. Conseqüentemente, a maioria dos sistemas de aprendizado emprega representações que facilitam a tarefa de inserção de conhecimento na Base. Muitas vezes, a inserção de novo conhecimento na BC implica substancial revisão e integração de maneira a preservar sua consistência. Esse problema, porém, está apenas começando a ser estudado [Morik 93].
- *extensibilidade da representação* — trata-se da construção de sistemas representacionais que permitam meta-conhecimento como parte da Base de Conhecimento de forma a permitir que o sistema examine e altere sua própria representação, adicionando termos e estruturas representacionais.

Com relação ao *conteúdo* sabe-se que um sistema não aprende a partir do nada, isto é, sem ter qualquer conhecimento anterior. O sistema de aprendizado deve usar algum conhecimento para entender a informação fornecida pelo ambiente, para elaborar hipóteses, para testá-las e refiná-las. Assim sendo, talvez seja mais apropriado abordar um sistema de aprendizado como um sistema que amplia e aperfeiçoa o conhecimento que já possui.

3.3.3 O Módulo de Desempenho

O módulo de desempenho é o centro de todo sistema de aprendizado, uma vez que são as suas ações que o módulo de aprendizado busca refinar. Existem três características importantes a ele relacionadas: *complexidade*, *realimentação* e *transparência*.

- A *complexidade* da tarefa é de extrema importância — tarefas complexas requerem mais conhecimento do que tarefas mais simples. À medida que as tarefas se tornam mais complexas e a Base de Conhecimento cresce em tamanho, os problemas de integração do novo conhecimento e o de diagnóstico de conhecimento incorreto tornam-se mais complexos. O problema de integração de novo conhecimento ao já existente é, sob essa perspectiva, difícil, uma vez que o sistema de aprendizado deve considerar todas as interações possíveis entre os dois.
- O problema do diagnóstico do conhecimento incorreto, também conhecido como o problema de atribuição de crédito, pode se tornar igualmente difícil em sistemas que realizam uma seqüência de ações antes de receber qualquer realimentação.

- A segunda característica importante do módulo de desempenho é a *realimentação* fornecida ao módulo de aprendizado. Todos os sistemas de aprendizado devem, de alguma forma, avaliar as hipóteses propostas pelo módulo de aprendizado. Uma técnica freqüentemente usada é a de ter o ambiente, freqüentemente um instrutor, fornecendo um padrão externo de desempenho. Através de observações de quão próximo o módulo de desempenho está do padrão, o sistema pode avaliar suas hipóteses. Em sistemas que aprendem um único conceito, o padrão de desempenho é a classificação correta de cada instância de treinamento.
- A terceira característica relevante à tarefa de desempenho é a *transparência* do módulo de desempenho. Para o módulo de aprendizado atribuir crédito ou débito à porções individuais da Base de Conhecimento, é conveniente que ele tenha acesso às ações internas do módulo de desempenho.

Devido à alta complexidade dos processos de aprendizado, o desenvolvimento de sistemas de aprendizado versáteis é um objetivo a ser atingido a longo prazo. Entretanto, com o investimento no desenvolvimento de SBCs, a implementação de algumas formas de aprendizado de máquina, mesmo que limitadas, têm sido necessárias.

3.4 Considerações sobre Aprendizado de Máquina

Não existe um modelo único que garanta o aprendizado efetivo. Esse fato pode ser confirmado pela proliferação de métodos existentes implementados em vários sistemas disponíveis, como mostrado nos capítulos que se seguem. São necessários não apenas novos métodos, mas também um melhor entendimento do critério para decidir qual é o mais apropriado para um determinado problema de aprendizado.

Sistemas começam a aprender um conceito, freqüentemente, a partir de instâncias positivas. Como consequência, pois, sistemas tendem a aprender o que já é sabido. Não é ainda possível aprendizado irrestrito. Via de regra, é necessário fornecer ao sistema uma indicação do que ele está tentando aprender, tal como uma heurística inicial, de maneira a deflagrar a busca.

A capacidade de expressão do conhecimento em um nível que facilita a modificação é muito importante ao aprendizado; a escolha da representação muda de acordo com o domínio. Associado à representação, está a facilidade de entendimento: é importante entender as regras utilizadas pelo sistema.

Não existe, ainda, um algoritmo que seja completamente independente do domínio. Assim sendo, a escolha do algoritmo mais apropriado para um determinado domínio é razoavelmente complexa, e o desenvolvimento de um novo algoritmo, mais complexo ainda.

A identificação das partes do sistema responsáveis por sucesso e/ou falha é particularmente difícil, especialmente no aprendizado de regras ou conceitos disjuntos. Existem muitas maneiras de atribuição de créditos e falhas, e algumas são, obviamente, melhores que outras. A crítica deve ter alguma maneira de compensar todas aquelas regras que

contribuem para o sucesso ou então de penalizar aquelas que contribuem para a falha. As perguntas: as regras mais gerais devem receber crédito/falha com um peso maior que as regras especializadas? Se sim, quanto maior?, ficam, ainda, sem resposta.

Aprendizado em domínios com ruído não é tão problemático atualmente devido à existência de algumas técnicas. Existem, entretanto, alguns domínios onde a tolerância a erros é baixa e o sistema não pode ser usado. É especialmente difícil usar a técnica de generalização/especialização em instâncias que mudam com o tempo.

3.5 Considerações Finais

Neste capítulo foram apresentadas e discutidas algumas das muitas perspectivas sob as quais aprendizado pode ser abordado, bem como os principais mecanismos que implementam alguns dos aspectos de aprendizado em um sistema de AM.

Um modelo simples de aprendizado foi apresentado e discutido, com o objetivo de introduzir os principais conceitos envolvidos num processo de aprendizado, visando a sua automação.

É conveniente lembrar que a definição de aprendizado proposta por Simon [Simon 83], muito embora seja adotada por muitos autores, é criticada por excluir o aprendizado *sem propósito* — a definição de Simon requer que a finalidade do aprendizado seja conhecida.

Capítulo 4

Paradigma Indutivo de Aprendizado de Máquina

4.1 Considerações Iniciais

Na literatura há diversas taxionomias de sistemas de AM. Algumas das várias razões que dão origem a esta diversidade de classificações são: diferentes estratégias de aprendizado adotadas, existência de diferentes representações de conhecimento, domínios de aplicação variados etc. Este capítulo apresenta uma das taxionomias de AM e avalia as principais características do paradigma de aprendizado indutivo. Introduz os conceitos relevantes de aprendizado indutivo por exemplos e discute a relevância das linguagens de representação de instâncias, de hipóteses e de teoria do domínio no aprendizado.

4.2 Taxionomia de Aprendizado de Máquina

Pode-se identificar quatro paradigmas principais nas pesquisas de AM [Carbonell 89]

1. *aprendizado indutivo* — aquisição de conceitos através de conjuntos de exemplos
2. *aprendizado analítico* — aprendizado baseado em explicações e certas formas de métodos de aprendizado analógicos e baseados em casos
3. *algoritmos genéticos* — sistemas classificadores
4. *métodos de aprendizado conexionistas* — redes neurais

A pesquisa desenvolvida neste trabalho está confinada ao paradigma de aprendizado indutivo, usando as linguagens de descrição de instâncias e de conceitos para caracterizá-lo em subparadigmas. Vale notar, neste contexto, que vários possíveis subparadigmas podem ser explicitados em outros paradigmas. A apresentação dos mesmos, no entanto, foge aos objetivos deste trabalho.

4.3 O Aprendizado Indutivo

O paradigma mais difundido de aprendizado indutivo de máquina é conhecido como *aprendizado indutivo a partir de exemplos* e consiste na indução de uma descrição geral do conceito a partir de um dado conjunto de exemplos¹ — *exemplos positivos* — e (usualmente) contra-exemplos — *exemplos negativos* — do conceito.

O conjunto de exemplos, conhecido como *conjunto de treinamento* é fornecido, geralmente, pelo instrutor ou ambiente. A indução do conceito corresponde a uma busca em um espaço de hipóteses, de forma a encontrar aquelas que *melhor* classificam os exemplos. Neste contexto, *melhor* pode ser definido em termos de precisão e compreensibilidade.

De uma maneira geral, um sistema que aprende a partir de exemplos recebe como dados informações na forma de situações específicas, cada uma delas devidamente classificada (geralmente por especialista humano no domínio) e produz, como resultado, uma hipótese que generaliza aquelas situações inicialmente fornecidas. Uma variante do aprendizado por exemplos é conhecida como aprendizado por experiência. Ambos usam o mesmo processo indutivo. No aprendizado por exemplos, entretanto, o conjunto de treinamento é cuidadosamente escolhido, enquanto que no aprendizado por experiência, o conjunto de treinamento é composto de dados provenientes de várias situações experimentadas anteriormente.

A tarefa de aprendizado consiste na construção da descrição do conceito — *hipótese* — a partir da qual todos os exemplos positivos podem ser rederivados por instanciação universal mas que, entretanto, nenhum dos exemplos negativos fornecidos previamente pode ser rederivado pelo mesmo processo [Carbonell 89]. Informalmente, o processo de inferência indutiva pode ser abordado de outra forma [Shaw 90, pg. 49].

...inferência indutiva é um processo de solução de problemas que obtém soluções — descrições do conceito indutivo — através de busca e de uma seqüência de transformações. Generalização e especialização são passos essenciais quando se faz inferência indutiva. Se a descrição do conceito Q é mais geral que a descrição do conceito P, a transformação de P para Q é chamada generalização, e a transformação de Q para P é chamada de especialização. P é dito ser mais geral que Q se (e apenas se) P cobre mais instâncias que Q. Inferência indutiva pode ser vista como um processo que faz iterações sucessivas de generalização e especialização nas descrições do conceito e é consistente com todos os exemplos. Então, relações de generalização/especialização entre descrições de conceito fornecem a estrutura básica para guiar a busca em aprendizado indutivo.

¹Também chamados de instâncias.

4.4 Características do Aprendizado Indutivo

Entre as principais características que distinguem os métodos de aprendizado indutivo encontram-se as arroladas a seguir.

Aprendizado Incremental e Não Incremental No caso incremental, a expressão do conceito vai sendo construída exemplo a exemplo e implica constante revisão; um novo exemplo pode, eventualmente, causar um rearranjo da expressão do conceito, para que este possa classificá-lo. A expressão do conceito vai se modificando à medida que os exemplos vão se tornando disponíveis. Já no caso não incremental, o conjunto de treinamento deve estar disponível desde o início do processo, uma vez que a expressão do conceito é induzida considerando todos os exemplos de uma vez.

Um exemplo de aprendizado incremental por exemplos é o realizado pelo programa *ARCHES* [Winston 75] que, incrementalmente, aprende o conceito de um arco, através de exemplos e contra-exemplos fornecidos por um instrutor, conforme mostrado na Figura 4.1. Os exemplos fornecidos são sequencialmente processados pelo sistema que, a cada novo exemplo, “atualiza” o conceito sendo aprendido. No exemplo, após o sistema ter processado os quatro exemplos, o conceito de arco por ele aprendido pode informalmente ser traduzido por

1. um arco consiste de três partes, sendo dois pilares e um lintel
2. os dois pilares têm forma retangular, enquanto que o lintel pode ser uma classe de polígonos. Isto pode ser concluído dos exemplos 1 e 4
3. os dois pilares não podem se tocar. Isso pode ser concluído do exemplo negativo 2
4. os dois pilares devem suportar o lintel. Isso pode ser concluído do exemplo negativo 3

Um exemplo de aprendizado não incremental pode ser visto na Seção 5.2, pg. 34.

Aprendizado de Um Conceito e Aprendizado de Vários Conceitos Relacionado à habilidade de um sistema poder aprender a expressão de apenas um ou então de vários conceitos de uma vez.

Uso de Teoria do Domínio Se um sistema não tem informação a respeito do problema de aprendizado sendo tratado, supostamente ele deve induzir a expressão do conceito apenas a partir dos exemplos disponíveis. Entretanto, para que as soluções de problemas difíceis de aprendizado sejam encontradas, é fundamental que um volume substancial de conhecimento sobre o problema esteja disponível, de maneira a subsidiar a indução do conceito. Esse conhecimento prévio existente é conhecido como *teoria do domínio* ou *conhecimento de fundo*. Obviamente, sistemas que fazem uso de teoria do domínio aprendem conceitos que são expressos muitas vezes em função (mas não apenas) do conhecimento disponível pela teoria do domínio.

1. É um arco

2. Não é um arco

3. Não é um arco

4. É um arco

Figura 4.1: Seqüência de Exemplos e Contra-exemplos para o Aprendizado do Conceito de Arco

A Linguagem de Descrição Em inferência indutiva, os exemplos, a teoria do domínio e as hipóteses formuladas necessitam ser expressos em alguma linguagem e geralmente são usadas linguagens formais. Algumas vezes os exemplos são mais facilmente expressos em uma linguagem, as hipóteses em outra e a teoria do domínio em uma terceira. As linguagens são denominadas

- *linguagem de descrição de instâncias* — usada para descrever os exemplos do conjunto de treinamento
- *linguagem de descrição de conceitos* — usada pelo algoritmo de aprendizado para construir as hipóteses do conceito sendo aprendido
- *linguagem de descrição da teoria do domínio* — usada para descrever o conhecimento disponível ao sistema de aprendizado

4.5 Critérios de Sucesso

Entre os critérios mais usuais para se medir o sucesso de um sistema de aprendizado estão

- *precisão de classificação* — é geralmente definida como o percentual de instâncias corretamente classificadas pela hipótese induzida $\mathcal{H}$. A precisão pode estar relacionada a instâncias não conhecidas (não presentes no conjunto de treinamento) ou então a instâncias presentes no conjunto de treinamento (relevante apenas quando o algoritmo de aprendizado permite a geração de hipóteses incompletas e inconsistentes).
- *transparência da descrição induzida do conceito* — em muitos domínios de aplicação (por exemplo diagnóstico médico) é fundamental que a descrição do conceito, gerada por um sistema de aprendizado, seja compreensível pelo ser

humano. O entendimento do conceito, por parte do ser humano, não apenas aumenta a credibilidade do sistema de aprendizado, mas também permite que o conceito possa ser assimilado e utilizado pelo especialista humano. Em muitas situações a transparência da descrição é medida pelo número de descritores e operadores usados na descrição do conceito. O princípio da *Navalha de Occam*² tem sido freqüentemente aplicado em AM, para descobrir a hipótese mais simples consistente com o conjunto de treinamento utilizado [Blumer 87].

- *complexidade computacional* — está relacionada com os recursos computacionais necessários (tempo e espaço) para realizar o aprendizado. É possível distinguir dois tipos de complexidade
 - complexidade de geração — recursos necessários para induzir uma descrição do conceito através de exemplos
 - complexidade de execução — complexidade em classificar uma instância usando uma hipótese induzida.

4.6 Conceitos Básicos

A fim de definir formalmente *aprendizado indutivo a partir de exemplos* é necessário primeiro introduzir alguns conceitos básicos. As definições que se seguem foram extraídas de [Bratko 89, Lavrač 94].

Definição 4.6.1 (*Conceito C*) Se U é o conjunto universo de objetos, o conceito C é um subconjunto de objetos de U .

Aprender um conceito C significa adquirir a habilidade de reconhecer se $x \in C, \forall x \in U$ e pode ser definido como

Definição 4.6.2 (*Aprendizado do Conceito C*) Dados os conjuntos $\mathcal{E}^+$ e $\mathcal{E}^-$ de exemplos positivos e negativos respectivamente do conceito C , encontrar a hipótese $\mathcal{H}$ expressa em uma dada linguagem de descrição $\mathcal{L}_{\mathcal{H}}$, tal que

- todo exemplo positivo $e \in \mathcal{E}^+$ é coberto³ por $\mathcal{H}$
- nenhum exemplo negativo $e \in \mathcal{E}^-$ é coberto por $\mathcal{H}$

Para testar a cobertura, a função

$$\text{cobre}(\mathcal{H}, e)$$

²Entidades não devem ser multiplicadas desnecessariamente.

³Uma descrição de conceito cobre uma descrição de objeto se a descrição de objeto satisfaz à descrição do conceito.

pode ser definida, tal que

$$\begin{aligned} \text{cobre}(\mathcal{H}, e) &= \text{verdade} && \text{se } e \text{ é coberto por } \mathcal{H} \\ \text{cobre}(\mathcal{H}, e) &= \text{falso} && \text{caso contrário} \end{aligned}$$

A função *cobre* redefinida para conjuntos de exemplos é expressa como

$$\text{cobre}(\mathcal{H}, \mathcal{E}) = \{ e \in \mathcal{E} \mid \text{cobre}(\mathcal{H}, e) = \text{verdade} \}$$

É importante notar que a definição da função *cobre* depende das linguagens de descrição de exemplos e de conceitos. A definição de aprendizado indutivo de conceito — Definição 4.6.2 — pressupõe que a hipótese $\mathcal{H}$ cubra todos os exemplos positivos e nenhum negativo. Dependendo de como a hipótese cobre os exemplos positivos e negativos, quatro situações, pictoricamente mostradas na Figura 4.2 [Lavrač 94], podem acontecer

1. $\mathcal{H}$ completa e consistente — cobre todos os exemplos positivos e nenhum negativo
2. $\mathcal{H}$ incompleta e consistente — não cobre todos os exemplos negativos e não cobre nenhum exemplo negativo
3. $\mathcal{H}$ completa e inconsistente — cobre todos os exemplos positivos e cobre alguns exemplos negativos
4. $\mathcal{H}$ incompleta e inconsistente — não cobre todos os exemplos positivos e cobre alguns exemplos negativos

Dado um número de exemplos positivos e negativos, geralmente um número potencialmente infinito de hipóteses que satisfazem os critérios de *completeza* e *consistência* podem ser geradas. Torna-se necessário, pois, restringir o espaço de possíveis hipóteses indutivas usando alguma informação adicional, tal como a fornecida pela teoria do domínio. Na presença da teoria do domínio a tarefa de aprendizado a partir de exemplos pode ser formalmente definida como

Definição 4.6.3 (*Aprendizado do Conceito C na Presença da Teoria do Domínio*)
Dado o conjunto $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ de exemplos positivos e negativos respectivamente do conceito C , a teoria do domínio $\mathcal{K}$, encontrar uma hipótese $\mathcal{H}$, expressa em uma dada linguagem de descrição de conceito $\mathcal{L}_{\mathcal{H}}$, tal que $\mathcal{H}$ é completa e consistente com relação a $\mathcal{K}$ e exemplos $\mathcal{E}$.

Redefinindo a função *cobre* de maneira a considerar a teoria do domínio

$$\begin{aligned} \text{cobre}(\mathcal{K}, \mathcal{H}, e) &= \text{cobre}(\mathcal{K} \cup \mathcal{H}, e) \\ \text{cobre}(\mathcal{K}, \mathcal{H}, \mathcal{E}) &= \text{cobre}(\mathcal{K} \cup \mathcal{H}, \mathcal{E}) \end{aligned}$$

$\mathcal{H}$: completa, consistente

$\mathcal{H}$: incompleta, consistente

$\mathcal{H}$: completa, inconsistente

$\mathcal{H}$: incompleta, inconsistente

Figura 4.2: Completeza e Consistência de uma Hipótese

A hipótese induzida $\mathcal{H}$ deve satisfazer as exigências de ser *completa* e *consistente* com relação ao conjunto de exemplos $\mathcal{E}$. Na presença da teoria do domínio, os conceitos de *completeza* e *consistência* de uma hipótese $\mathcal{H}$ são definidos como

Definição 4.6.4 (*Hipótese Completa*) Uma hipótese $\mathcal{H}$ é completa com relação à teoria do domínio $\mathcal{K}$ e exemplos $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ se todos os exemplos positivos são cobertos, isto é, se $\text{cobre}(\mathcal{K}, \mathcal{H}, \mathcal{E}^+) = \mathcal{E}^+$.

Definição 4.6.5 (*Hipótese Consistente*) Uma hipótese $\mathcal{H}$ é consistente com relação à teoria do domínio $\mathcal{K}$ e exemplos $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$ se nenhum exemplo negativo é coberto, isto é, se $\text{cobre}(\mathcal{K}, \mathcal{H}, \mathcal{E}^-) = \emptyset$.

Em um sistema computacional a tarefa de aprendizado não vai além dos limites definidos pelas linguagens utilizadas. Assim, as linguagens nas quais exemplos, hipóteses e teoria do domínio são expressos caracterizam o escopo e definem os limites da tarefa de aprendizado.

4.7 Relevância das Linguagens de Descrição no Aprendizado Indutivo

Como já mencionado, sistemas de aprendizado de máquina empregam linguagens formais para a descrição de instâncias e conceitos, referenciadas como *linguagem de descrição de instâncias* e *linguagem de descrição de conceitos* respectivamente.

Para a representação dos exemplos de treinamento, os sistemas indutivos proposicionais de aprendizado usam uma linguagem baseada em atributos. Um atributo é uma possível característica relevante do conceito a ser aprendido; exemplos de treinamento são descritos como vetores de pares *atributo-valor* e de uma classe. Em um formalismo baseado em atributos, o conjunto de atributos usado para a descrição dos exemplos é fixo. Cada exemplo pertence a uma, entre várias classes mutuamente exclusivas. Os atributos são frequentemente discretos (nominal, ordinal ou intervalar) e alguns sistemas exigem, também, informação a respeito dos nomes de atributos e do escopo de seus valores. Num sistema de aprendizado que faz uso de uma linguagem baseada em atributos, a tarefa de aprendizado pode ser descrita como

dado um conjunto de exemplos de treinamento expressos como vetores de pares atributo-valor, cujas classes são conhecidas, encontrar uma regra que prediga a classe de um novo exemplo em função de seus atributos e valores

Num sistema de aprendizado proposicional indutivo, as hipóteses induzidas são tipicamente expressas como regras de produção ou árvores de decisão, que podem ser tratadas como variações de linguagens baseadas em atributos. O aprendizado utilizando árvores de decisão pode ser equacionado como uma generalização da classificação de objetos em classe positiva ou negativa.

O Capítulo 5, pg. 33, apresenta e discute sistemas proposicionais indutivos, que adotam como linguagem de descrição de instância uma linguagem baseada em atributos e como linguagem de descrição de hipóteses, árvores de decisão.

Uma tentativa de contornar a expressividade limitada de uma linguagem baseada em atributos é através do uso de *indução construtiva*, que será tratada no Capítulo 6. Indução construtiva é um processo que muda o espaço de descrição, podendo ser definida como qualquer forma de indução que gera novos atributos, não presentes no conjunto inicial de treinamento. Através da indução construtiva o vocabulário de uma linguagem de descrição é ampliado, provocando, com isso, uma mudança nos espaços representacionais de instâncias e conceitos.

Apesar de seu sucesso, métodos de aprendizado que usam linguagens baseadas em atributo (fazendo ou não uso de indução construtiva) são limitados pela linguagem, bem como pela forma restrita e inexpressiva (quando existente) com que a teoria do domínio pode ser incorporada ao processo de aprendizado. Uma linguagem baseada em atributos tem o mesmo poder de expressão que uma linguagem baseada em lógica proposicional. Conseqüentemente, apenas aqueles conceitos expressáveis em lógica proposicional são candidatos a serem aprendidos por um sistema que usa uma linguagem baseada em atributos [Nicoletti 93c].

Esta forte restrição previne a representação de objetos estruturados bem como a representação de relações entre objetos ou entre seus componentes. Aspectos relevantes das instâncias de treinamento, as quais de alguma forma poderiam caracterizar o conceito sendo aprendido, não podem ser representados. Além disso, algumas descrições relacionais, quando expressas usando uma linguagem baseada em atributos, podem ser longas, inexatas e, algumas vezes, completamente incompreensíveis para seres humanos.

Por essas razões, em muitas aplicações práticas, representações baseadas em atributos são insuficientes. Isso inclui domínios onde descrições relacionais necessitam ser expressas, como por exemplo em diagnósticos médicos.

Com o objetivo de contornar as limitações representacionais impostas por uma linguagem baseada em atributos, o aprendizado em representações mais poderosas, tais como algumas variantes da lógica de primeira ordem, têm recebido mais atenção recentemente e é também objeto de pesquisa deste trabalho. Pode se notar um aumento considerável em pesquisas relacionadas à

- extensão do poder representacional de sistemas de aprendizado de máquina através do uso de lógicas de primeira ordem restritas como linguagem de descrição de conceito e
- incorporação efetiva da teoria do domínio para a indução de hipóteses mais compactas

Os exemplos a seguir mostram a diferença no poder de representatividade entre uma linguagem baseada em atributos e uma linguagem baseada em lógica de primeira ordem.

Exemplo 4.7.1 *Sejam A e B dois atributos booleanos. Ao expressar que A e B têm o*

mesmo valor

- usando uma linguagem baseada em atributos, tem-se

$$(A = false) \wedge (B = false) \vee (A = true) \wedge (B = true)$$

- usando uma linguagem de primeira ordem, tem-se

$$A = B$$

Exemplo 4.7.2 No aprendizado do conceito de par, no jogo de pôquer

<i>Naipes₁</i>	<i>Valor₁</i>	<i>Naipes₂</i>	<i>Valor₂</i>	<i>Classe</i>
♦	7	♣	7	<i>par</i>
♠	<i>Q</i>	♥	<i>Q</i>	<i>par</i>
♣	<i>A</i>	♦	<i>J</i>	<i>não_par</i>
...	...	...	...	...

- usando uma linguagem baseada em atributos, tem-se

$$\begin{aligned} \text{par se } & (Valor_1 = 7 \wedge Valor_2 = 7) \\ & \vee \\ & (Valor_1 = Q \wedge Valor_2 = Q) \\ & \vee \\ & \dots \end{aligned}$$

- usando uma linguagem baseada em lógica de primeira ordem, tem-se

$$\text{par if } Valor_1 = Valor_2$$

Expressões em lógica de primeira ordem podem ser consideradas uma maneira natural de descrever relações de uma forma concisa, bem como uma maneira conveniente de descrever informações relacionadas a objetos estruturados.

A área conhecida como programação lógica é uma área tradicional de pesquisa que faz uso de lógica de primeira ordem para a definição de programas. Num contexto amplo, programação lógica pode ser definida como o uso de lógica simbólica para a representação explícita de problemas e de suas bases de conhecimento associadas, bem como o uso de inferência lógica controlada para a solução efetiva desses problemas.

Atualmente programação lógica é entendida em termos mais específicos. A linguagem de representação do problema é um subconjunto da lógica clássica de predicados de primeira ordem — o conjunto das *cláusulas de Horn* — e o mecanismo para a resolução do problema é uma forma particular da inferência clássica de primeira ordem — a *resolução* [Kowalski 87]. Uma das realizações práticas de programação lógica é a linguagem de programação lógica chamada *Prolog*.

Programação Lógica Indutiva — PLI — é uma área de pesquisa relativamente recente, que objetiva integrar as técnicas já disponíveis e estabelecidas em programação lógica, em um ambiente de aprendizado, visando induzir programas lógicos a partir de exemplos, fazendo uso de teoria do domínio.

Em PLI o conhecimento do sistema consiste de exemplos e de teoria do domínio expressos como programas lógicos. A expressividade de programas lógicos e o uso de teoria do domínio promoveram PLI a um poderoso paradigma de aprendizado indutivo. PLI vai além do estabelecido para aprendizado empírico porque faz uso de uma lógica relacional quantificada, articulada ao uso de uma teoria do domínio. PLI vai, igualmente, além do estabelecido para um aprendizado baseado em explicação, uma vez que não pressupõe uma teoria do domínio completa e correta [Muggleton 93].

A maioria das pesquisas em PLI usa a lógica de primeira ordem como linguagem de representação de conhecimento. A maioria delas tem usado subconjuntos restritos da linguagem de programação lógica Prolog, para a representação de conceitos. Uma vantagem no uso da representação em Prolog pelos sistemas de aprendizado está no fato da semântica e da complexidade da linguagem Prolog já terem sido bem estudados e estarem consolidados [Sammur 93]. Um dos problemas básicos que PLI enfrenta é o de ter que reduzir o espaço de hipóteses, de maneira a tornar a tarefa de aprendizado exequível evitando, dessa forma, explosão combinatória durante o processo de busca. Isso é realizado através do controle da linguagem de descrição de instâncias, de hipóteses e da teoria do domínio. Investigação de formas de controle através da restrição das linguagens são conduzidas nos Capítulos 9, pg. 110 e 10, pg. 130.

4.8 Considerações Finais

Neste capítulo foi apresentada uma das possíveis taxionomias de sistemas de aprendizado de máquina existentes, na qual aprendizado indutivo é identificado como o paradigma de aprendizado que realiza aquisição de conceitos através de conjuntos de exemplos.

Aprendizado indutivo a partir de exemplos foi definido e caracterizado em função de: ser incremental ou não, aprender um ou vários conceitos, usar ou não informação da teoria do domínio. A relevância das linguagens de descrição utilizadas para expressão das instâncias, hipóteses e teoria do domínio foi discutida e aprendizado indutivo também caracterizado em função do poder de representatividade dessas linguagens.

Capítulo 5

Aprendizado de Árvores de Decisão

5.1 Considerações Iniciais

Muitos dos sistemas e programas que realizam AM implementam técnicas — algumas vezes complementares, outras vezes alternativas — para a solução de problemas isomorfos. Na maioria das vezes isso não fica evidente devido ao uso, pelos vários sistemas, de formalismos e terminologias diferentes.

Uma tendência que pode ser observada na área de AM é o investimento no chamado aprendizado indutivo por exemplos, tratado na Seção 4.3, pg. 23, onde os exemplos são vetores de pares *atributo-valor* e uma classe associada. Essa tendência é evidenciada pela publicação constante de novos algoritmos que, via de regra, são refinamentos e/ou adaptações e melhoramentos de dois algoritmos básicos: o ID3 [Quinlan-86], que utiliza árvores de decisão para a expressão de hipóteses induzidas e o AQ [Michalski 83], que expressa as hipóteses induzidas como regras de produção.

Sob o título de família TDIDT¹ são agrupados os algoritmos e sistemas mais representativos que usam como algoritmo básico de aprendizado o ID3. Aqueles que usam como algoritmo básico de aprendizado o AQ, via de regra implementam uma busca *beam*², fazem parte da família AQ e não são objeto de pesquisa deste trabalho.

Os algoritmos e sistemas pertencentes à família TDIDT caracterizam-se por representar suas hipóteses como árvores de decisão. Esta forma de representação de conhecimento, relativamente simples, não tem o poder expressivo de redes semânticas ou de outras representações de primeira ordem. Apesar da simplicidade, entretanto, consegue-se expressar problemas complexos usando árvores de decisão. Como consequência da simplicidade da representação de conhecimento utilizada, esses algoritmos são signi-

¹Top Down Induction of Decision Trees.

²Numa busca *hill-climbing* (ver Apêndice A) o algoritmo guarda a *melhor* cláusula e a substitui pelo seu melhor refinamento a cada passo da especialização, até o critério de parada ser satisfeito. Numa busca *beam*, o algoritmo guarda várias das *melhores* cláusulas, ao invés de apenas uma.

ficativamente menos complexos que aqueles que fazem uso de uma representação de conhecimento mais elaborada.

Este capítulo caracteriza a família TDIDT, apresentando e discutindo os aspectos e particularidades mais relevantes que deram origem à diversidade de algoritmos e sistemas existentes nesta família. Os mecanismos de *janela* e de *poda* adotados por alguns dos sistemas da família TDIDT são apresentados e discutidos. Por fim, o ambiente computacional para esta família de algoritmos, por nós projetado e implementado, é mostrado em detalhes.

5.2 Árvores de Decisão como Linguagem de Descrição de Hipóteses

Pode-se definir formalmente uma árvore de decisão como sendo

Definição 5.2.1 (*Árvore de Decisão — AD*) [Utgoff 89]

1. um nó folha — ou nó resposta — que contém um nome de classe, ou
2. um nó folha — ou nó de decisão — que contém um teste de atributo que, para cada um dos possíveis valores de atributo, tem um ramo para uma outra árvore de decisão

Na família TDIDT as árvores de decisão são construídas de uma maneira top-down, ou seja, a partir da raiz, descendo até as folhas. O próprio nome da família TDIDT enfatiza esta característica. A indução top-down realiza uma busca *hill-climbing* (ver Apêndice A, pg. 169) guiada por uma heurística estatística e sem *backtracking*.

Os exemplos classificados, a partir dos quais é gerada a árvore de decisão, são conhecidos somente através de um conjunto de atributos e de seus respectivos valores. Costuma-se, geralmente, considerar a classe do exemplo como um atributo. As árvores de decisão, por sua vez, são construídas usando estes mesmos atributos. No exemplo a seguir [Quinlan 87] para classificar pacientes com hipotireoidismo, o atributo de classe pode assumir os valores

$\{\text{hipo-primário}, \text{hipo-compensado}, \text{negativo}\}$

Para a descrição dos exemplos, alguns dos atributos utilizados neste caso são contínuos, tais como medições de *TSH*, *FTI* e *TT4*, enquanto que outros são discretos, como *sexo* com valores $\{f, m\}$, ou *cirurgia-tireoides* com valores $\{f, v\}$.

Cada nó da árvore de decisão corresponde a um atributo e as folhas, às classes. Para a construção de um nó na árvore, é necessário escolher um determinado atributo e agrupar os exemplos associados ao nó, usando como critério os valores daquele atributo escolhido. No exemplo da Figura 5.1 (pictoricamente mostrado também na Figura 5.2), a primeira escolha — a raiz da árvore — recaiu sobre o atributo *TSH*. Desta forma,

todas as observações de hipotireoidismo do conjunto de treinamento foram divididas em dois grupos, com base no valor do TSH ser menor ou maior que 6.05. Se um ramo da árvore contém exemplos de uma única classe, pode-se dizer que ele está ordenado e possui entropia zero: sua desorganização é nula. A entropia do ramo $TSH < 6.05$ é zero uma vez que a ele estão associados somente exemplos da classe negativa. O outro ramo, por sua vez, tem a ele associados exemplos de diferentes classes tendo, portanto, entropia maior que zero.

A Figura 5.1 mostra uma árvore para classificação de hipotireoidismo, gerada utilizando o algoritmo ID3. Alguns dos exemplos de treinamento usados para a construção da árvore estão na Tabela 5.1.

<i>sexo</i>	<i>cirurgia_tireoides</i>	<i>TSH</i>	<i>FTI</i>	<i>TT4</i>	<i>tiroxina</i>	<i>classe</i>
f	f	5.8	66.2	156.2	f	negativo
f	f	7.3	70.2	152.9	v	negativo
m	v	8.8	60.2	148.4	f	hipo_primario
f	f	6.9	65.4	132.7	f	hipo_compensado
m	f	5.4	65.2	150.9	f	negativo
...	...	...	...	...	...	...

Tabela 5.1: Conjunto de Treinamento

TSH < 6.05 : negativo

TSH > 6.05 :

FTI < 64.5 : hipo-primario

FTI > 64.5 :

tiroxina = v : negativo

tiroxina = f :

cirurgia-tireoides = v : negativo

cirurgia-tireoides = f :

TT4 < 150.5 : hipo-compensado

TT4 > 150.5 : negativo

Figura 5.1: Árvore de Classificação de Hipotireoidismo

É importante notar que muito embora o atributo *sexo* faça parte dos exemplos de treinamento, tal atributo foi descartado pela função de avaliação durante a construção da árvore. Cada um dos ramos da árvore representa uma regra. O processo de “tradução” de uma árvore de decisão em regras é trivial. Algumas das regras são descritas na Tabela 5.2.

5.3 Algoritmo Geral dos Sistemas da Família TDIDT

A construção da árvore de decisão é um processo iterativo, onde cada nó da árvore corresponde a um atributo. A princípio, as possíveis ramificações em cada nó correspondem aos possíveis valores que o atributo em questão pode ter. A cada um dos ramos

Figura 5.2: Estrutura de Classificação Pictoricamente Representada como Árvore

(valor) que parte do nó (atributo), estão associados todos os exemplos para os quais o atributo tem aquele valor.

O objetivo de um sistema de aprendizado é gerar a menor árvore de decisão que classifique corretamente todos os exemplos do conjunto de treinamento. Para isso, é necessário escolher, a cada passo, o atributo mais relevante, isto é, o atributo que melhor particiona os exemplos segundo o valor da classe. A escolha do atributo mais promissor é realizada utilizando uma função chamada de *função de avaliação*. Existem diversas funções de avaliação disponíveis. Elas são definidas, geralmente, com base na teoria da informação ou em medidas estatísticas [Mingers 89].

O processo de escolha de um atributo — utilizando uma determinada função de avali-

- 1 Se $TSH < 6.05$
então negativo.
- 2 Se $TSH > 6.05$ e
FTI < 64.5
então hipo-primário.
- ...
- 6 Se $TSH > 6.05$ e
FTI > 64.5 e
tiroxina = falso e
cirurgia-tireoides = falso e
TT4 > 105.5
então negativo.

Tabela 5.2: Regras de Decisão para Classificação de Hipotireoidismo

ação — e de partição dos exemplos segundo os valores desse atributo é repetido para cada nó da árvore, enquanto a *condição de parada* não for verificada no nó. A condição de parada, geralmente, é a de que todos os exemplos associados a um nó — ou a maioria deles — pertençam à mesma classe. Se esta condição for satisfeita, o nó torna-se uma folha rotulada com o valor da classe.

O algoritmo geral dos sistemas da família TDIDT começa com uma árvore de decisão vazia, que é construída gradualmente até que classifique corretamente todos os exemplos do conjunto de treinamento. O procedimento geral é o seguinte

Dados um conjunto de exemplos de aprendizado E
 uma condição de parada $t(E)$
 uma função de avaliação $aval(E, atributo)$

Se todos os exemplos em E satisfazem a condição de parada $t(E)$

então retorne o valor da classe

caso contrário

1. Para cada atributo a_i determine o valor da função $aval(E, a_i)$. Seja a_j o atributo que possui o melhor valor³ de $aval(E, a_i)$
2. Se a_j possui valores $a_{j1}, a_{j2}, \dots, a_{jk}$ crie o seguinte nó da árvore

3. Particione os exemplos do conjunto E nos subconjuntos $E_1, E_2, \dots, E_k$ segundo os valores de a_j na árvore de decisão
4. Aplique o algoritmo recursivamente para cada um dos subconjuntos E_i

O critério de parada t pode ser definido tanto para

- construir a árvore de decisão, que classifica exatamente todos os elementos do conjunto de aprendizado em domínios completos — sistema ID3 [Quinlan 79], quanto para
- decidir pela não expansão da árvore quando a evidência for insuficiente nos exemplos fornecidos. Este mecanismo é denominado pré-poda da árvore de decisão — utilizado pelos sistemas C4 [Quinlan-86] e Assistant-86 [Cestnik 87]. A função de avaliação $aval$ pode ser definida de várias formas diferentes

³Atributo mais relevante para realizar a partição, segundo a função de avaliação.

As diversas condições de parada, funções de avaliação e o fato de serem ou não incrementais, contribuem para a caracterização dos vários algoritmos desta família.

5.4 Critérios para Avaliação de Árvores de Decisão

Uma vez que a família TDIDT adota árvore de decisão para a expressão de suas hipóteses, é importante que os principais aspectos que caracterizam uma árvore de decisão sejam evidenciados e definidos de maneira a subsidiarem o estabelecimento de critérios para a avaliação de uma árvore. Entre os principais aspectos destacam-se

- *tamanho* — medida associada ao número de nós/folhas da árvore. Geralmente o número de folhas é selecionado como medida de tamanho, uma vez que corresponde ao número de regras distintas que podem ser extraídas da árvore.
- *precisão* — medida associada à capacidade de predição de uma árvore de decisão, quando utilizada na classificação de um conjunto de teste independente⁴. Esta medida, de certa forma, não é muito refinada, uma vez que não reflete a precisão na predição de diferentes classes contidas no conjunto de teste. Entre as possíveis opções para medida de precisão, duas delas são mais freqüentemente utilizadas
 - *média majoritária* — porcentagem de classificações corretas — razão entre o número de exemplos classificados corretamente e o número de exemplos de teste
 - *medida probabilística* — razão entre a soma das probabilidades das classes dos exemplos preditas pela regra de classificação, e o número de exemplos do conjunto de teste
- *facilidade de compreensão* — espera-se que em Sistemas Baseados em Conhecimento o conhecimento seja representado explicitamente, de maneira que tanto o especialista quanto o usuário (até um certo nível), possam entendê-lo facilmente. A facilidade de compreensão é uma vantagem que árvores de decisão têm sobre, por exemplo, técnicas estatísticas que realizam a mesma função, tal como análise discriminante. Isto se deve ao fato das árvores de decisão serem facilmente “traduzidas” em regras de decisão e à maior facilidade de compreensão de regras de decisão. É importante lembrar, entretanto, que facilidade de compreensão é uma característica difícil de ser quantificada, principalmente por ser relativamente subjetiva.

É consenso que quanto maior a profundidade da árvore, menos compreensíveis elas se tornam. Este fato é particularmente evidenciado em árvores cujos atributos foram binarizados (ver Seção 5.5), uma vez que elas tendem a se tornar muito profundas testando, freqüentemente, o mesmo atributo muitas vezes em um mesmo ramo. Além disso, quando se binariza atributos multivalorados, não existe garantia que os agrupamentos de valores de atributo resultantes terão qualquer

⁴Conjunto de exemplos que não o conjunto de treinamento.

significado para o especialista. Apesar da compreensibilidade ser uma característica difícil de ser medida ou quantificada, existem fortes indícios que árvores com maior profundidade e, particularmente, as que têm os atributos binarizados, são as menos compreensíveis pelo ser humano.

5.4.1 Tipos de Atributos Utilizados em Linguagens Proposicionais

Atributos se caracterizam em termos do número de possíveis valores que podem assumir bem como em termos da natureza desses valores. Segue uma possível classificação de tipos de atributos

- *intervalo* — atributo que tem valor inteiro ou real, tal como dinheiro, idade, peso, etc.
- *ordinal* — atributo cujo conjunto de valores pode ser ordenado e/ou agrupado em categorias, tal como o nível obtido em um exame (A,B,C,D ou E)
- *nominal* — atributo que assume valores que não têm uma ordem inerente estabelecida, tal como tipo de carro, sexo etc.
- *estruturado* — atributo que pode misturar os tipos acima e que são difíceis de serem utilizados [Bundy 85]

5.4.2 Critérios para a Avaliação dos Atributos

Durante a construção da árvore de decisão, existem dois conjuntos associados a cada nó da árvore: um de exemplos e outro de atributos. Uma maneira de determinar o melhor atributo dentre os disponíveis, de maneira a prosseguir com o processo de construção, é analisar quão bem cada um deles separa os exemplos nas várias classes, e selecionar aquele que promove uma melhor discriminação entre elas. Entre as várias funções de avaliação utilizadas para descobrir qual o melhor, dentre os atributos disponíveis, estão

1. medida de informação
2. estatística χ^2 (Chi-quadrado)
3. estatística G (baseada em medida de informação) [Sokal 81]
4. índice de diversidade GINI [Breiman 84]
5. medida Gain-Ratio (variante da medida de informação)

Os experimentos conduzidos por Mingers [Mingers 89] utilizando quatro domínios de conhecimento: BABS, *Cancer*, *Iris*, *Digits* mostraram que o uso de qualquer dos critérios de seleção de atributos reduz aproximadamente pela metade o tamanho da árvore gerada, quando comparada com árvores geradas utilizando uma estratégia de seleção

randômica de atributos. Na comparação de desempenho entre as várias funções de avaliação, utilizando aqueles mesmos domínios, evidenciou-se uma diferença significativa: o Gain-Ratio produz as menores árvores e o Chi-quadrado as maiores.

Com relação à precisão na classificação de novas instâncias, os experimentos mostraram que ela não melhora significativamente com o uso de uma função de avaliação de atributos: a seleção de atributos feita de maneira inteiramente randômica produz árvores que são tão precisas quanto aquelas produzidas usando uma função de avaliação de atributos. É importante notar, entretanto, que o uso de uma função de avaliação influencia significativamente o tamanho das árvores não podadas (ver Seção 5.7).

5.5 Tratamento do Conjunto de Treinamento

Considerando a própria natureza de processos de amostragem e coleta de dados, pode haver, em muitos problemas reais, um grau de incerteza presente nos dados, consequência, por exemplo, de uma medida feita erradamente ou, então, da presença de dados incompletos e/ou inconsistentes. Sistemas de aprendizado indutivo devem viabilizar maneiras de tratamento de tais tipos de dados. Seguem algumas abordagens adotadas por sistemas de aprendizado para tratamento do conjunto de treinamento.

Limpeza Inicial do Conjunto de Treinamento Alguns sistemas, com o objetivo de obter um conjunto de treinamento com um número menor de inconsistências e/ou ruídos, fazem um pré-processamento deste conjunto, de maneira a remover certos exemplos⁵ e, assim, obter um conjunto mais representativo. Durante o pré-processamento tais exemplos são eliminados do conjunto de treinamento. A identificação dos exemplos a serem eliminados é feita, geralmente, através de métodos estatísticos.

Binarização de Atributos Atributos multivalorados⁶ tendem a discriminar melhor entre as classes, simplesmente porque têm mais valores possíveis (o que nem sempre é conveniente). Como consequência, esses atributos são favorecidos e tendem sempre a ser incorporados na árvore de decisão.

Uma possível solução para evitar o favorecimento [Breiman 84], adapta o algoritmo ID3 de maneira que tal algoritmo trate todos atributos como se fossem atributos binários. A binarização é feita através do agrupamento dos possíveis valores de cada atributo em dois conjuntos, e do tratamento de cada conjunto como se fosse um único valor de atributo. Para cada atributo, todas as possíveis *binarizações* de valores são avaliadas. Se o atributo for ordinal, apenas valores contíguos podem ser agrupados; entretanto, se o atributo for nominal, qualquer combinação é permitida. A binarização de atributos tem como efeito colateral a construção de árvores binárias: os dois possíveis ramos

⁵Por exemplo, exemplos contraditórios — idênticos valores de atributos mas associados a diferentes classes.

⁶Atributos que têm mais do que dois possíveis valores.

que saem de um nó apontam para os dois possíveis conjuntos de valores do atributo representado pelo nó.

Atributos Numéricos Via de regra, um atributo numérico é tratado através da divisão de seu domínio de valores em dois intervalos — binarização numérica — divisão essa feita de maneira exaustiva, a cada passo da iteração. Do conjunto de possíveis intervalos binários, é escolhido o melhor, geralmente usando como critério o valor mais baixo de entropia.

Essa abordagem, mesmo que largamente utilizada por sistemas indutivos de aprendizado, tem alguns inconvenientes, tais como os apontados em [Hooke 89]. Uma única divisão em dois intervalos pode, eventualmente, falhar no reconhecimento de determinadas distribuições de classes, e o processo todo pode terminar construindo um conjunto de regras pouco representativas.

Isso tipicamente acontece quando os valores de um certo atributo numérico, que caracterizam uma classe, estão concentrados entre determinados valores do mesmo atributo, que caracterizam uma outra classe. Neste caso, o processo de construção da divisão binária de intervalos pode falhar na identificação da separação das classes, com relação ao atributo numérico em questão.

Essa situação é mostrada no exemplo que se segue. Nele, um determinado atributo numérico, identificado por A , pode assumir valores numéricos inteiros entre 1 e 20 e as instâncias de treinamento pertencem a duas possíveis classes, identificadas por $\circ$ e $\diamond$. Nas Figuras 5.3 e 5.4, as freqüências — freq — fornecem o número de exemplos pertencentes a cada uma das classes, para cada um dos possíveis valores do atributo A (20 possíveis valores). Por exemplo, quando $A = 1$, existem 5 exemplos, todos eles pertencentes à classe $\circ$. Quando $A = 12$, existem 2 exemplos pertencentes à classe $\circ$ e 6 pertencentes à classe $\diamond$.

Na Figura 5.3 é considerada uma única divisão no domínio dos valores do atributo A . O valor da entropia associada à essa divisão é de 0.490, calculado como segue — ver Apêndice B.

$$\begin{aligned} H(C|A \leq 10) &= -p(\circ|A \leq 10) \log p(\circ|A \leq 10) - p(\diamond|A \leq 10) \log p(\diamond|A \leq 10) \\ &= -72/72 \log(72/72) \\ &= 0 \end{aligned}$$

$$\begin{aligned} H(C|A > 10) &= -p(\circ|A > 10) \log p(\circ|A > 10) - p(\diamond|A > 10) \log p(\diamond|A > 10) \\ &= -28/78 \log(28/78) - 50/78 \log(50/78) \\ &= 0.94182854 \end{aligned}$$

$$\begin{aligned} H(C|A) &= 72/150 \times 0 + 78/150 \times 0.94182854 \\ &= 0.48975084 \approx 0.490 \end{aligned}$$

A Figura 5.4 mostra o mesmo conjunto de exemplos, mas submetidos a duas divisões no domínio de valores da variável A , com a entropia resultante de 0.261, calculada como segue.

$$H(C|A \leq 10) = 0$$

Figura 5.3: Uma Única Divisão do Intervalo - Entropia = 0.490

$$\begin{aligned}
 H(C|10 < A \leq 17) &= -p(o|10 < A \leq 17) \log p(o|10 < A \leq 17) - \\
 &\quad p(d|10 < A \leq 17) \log p(d|10 < A \leq 17) \\
 &= -8/57 \log(8/57) - 49/57 \log(49/57) \\
 &= 0.58515699
 \end{aligned}$$

$$\begin{aligned}
 H(C|17 < A \leq 20) &= -p(o|17 < A \leq 20) \log p(o|17 < A \leq 20) - \\
 &\quad p(d|17 < A \leq 20) \log p(d|17 < A \leq 20) \\
 &= -20/21 \log(20/21) - 1/21 \log(1/21) \\
 &= 0.27619542
 \end{aligned}$$

$$\begin{aligned}
 H(C|A) &= 72/150 \times 0 + 57/150 \times 0.58515699 + 21/150 \times 0.27619542 \\
 &= 0.26102701 \approx 0.261
 \end{aligned}$$

A divisão extra mostrada na Figura 5.4 produz uma redução relativa de aproximadamente de 50% na entropia. Isto mostra a inadequação de uma única divisão. No caso particular deste exemplo, a adoção de duas divisões é muito mais apropriada. Com relação a atributos numéricos, é importante observar que

- não existem quase estudos que abordam o problema
- a maioria dos sistemas indutivos existentes adota binarização
- o estudo sobre multidivisões [Hooke 89] não é conclusivo, uma vez não deixa estabelecido
 - quando efetivamente usar e
 - quais as maneiras de realizar multidivisões, assim como quantas devem ser e como escolher a melhor

Atributos com Valor Desconhecido Em domínios reais acontece, às vezes, de existirem exemplos no conjunto de treinamento para os quais valores de determinados

Figura 5.4: Duas Divisões do Intervalo - Entropia = 0.261

atributos estão ausentes. Esta situação geralmente é tratada

- pela introdução de um novo valor de atributo, por exemplo, *valor_desconhecido*, para todos aqueles atributos onde tal ausência foi verificada; a introdução deste novo valor de atributo provoca um aumento no tamanho da árvore de decisão
- pela atribuição do valor mais provável e
- pela atribuição de todos os possíveis valores, cada um com uma probabilidade condicional associada (obtida pela frequência relativa dos exemplos de uma dada classe no conjunto de treinamento)

Domínio Incompleto Se o domínio for incompleto, i.e., o conjunto de atributos não for suficiente para uma classificação exata de todos os exemplos, a árvore de decisão obtida não é confiável.

Atributos com Valor Irrelevante (“don’t care”) Quando um atributo tem valor irrelevante associado, via de regra o exemplo de treinamento é expandido em tantos exemplos quantos forem os possíveis valores daquele atributo.

5.6 O Mecanismo de Janela

Como visto na Seção 5.3, o algoritmo geral dos sistemas da família TDIDT começa com um conjunto de exemplos de aprendizado *E* e uma árvore de decisão vazia, que vai sendo gradualmente construída, até que classifique corretamente os exemplos de *E*.

Quando o número de exemplos de treinamento em E for muito grande, os algoritmos não incrementais da família TDIDT, por exigirem que todos os exemplos do conjunto de treinamento estejam simultaneamente na memória, provocam um problema se houver limitações no uso de memória. Para contornar esse problema é usada a estratégia iterativa de janelas, que procura evitar o aprendizado a partir de **todos** os exemplos [Wirth 88, Nicoletti 92]. Esta estratégia, mostrada na Figura 5.5, consiste em

Escolher aleatoriamente um subconjunto do conjunto original de exemplos, chamado de janela — constrói-se então uma árvore de decisão utilizando os exemplos da janela. A árvore assim construída é então usada para classificar os exemplos do conjunto original que não participaram da janela. Aqueles não classificados são inseridos na janela, e com essa janela ampliada constrói-se uma nova árvore de decisão. O processo se repete até que a árvore construída utilizando a janela classifique todos os exemplos

Figura 5.5: Mecanismo de Janela

Se a janela não ficar muito grande, com certeza a construção da árvore irá requerer, significativamente, menos memória. A justificativa para o uso de janelas baseia-se na idéia que conjuntos de treinamento muito grandes, via de regra, contêm muitos elementos redundantes e que, na realidade, apenas um conjunto bem menor de exemplos seria necessário para o aprendizado do conceito em questão. Se, entretanto, não for esse o caso, a janela pode se expandir tanto que, eventualmente, irá coincidir com o próprio conjunto original de exemplos de treinamento. Esse é o pior caso do uso do

mecanismo de janela.

Uma outra dificuldade é que, mesmo se o conjunto de exemplos necessários para a expressão do conceito for pequeno, não existe garantia alguma que tais exemplos irão ser encontrados rapidamente. A efetividade do uso de janelas pode também ser comprometida na presença de dados com ruídos.

A avaliação do uso de janelas em oito domínios: *Sono*, *Contratos*, *LED*, *Xadrez*, *Black-jack*, *Prob-Dist*, *Jogo da Velha* e *Cogumelos* [Wirth 88], desaconselha o não uso de janelas para a maioria dos domínios. O uso de janelas, entretanto, deve ser considerado em situações onde as limitações físicas de memória constituem um sério problema. Seu uso é recomendado a domínios sem ruídos, onde o conjunto de exemplos de treinamento é bem extenso, mas com alta incidência de redundância. Os custos e benefícios do processo de janela podem ser avaliados com relação a

- desempenho do processo indutivo
 - número de iterações requeridas do programa de aprendizado
 - percentual do conjunto de treinamento usado na iteração final
 - tempo de CPU
- qualidade da árvore obtida
 - precisão de predição
 - complexidade da árvore, medida em número de nós

Nos experimentos analisados em [Wirth 88] a janela inicialmente contém 10% dos exemplos do conjunto de treinamento. O número de exemplos adicionados a cada iteração é determinado como o menor entre: o número de erros de classificação nos exemplos restantes e o número que representa 20% da janela inicial.

5.7 O Mecanismo de Poda

Em muitos problemas reais pode haver um grau de incerteza presente nos dados, bem como exemplos observados que não são representativos do conceito a ser aprendido. A incerteza pode ter duas origens distintas. A primeira delas é quando a medida é feita erradamente. Por uma série de razões o valor de um atributo ou classe pode ter sido medido incorretamente, ou mesmo, não ter sido medido. Isto acontece devido a uma possível medida, gravação ou transcrição feita incorretamente — esse tipo de incerteza é conhecido como *ruído*. A segunda origem da incerteza é devida à ocorrência de fatores externos, dos quais não se tem registro, mas que afetam os resultados. Nesta situação, a classe de um exemplo não pode ser totalmente determinada a partir de seus valores de atributos e esse tipo de incerteza é conhecido como *variação residual* [Mingers 89a].

Quando os algoritmos da família TDIDT classificam tais dados, a árvore resultante tende a ser bem grande. Muitos dos ramos desta árvore, entretanto, refletem apenas

ocorrências acidentais nos dados em questão ao invés de, efetivamente, representarem relações existentes entre eles. Conseqüentemente, tais ramos dificilmente serão representativos na classificação de novos exemplos.

Um método de poda identifica os ramos menos confiáveis em uma árvore e os remove. Em domínios onde o grau de incerteza é alto, isso freqüentemente resulta na remoção de todos os níveis da árvore, com exceção dos dois ou três primeiros.

É importante lembrar que a poda de uma árvore irá aumentar o número de erros de classificação dos exemplos do conjunto de treinamento, pois a árvore foi construída usando somente esses exemplos. Em contrapartida, porém, deve diminuir o índice de ocorrência de erros na classificação de novos exemplos. Há dois tipos de poda em árvore de decisão, conhecidos como

- *pré-poda*, realizada durante o processo de construção da árvore
- *pós-poda*, realizada após a árvore de decisão ter sido construída

A pré-poda é usada para parar a ramificação de um nó da árvore, durante a sua construção, quando a classificação correspondente deixa de ser confiável. A pós-poda é utilizada após a construção da árvore, quando o erro de classificação de novos exemplos é calculado e, em função desse erro, decide-se pela poda de algumas subárvores.

A pós-poda na árvore de decisão, em geral, produz resultados mais satisfatórios que a pré-poda, uma vez que, com a pós-poda, tanto a exatidão quanto a complexidade das árvores de decisão tendem a melhorar [Mingers 89a]. É importante, pois, analisar a variação da exatidão e da complexidade de árvores de decisão — geradas com exemplos de diversos domínios — após a aplicação do mecanismo de pós-poda. Na literatura podem ser encontrados diversos métodos de poda. Em [Mingers 89a], por exemplo, são analisados cinco dos mais conhecidos métodos. Dentre eles

- *Valor Crítico* [Mingers 87], *Custo-Complexidade* [Breiman 84] e *Redução do Erro* [Quinlan 87] têm um bom desempenho, enquanto que
- *Mínimo Erro* [Niblett 88] e *Erro Pessimista* [Quinlan 87] podem eventualmente dar problemas

5.8 Método de Poda da Redução do Erro e seu Variante

A aplicação do método de poda da Redução do Erro pressupõe a existência de um conjunto separado de exemplos de testes, tal que cada um dos exemplos deve ser classificado pela árvore original — construída a partir do conjunto de exemplos de treinamento. O método produz uma série de árvores podadas, usando o conjunto de teste diretamente, ao invés de usá-lo apenas na seleção da melhor árvore. A árvore de decisão obtida é aquela que produz o menor número de exemplos não classificados para o conjunto de teste. O método de poda da Redução do Erro é descrito como

Para toda subárvore não folha S de T , são contados o número de exemplos não classificados, d_s , bem como o número de exemplos d_{ps} que não seriam classificados, se a subárvore S em questão fosse substituída por sua melhor folha. Se com a nova árvore o número de erros de classificação obtido (g) é igual ou menor ao obtido com a árvore anterior, ou seja, $g_s = d_s - d_{ps} \geq 0$, e se S não contiver qualquer subárvore com a mesma propriedade, S é definitivamente substituída pela sua melhor folha e o processo prossegue com essa nova árvore. O processo finaliza quando não for mais possível diminuir o valor do erro, ou seja, qualquer substituição de subárvore por folha aumenta o erro de classificação.

O fato de $g_s > 0$ significa que um número maior de exemplos será classificado pela árvore de decisão se a subárvore S for podada, ou seja, substituída por sua melhor folha. A poda, quando $g_s = 0$ (ganho nulo) não incrementa a precisão da árvore, mas favorece a diminuição do tamanho da árvore.

Considere, por exemplo, uma subárvore de uma árvore de classificação (árvore construída a partir de 200 exemplos, com 3 classes), mostrada na Figura 5.6.

Figura 5.6: Subárvore de uma Árvore de Classificação

O primeiro passo num processo de poda que utiliza conjunto de teste é construir uma árvore chamada *árvore estatística*. A árvore estatística é exatamente a árvore original obtida a partir do conjunto de treinamento, com a informação adicional em cada nó, do número total de exemplos/por classe que o acessaram durante o processo de classificação do conjunto de teste.

A subárvore mostrada na Figura 5.6 tem por raiz o nó 26 e nele chegaram, para a classificação do conjunto de teste, 20 exemplos da classe 1, 10 da classe 2 e 5 da classe 3. A subárvore em questão tem 4 folhas. Considerando a subárvore cuja raiz é o nó 27, verifica-se facilmente que de acordo com os critérios estabelecidos pelo método da Redução do Erro, esta subárvore deve ser podada. Se a subárvore for mantida, há 5 erros de classificação de exemplos do conjunto de teste, ou seja

- no nó 30, 4 exemplos de classe 2 foram classificados erradamente como classe 1
- no nó 31, 1 exemplo de classe 1 foi classificado erradamente como classe 2

Se a subárvore em questão for podada e substituída por sua melhor folha (no caso, a folha de classe 1, uma vez que esta é a que mais classifica) continuarão ainda a existir 5 erros de classificação, mas a árvore obtida é menor. O nó 26 e seus descendentes são substituídos pela folha classe 1, restando os números de exemplos/classe que estavam associados àquele nó. Com essa nova árvore, o processo continua até que o critério de parada seja satisfeito.

O método da Redução do Erro gera um conjunto de árvores e termina com a obtenção da árvore podada que melhor classifica o conjunto de testes. Pode ser que aconteça a situação de existirem várias subárvores com o mesmo valor de g . Quinlan não especifica qual subárvore escolher nesta situação — por exemplo, se a menor ou a maior. Mingers em [Mingers 89a] comenta, entretanto, que experimentos mostram que a escolha de qual dessas subárvores interfere muito pouco na precisão de classificação; escolhe-se, portanto, a maior, na certeza que essa escolha irá reduzir o número de iterações necessárias para podar completamente a árvore.

Este método possui duas desvantagens: a necessidade de um conjunto de teste separado e a poda indesejável de partes da árvore original, correspondentes a casos especiais mais raros, eventualmente não presentes no conjunto de teste utilizado para realizar a poda.

O método de Redução do Erro e seu Variante [Castiñeira 90a] diferem na ordem de visita às subárvores. O primeiro começa a análise das subárvores a partir dos nós terminais, e o segundo, a partir dos filhos da raiz. Em [Nicoletti 92] são apresentadas e discutidas as implementações Prolog de ambos os métodos e em [Nicoletti 93b] o desempenho de ambos é comparado, quando utilizados no aprendizado construtivo (ver Capítulo 6) de funções booleanas.

5.9 Família TDIDT

São apresentados, a seguir, os principais sistemas/algoritmos que fazem parte da família TDIDT. Na literatura pode ser encontrado um número bem maior de sistemas/algoritmos que poderiam participar da família TDIDT. O objetivo, aqui, é o de apresentar apenas os mais significativos, que estão discriminados na Figura 5.7

CLS — O Concept Learning System [Hunt 66] foi o primeiro software construído com o objetivo de generalizar automaticamente regras a partir de exemplos. As generalizações produzidas pelo CSL estão na forma de árvores de decisão.

ID3, ID3' — O algoritmo Iterative Dichotomizer 3 [Quinlan 79, Quinlan 83] constrói uma árvore de decisão que discrimina exemplos positivos de exemplos negativos de um determinado conceito, onde cada exemplo é descrito por um conjunto de atributos e

Figura 5.7: Família TDIDT

seus respectivos valores. O ID3 usa entropia como função de avaliação. O algoritmo ID3' [Schlimmer 86] é uma tentativa de tornar o ID3 incremental. O ID3' constrói uma nova árvore, a cada nova observação de instância de treinamento, o que se constitui num procedimento tremendamente ineficiente.

ACLS, Expert-Ease, Ex-Tran, RuleMaster, SuperExpert — Esses cinco sistemas, disponíveis comercialmente, são shells indutivas para Sistemas Especialistas. Foram desenvolvidos e são comercializados pela Intelligent Terminals Ltd. (ITL), Glasgow, Escócia. Todos eles implementam o ID3 — variam entre si no tratamento dado ao conjunto de treinamento (ver Seção 5.5) bem como no tratamento dado à expressão final da hipótese induzida.

Assistant, Assistant-86 — O Assistant-86 [Cestnik 87] é uma implementação do Assistant [Kononenko 85] para PC/DOS. Todos os melhoramentos que o Assistant propôs para o ID3 original, foram implementados no Assistant-86. Os problemas tratados pelo Assistant-86, bem como as respectivas soluções estão mostradas na Tabela 5.3.

problema	tratamento
atributo com valor desconhecido	probabilidade
conjunto de atributos insuficiente	classificação Bayes
ruídos nos dados	pré e pós poda
atributos multivalorados	binarização
atributos com valores contínuos	binarização

Tabela 5.3: Problemas Tratados pelo Assistant-86

C4 — O C4 [Quinlan-86] é um descendente do ID3 implementado em C para um ambiente Unix. Caracteriza-se por

- permitir dados discretos e reais como valores de atributos
- tratar atributos que tenham valor desconhecido
- ter classes mutuamente exclusivas — cada instância pertence à exatamente uma classe

- usar como função de avaliação de atributo o *Gain Ratio* [Quinlan-86, Quinlan 86a]
- induzir a árvore de decisão usando a estratégia de *janela*
- utilizar um critério de parada que permite a interrupção do processo iterativo de geração da árvore, quando não forem detectadas melhorias nas últimas iterações
- podar as árvores de decisão obtidas utilizando apenas o conjunto de treinamento. Essa poda, chamada de *Erro Pessimista*, tem duas vantagens: é rápida e não utiliza conjunto de teste. Como comentado na Seção 5.7, pg. 45, entretanto, ela não produz resultados tão bons quanto esperados

$ID4$, $\widehat{ID4}$, $ID5$, $\widehat{ID5}$, $ID5R$, $\widehat{ID5R}$ — O $ID4$ [Schlimmer 86] foi uma primeira tentativa de tornar o algoritmo $ID3$ incremental. Apesar do $ID4$ oferecer uma alternativa incremental a algoritmos do tipo $ID3$, seu uso pode ser extremamente desvantajoso. Sua principal desvantagem está na eliminação de toda, ou de parte da árvore de decisão, sempre que durante sua construção um atributo de teste for substituído por um atributo considerado mais informativo. O $ID5R$ é um algoritmo incremental, sucessor do $ID5$ [Utgoff 88], que constrói a mesma árvore de decisão que o $ID3$, para um dado conjunto de treinamento. O algoritmo $ID5$ é equivalente ao $ID5R$. No $ID5$, entretanto, quando acontece uma reestruturação da árvore com o objetivo de levar um atributo desejado à posição de raiz, as subárvores não são reestruturadas recursivamente.

O $ID5R$ adota o mesmo artifício do $ID4$ ao manter nos nós a informação necessária para calcular a entropia de um atributo. Isso viabiliza a mudança do atributo de teste em um nó por outro que tenha um maior ganho de informação. Se a inserção de um novo exemplo na árvore provocar uma situação em que o atributo de teste no nó corrente não for o que tem o maior ganho de informação, a árvore deve ser reestruturada, e o atributo no nó substituído.

O $ID5R$ difere do $ID4$ na maneira como substitui o atributo no nó. Ao invés de eliminar as subárvores penduradas no nó a ser substituído, o $ID5R$ reestrutura a árvore empurrando o novo atributo de teste para cima. O processo de reestruturação — *pull-up* — é um rearranjo da árvore que preserva a consistência com as instâncias de treinamento até então observadas, e que coloca o atributo em questão como nó raiz da árvore ou subárvore.

Quatro novos algoritmos, identificados por $\widehat{}$, surgiram pelo uso de uma estratégia alternativa de construção da árvore — a de apenas reconstruir (ou reestruturar) a árvore atual — quando uma nova instância de treinamento for classificada erradamente por ela. Esses algoritmos são: $\widehat{ID3'}$, $\widehat{ID4}$, $\widehat{ID5}$, $\widehat{ID5R}$.

IDL — O *IDL* é um algoritmo que realiza indução incremental de árvores de decisão, tendo como principal objetivo a construção de árvores pequenas. A justificativa para obtenção de árvore pequenas, conforme apontado por Van de Velde em [Thrun 91], é por razões de eficiência, compreensibilidade e precisão de classificação.

5.10 Um Ambiente Experimental para a Família TDIDT

A multiplicidade de sistemas agrupados sob a família TDIDT comprova as diferentes abordagens e diferentes tratamentos dados a problemas e decisões que surgem quando da construção de árvores de decisão.

O objetivo do desenvolvimento do ambiente experimental TDIDT, implementado na linguagem de programação Prolog e mostrado nas Figuras 5.8 e 5.9, foi o de providenciar um ambiente controlável de software, flexível o suficiente para se adequar facilmente a mudanças, de maneira a poder testar idéias e novas abordagens.

O desenvolvimento teve início com uma implementação geral de algoritmos da família TDIDT, que possibilita facilmente a redefinição, para algoritmos particulares, do critério de avaliação e do critério de parada [Castiñeira 90, Castiñeira 91]. As estruturas de dados utilizadas foram listas e árvores. Com o objetivo de realizar o cálculo da função de avaliação eficientemente, a implementação em questão usa uma árvore binária duplamente indexada, como estrutura de dados da árvore de decisão. A estrutura geral da árvore de decisão construída pela implementação é

$$\textit{MelhorAtributo} - [v_1 : \textit{Subarv}_1, v_2 : \textit{Subarv}_2, \dots, v_n : \textit{Subarv}_n]$$

onde $v_1, v_2, \dots, v_n$ são os valores que *MelhorAtributo* pode assumir, e cada uma das *Subarv_i*, $i = 1, 2, \dots, n$ é uma subárvore que contém todos os exemplos para os quais o valor de *MelhorAtributo* é v_i . Cada uma dessas subárvores possui uma estrutura semelhante ou, então, representa um valor da classe, quando o nó for uma folha. Por exemplo, a árvore da Figura 5.1, pg. 35 na implementação realizada tem a seguinte representação⁷

```
TSH - [ <6.05 : negativo,
        >6.05 :
          FTI - [ <64.5 : hipo_primario,
                  >64.5 :
                    tiroxina -
                      [ v : negativo,
                      f : cirurgia_tir -
                        [ v : negativo,
                        f :
                          TT4 -
                            [ <150.5 : hipo_compensado,
                              >105.5 : negativo ]
                        ]
                      ]
                    ]
          ]
        ]
      ]
```

A nossa implementação leva em conta várias situações que podem acontecer durante o processo de construção da árvore. O procedimento `forme_arvore/4` é o que deflagra

⁷Neste trabalho, as respostas Prolog às interrogações, quando necessário, foram reformatadas, objetivando sua melhor compreensão.

Figura 5.8: Ambiente Experimental TDIDT

Figura 5.9: Visão Geral do Ambiente Experimental TDIDT

a construção da árvore de decisão. A idéia geral da atuação desse procedimento é explicada a seguir. Dada uma lista de exemplos e uma lista de atributos

1. obter o melhor atributo segundo um determinado critério
2. particionar os exemplos de acordo com os valores deste melhor atributo
3. retirar este melhor atributo da lista de possíveis atributos
4. continuar ramificando, recursivamente, cada uma dessas partições

A implementação trata as seguintes situações

- Não se pode mais ramificar a árvore, uma vez que todos os atributos já foram usados — a lista de atributos está vazia — e os exemplos do ramo em questão não pertencem à mesma classe. Quando isso acontece, determina-se qual é a classe com o maior número de exemplos — classe majoritária — e essa classe rotula o nó folha correspondente.
- Quando, para um determinado valor de atributo, não existirem exemplos no conjunto de treinamento que sejam classificados naquele correspondente ramo.
- Quando, durante o processo de construção da árvore, não é possível classificar precisamente exemplos que tenham o mesmo valor de atributos, mas que pertencem a classes diferentes. Neste caso, após a manipulação de todos os exemplos do conjunto de aprendizado, a estratégia adotada é a de rotular essa folha correspondente com o valor da classe mais provável.
- Algumas implementações precisam conhecer, durante o processo de construção da árvore, todos os valores de um atributo. Se um determinado valor de um atributo considerado pela árvore de decisão não estiver no conjunto de aprendizado, é criado, associado a esse valor, uma folha adicional, rotulada como *null*. Esta abordagem — que requer conhecimento não somente dos atributos, mas de todos os possíveis valores que cada atributo pode assumir — exige o pré-processamento de todo o conjunto de exemplos. Além disso, não há garantia que exemplos do

conjunto de teste — considerado posteriormente — possuam os mesmos valores de atributos que o conjunto de aprendizado.

Para resolver este problema, na nossa implementação as folhas *null* são criadas dinamicamente — após a construção da árvore — se necessário. Isto acontece em uma situação em que for processado um exemplo que não consegue ser classificado pela árvore, por possuir um valor de atributo que não está presente na árvore. Tal fato significa que não é possível uma classificação exata para esse exemplo. Há várias formas, contudo, de se contornar este problema. Uma delas, adotada neste trabalho, é a de computar as probabilidades condicionais de cada classe no ramo que vai da raiz à folha *null* da árvore, e rotular a folha *null* com a classe mais provável.

O sistema também possui opções para construir a árvore com informações adicionais, tais como o número total de exemplos classificados em cada um dos nós da árvore, o número de exemplos desclassificados etc.

A fim de lidar com um conjunto de treinamento grande, existe disponível uma implementação do mecanismo de janela, como descrito na Seção 5.6, pg. 43. O tratamento de ruídos foi implementado utilizando

- o conceito de *árvore estatística*, construída a partir de uma árvore e de um conjunto de exemplos de teste
- os métodos de poda da *Redução do Erro* e seu *Variante*

Este ambiente tem se mostrado extremamente útil para testar novas idéias. A fim de ilustrar a sua utilização, o Apêndice C, pg. 177, mostra uma mesma situação de aprendizado, sendo tratada pelo nosso ambiente e pelo *Assistant Professional*.

É importante notar que os procedimentos foram implementados inicialmente em Arity Prolog [Arity 91], para ambiente PC/DOS, depois migrados para ambiente Unix em WS⁸, usando o Prolog de Edinburgh [Edinburgh 87] e, atualmente, estão sendo customizados para o Sicstus Prolog [Sicstus 93], ambiente Unix, recentemente adquirido, que possui melhores recursos que o Prolog de Edinburgh. O Ambiente Experimental TDIDT faz parte de um ambiente geral, que denominamos de *Aprendizado Proposicional*, descrito a seguir.

5.11 O Ambiente de Aprendizado Proposicional

O ambiente de aprendizado proposicional pode ser abordado como dois subambientes: um totalmente experimental, descrito na seção anterior⁹ e o outro constituído, até o momento, de cinco sistemas/programas de aprendizado proposicional existentes, conseguidos via solicitação e/ou ftp. O ambiente de aprendizado proposicional é mostrado

⁸Workstation.

⁹Que inclui o Ambiente Experimental Construtivo, descrito no próximo capítulo.

na Figura 5.10, onde o papel do customizador proposicional é o de adequar as entradas ao formato exigido por cada um dos softwares. Os sistemas que fazem parte do segundo subambiente, são brevemente descritos a seguir.

Figura 5.10: Ambiente de Aprendizado Proposicional

IDT — É uma implementação mais simplificada do algoritmo ID3, que aquela do ambiente experimental TDIDT. Foi desenvolvida em 1989 junto ao Laboratório de Inteligência Artificial e Ciências de Computação da Universidade do Porto, Portugal, por Luis Torgo, e vem sendo mantida desde então. Foi originalmente implementada para Quintus Prolog, mas já está operacional sob o Sicstus Prolog [Sicstus 93].

OC1 — O sistema OC1[Murthy 93] (Oblique Classifier 1) é um sistema que induz árvores de decisão, e foi especialmente projetado para aplicações onde as instâncias têm features que assumem valores numéricos. O sistema OC1 constrói árvores de decisão que contêm combinações lineares de um ou mais atributos em cada nó interno da árvore; essas árvores particionam então o espaço de exemplos em hiperplanos. OC1 também constrói árvores de decisão usuais, que contêm testes de apenas um atributo em cada nó.

C4.5 — Esse sistema é uma versão mais recente do sistema C4, tratado na Seção 5.9, pg. 48.

NewId — O NewId [Boswell 90] é um sistema indutivo de aprendizado que, a partir de um conjunto de exemplos (vetores de atributos-valores de atributos), gera uma árvore de decisão que os classifica. Permite também que a precisão da árvore gerada possa ser avaliada (em termos de um conjunto pré-classificado de exemplos), e implementa também a poda da árvore. Pode traduzir a árvore de decisão obtida para código C. Essa facilidade permite que as regras aprendidas pelo sistema possam, como um programa, ser diretamente incorporadas a um programa C (por exemplo, um Sistema Especialista).

CN2 — O CN2 [Clark 89] é um sistema que, a partir de um conjunto de exemplos (vetores de atributos-valores de atributos) gera um conjunto de regras de produção que classificam esses exemplos. Permite também que a precisão do conjunto de regras geradas possa ser avaliada (em termos de um conjunto de exemplos pré-classificados). O CN2 pode ser considerado um sistema híbrido TDIDT/AQ uma vez que incorpora idéias de algoritmos de ambas famílias.

O CN2 adota a habilidade de lidar com dados com ruído dos algoritmos da família TDIDT e a estratégia flexível de busca, da família AQ. A versão CN2 disponível é a 4.1, de janeiro de 1990 [Boswell 90a], implementada em C para WS.

5.12 Considerações Finais

Indução de árvores de decisão é um processo limitado, tanto pela quantidade quanto pela qualidade dos dados disponíveis. De maneira análoga a técnicas estatísticas, métodos que induzem árvores de decisão podem apenas capturar padrões existentes nos dados. A qualidade de uma árvore de decisão, como classificadora de novas instâncias, é intrinsecamente dependente da qualidade e representatividade dos exemplos que a geraram.

Árvore de decisão como representação de conhecimento é limitada. O algoritmo básico de construção de árvore de decisão pode apenas gerar árvores de uma única forma específica — cada nó representa um único atributo. A cada nó o melhor atributo é escolhido; entretanto, uma vez feita a escolha, ela não pode ser desfeita usando, por exemplo, *backtracking*.

Uma possível extensão desta abordagem é a de associar múltiplos atributos a um nó, o que torna o processo de busca mais complexo. Árvores de decisão são mais difíceis de serem entendidas do que conjuntos de regras ou *frames*; entretanto, a transformação de árvores de decisão em regras de produção é simples. A modificação de uma árvore de decisão, de maneira a adequá-la a novas instâncias de treinamento, é um processo complicado que, via de regra, implica sua completa recriação. Os algoritmos ID5 e variações, entretanto, oferecem a possibilidade de construí-las incrementalmente. A implementação de algoritmos incrementais é objeto de outro trabalho sendo desenvolvido [Martins 93], e tão logo esteja disponível, será incorporada ao ambiente experimental TDIDT. É interessante lembrar, contudo, que a construção da árvore usando métodos incrementais é um processo altamente oneroso, uma vez que implica a constante avaliação da árvore construída até então e, quando for o caso, reestruturações de subárvores.

Capítulo 6

Aprendizado Construtivo

6.1 Considerações Iniciais

Como visto anteriormente, um problema fundamental presente em qualquer sistema indutivo de aprendizado, que interfere na qualidade do conceito induzido, é o da determinação de quais atributos são significativos à tarefa do aprendizado. Se os atributos captam as propriedades essenciais do conceito, o papel do sistema de aprendizado é o de rearranjar esses atributos em uma assertiva indutiva; entretanto, se os atributos usados nas descrições dos exemplos não expressarem as relações inerentes ao processo classificatório, as generalizações produzidas por qualquer algoritmo de aprendizado terão baixo desempenho na classificação de novos exemplos. Geralmente, o que se verifica é uma relação direta entre a qualidade dos atributos escolhidos para a descrição dos exemplos e o número de exemplos necessários para o aprendizado.

Os exemplos do conjunto de treinamento são descritos, muitas vezes, em termos de atributos iniciais não diretamente relevantes ao conceito mas que, quando convenientemente compostos formando novos atributos — chamados de *features* —, passam a ser diretamente relevantes.¹ Os atributos iniciais são referenciados como *features primitivas*. O processo de construção de features pode ser direcionado e controlado pelo usuário ou, então, automaticamente conduzido pelo sistema. Quando o processo de construção de features está inserido no contexto de um sistema de aprendizado indutivo, o aprendizado realizado é conhecido como *aprendizado construtivo* ou *indução construtiva*.

Neste capítulo é discutido o uso de indução construtiva como uma das maneiras de dinamicamente ampliar o conjunto inicial de descritores presentes na linguagem utilizada para a descrição do conjunto de treinamento. Os benefícios da ampliação da linguagem de descrição que se refletem na expressão do conceito serão, igualmente, discutidos. Indução construtiva é abordada no domínio de funções booleanas.

Três algoritmos que implementam *biases*² de construção automática de features, usando

¹Neste trabalho, os termos *atributo* e *feature* serão usados indistintamente.

²Heurísticas que direcionam o aprendizado. Neste trabalho, também tratados como estratégias.

árvore de decisão como formalismo para a representação das descrições induzidas, são descritos em detalhe. São eles: *root* [Matheus 90], *fringe* [Pagallo 90] e *root-fringe* [Nicoletti 92a], esse último proposto por nós. Devido ao alto custo computacional associado a um processo indutivo construtivo — na maioria das vezes envolvendo um número alto de iterações — é por nós proposta uma estratégia de pré-processamento do conjunto inicial de treinamento, na tentativa de minimizar tais custos. O pré-processamento constrói algumas possíveis features, antes do início do processo construtivo. Por fim, é descrito o ambiente experimental, resultante desses trabalhos, que implementa indução construtiva.

6.2 Ampliação da Linguagem de Descrição de Exemplos Através da Construção de Features

Em um sistema indutivo de aprendizado, com uma linguagem de representação de conceitos fixa, a granularidade das features determina a complexidade da regra de classificação. Features apropriadas de alto nível facilitam a representação do conceito; entretanto, se elas forem primitivas de baixo nível na expressão do conceito, o sistema de aprendizado freqüentemente deverá compilar uma regra complexa para expressar suas hipóteses [Flann 86]. Essas regras complexas, usualmente, são difíceis de serem identificadas e, via de regra, não são facilmente inteligíveis para as pessoas. A determinação de features significativas que sejam apropriadas é um problema complexo, como comentado em [Breiman 84, pg. 140]

a construção de features é uma ferramenta poderosa para a melhoria tanto da precisão quanto do entendimento da estrutura, particularmente em problemas com muitas dimensões. A seleção de features é uma arte guiada pela intuição do analista e pela exploração preliminar dos dados.

Um exemplo típico, freqüentemente citado na literatura de AM para ilustrar a importância da ampliação da linguagem de descrição de exemplos é o seguinte. Suponha que seja necessário encontrar as descrições que identificam um robô amigo e inimigo, respectivamente, em função de um conjunto de características, dado um conjunto de exemplos expressos em função dessas características e de seus valores, como mostrado na Tabela 6.1.

Atributo-valor					classe
<i>sorri</i>	<i>segura</i>	<i>tem_gravata</i>	<i>cabeça</i>	<i>corpo</i>	
sim	balão	sim	quadrada	quadrada	amigo
sim	bandeira	sim	triangular	triangular	amigo
sim	espada	sim	redonda	triangular	inimigo
sim	espada	não	quadrada	triangular	inimigo
não	espada	não	triangular	redonda	inimigo
não	bandeira	não	redonda	triangular	inimigo

Tabela 6.1: Exemplos de Robôs Amigos e Robôs Inimigos

O algoritmo da família TDIDT (ver Seção 5.3, pg. 35), utilizando a entropia como função de avaliação, induz a árvore de decisão mostrada na Figura 6.1.

Figura 6.1: Árvore de Decisão que Classifica um Robô em *amigo* ou *inimigo*

No caso do exemplo considerado, os atributos *tem_gravata*, *cabeça* e *corpo* são irrelevantes na indução daquela árvore de decisão. Entretanto, se a linguagem de descrição for estendida, combinando os atributos primitivos *cabeça* e *corpo* através do operador = em um novo atributo, *mesma_forma*, o conjunto de exemplos considerando esse novo atributo passa a ser aquele mostrado na Tabela 6.2.

Atributo-valor						classe
<i>sorri</i>	<i>segura</i>	<i>tem_gravata</i>	<i>cabeça</i>	<i>corpo</i>	<i>mesma_forma</i>	
sim	balão	sim	quadrada	quadrada	v	amigo
sim	bandeira	sim	triangular	triangular	v	amigo
sim	espada	sim	redonda	triangular	f	inimigo
sim	espada	não	quadrada	triangular	f	inimigo
não	espada	não	triangular	redonda	f	inimigo
não	bandeira	não	redonda	triangular	f	inimigo

Tabela 6.2: Exemplos de Robôs Amigos e Robôs Inimigos Estendidos com o Novo Atributo *mesma_forma*

Para este último conjunto de exemplos, a árvore de decisão correspondente está mostrada na Figura 6.2. Nela, pode ser visto que o novo atributo criado — *mesma_forma* — é mais informativo que os primitivos.

Figura 6.2: Árvore de Decisão que Classifica um Robô em *amigo* ou *inimigo* Usando a Feature *mesma_forma*

A construção de features que descrevam apropriadamente os exemplos de treinamento pode ser feita por um especialista (quando existe um) no domínio de conhecimento [Rodrigues 93a] ou, então, através de um sistema de aprendizado que automaticamente as detecte e as construa. Uma das linhas de pesquisa deste trabalho investiu no desenvolvimento de um ambiente de aprendizado automático de features no domínio de funções booleanas. Matheus [Matheus 90] define construção de features como

a aplicação de operadores construtivos³ à features já existentes, resultando na definição de uma ou mais novas features

O processo de construção de features é difícil devido ao tamanho do espaço definido pelas features candidatas. Em um dado instante, o número de features potenciais que podem ser construídas é uma função combinatória do número de features que já existem, multiplicado pelo número de operadores construtivos adotados. No caso geral, o processo de busca para determinar um conjunto apropriado de features é intratável. Sistemas que constroem features são viáveis apenas quando articulados a heurísticas poderosas ou, na terminologia de Aprendizado de Máquina, a biases corretos e poderosos, que possibilitam a redução do tamanho do espaço definido pelas features candidatas [Utgoff 86].

6.2.1 O Uso de Operadores Construtivos

Um *operador construtivo* é uma função que associa uma n-upla de features a uma nova feature. As features às quais o operador construtivo é aplicado são chamadas de *operandos construtivos*. Como por sua vez a feature construída pode servir como operando, a construção de features é um processo iterativo. Desta maneira, operadores construtivos simples (com aridade⁴ pequena), podem ser aplicados repetidamente na construção de features mais complexas. É importante notar que, independentemente da complexidade da feature obtida, a informação que ela fornece a respeito das instâncias é sempre definida como uma composição das features primitivas.

O uso de operadores construtivos viabiliza a interpretação do processo de construção de features como um problema de busca, como pode ser visualizado na Figura 6.3. Na figura, cada nó representa um conjunto de features; o estado inicial representa o conjunto de features primitivas usadas na descrição do conjunto de treinamento e os estados finais são conjuntos de features apropriadas para a descrição do conceito. As transições de um estado para outro são obtidas através da aplicação de operadores construtivos.

Devido à natureza composicional do processo construtivo, o espaço de busca pode ser ilimitado, i.e., infinito. Apesar da Figura 6.3 mostrar o espaço como um grafo direcionado, a adição de procedimentos de avaliação de features e de poda pode, eventualmente, produzir um grafo bidirecional e cíclico. Como resultado disso, buscas exaustivas (por exemplo, *breadth-first* e *depth-first* — ver Apêndice A, pg. 169) são intratáveis no caso geral. Soluções práticas do problema de construção de features devem considerar uma redução drástica do tamanho do espaço de busca ou, então, a adoção de heurísticas efetivas que viabilizam a determinação de um conjunto de features apropriado.

Um dos problemas fundamentais em aprendizado construtivo é o relacionado à escolha de qual operador⁵ construtivo utilizar e, uma vez feito isso, determinar a quais operandos tal operador vai ser aplicado. Mesmo com o uso de biases poderosos, o processo de

³Operadores usados para compor features.

⁴Número de argumentos do operador.

⁵Pode existir mais do que um.

Figura 6.3: Construção de Feature: um Problema de Busca

construção de features é, via de regra, computacionalmente caro. É importante observar que quanto maior for o conjunto de features, maior será o número de possíveis operadores construtivos potenciais. No caso de um nó consistindo de N features booleanas, o número de caminhos potenciais⁶ saindo do nó é 2^{2^N} . No caso de existirem duas features booleanas f_1 e f_2 , a Tabela 6.3 mostra a atuação dos possíveis 16 operadores (Op_0 até Op_{15}) que podem ser construídos, ou seja $Op_i(f_1, f_2)$, $i = 0, \dots, 15$.

Op_0			Op_1			Op_2			Op_3			Op_4			Op_5			Op_6			Op_7		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1
1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1

Op_8			Op_9			Op_{10}			Op_{11}			Op_{12}			Op_{13}			Op_{14}			Op_{15}		
0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
0	1	0	0	1	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1	0	1	1
1	0	0	1	0	0	1	0	1	1	0	1	1	0	0	1	0	0	1	0	1	1	0	1
1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1	1	0	1	1	1

Tabela 6.3: Dezesesseis Possíveis Operadores Construídos a Partir de Duas Features Booleanas — Op_1 Corresponde ao *and* Booleano e Op_7 Corresponde ao *or*

6.2.2 Principais Características de um Sistema Construtivo

São vários os aspectos a serem considerados em sistemas que implementam aprendizado construtivo. O sistema Citre [Matheus 90], por exemplo, que é um sistema de aprendizado indutivo que usa árvores de decisão para a representação de conceitos, implementa um processo de indução construtiva e incorpora as características mais relevantes de sistemas construtivos.

O Citre seleciona os operandos que irão compor as novas features nos ramos positivos

⁶Obtidos pela aplicação dos possíveis operadores.

da árvore de decisão. Embora neste aspecto seja semelhante ao *fringe* [Pagallo 89, Pagallo 90] (a ser tratado na Seção 6.3, pg. 63), difere deste por abordar o processo de construção indutiva de maneira sistemática, adotando a seguinte seqüência de 4 subprocessos

- *Deteção da necessidade do uso de features* — Uma vez que o processo de construção de features pode ser computacionalmente dispendioso, uma fase preliminar para a detecção da necessidade de novas features é recomendada. É evidente que em sistemas de aprendizado capazes de adquirir a expressão adequada do conceito — a partir do conjunto primitivo de features —, o processo de construção de novas features é, obviamente, supérfluo. É importante notar que a detecção da necessidade do uso de uma técnica construtiva difere da detecção de sua aplicabilidade. A necessidade de novas features pode existir, mesmo não existindo a viabilidade de implementação de técnicas para a sua construção e vice-versa, ou seja, a implementação de técnicas de construção de features pode ser perfeitamente factível, podendo, no entanto, não existir a necessidade de novas features.

Sistemas projetados para a construção automática de features não dispõem, frequentemente, de mecanismos para detecção da necessidade do uso de features. Elas são construídas compulsoriamente.

- *Seleção de construtores*⁷ — A dificuldade com o processo de seleção se deve ao fato do espaço de construtores potenciais ser intratavelmente grande, na maioria dos casos. Durante a seleção deve ser identificado um subconjunto pequeno de construtores potenciais que vão conduzir a um conjunto de features apropriadas ao problema de aprendizado em questão.

Operadores simples têm a vantagem de serem aplicáveis a uma vasta gama de problemas. Em contrapartida, porém, o seu uso requer um número considerável de iterações do processo construtivo, de maneira a construir features mais complexas. Os operadores específicos do problema, por sua vez, podem reduzir a complexidade do processo de construção e, conseqüentemente, o tempo envolvido. Em contrapartida, entretanto, são limitados àquele problema necessitando, eventualmente, de esforço e conhecimento adicional para serem implementados. Alguns sistemas usam conhecimento específico do domínio, para realizar a seleção de construtores. O conhecimento do domínio, geralmente, é especificado na forma de restrições o que, de certa forma, provoca indiretamente a incorporação de tal conhecimento no processo de construção de features e, conseqüentemente, no processo indutivo como um todo.

- *Generalização de construtores* — O conjunto de construtores evidenciados na fase anterior pode, dependendo dos métodos de seleção usados, ser muito específico. Existe, pois, a possibilidade de generalização desses construtores antes de efetivamente serem incorporados como features. Construtores específicos podem ser generalizados de maneira análoga à generalizações que ocorrem em alguns

⁷Operador e seus operandos.

Figura 6.4: Representação da Relação entre o Sistema e o Processo de Construção de Features

métodos de aprendizado (e.g., substituindo constantes por variáveis, eliminando termos, etc.).

- *Avaliação de features* — Uma vez que a adição de novas features aumenta o tempo de processamento de um sistema de aprendizado, eficiência pode ser frequentemente melhorada limitando-se o número de features ativas. Através de um processo de avaliação de features, aquelas menos representativas podem ser descartadas. A poda de features não representativas ao processo de aprendizado não apenas melhora o tempo de aprendizado, como também produz hipóteses mais precisas e concisas.

Essas quatro fases, perfeitamente delineadas no Citre, não estão necessariamente presentes em sistemas que realizam indução construtiva; na maioria das vezes, mesmo quando presentes, elas não são perfeitamente identificadas no sistema.

Na Figura 6.4 o sistema de aprendizado e o processo de construção de features são abordados como dois processos separados, muito embora essa separação não seja, necessariamente, evidente a nível de implementação na maioria dos sistemas. Nessa figura, I e I' representam instâncias de treinamento, F e F' conjuntos de features, H hipóteses e K teoria do domínio.

6.3 Aprendizado de Funções Booleanas com Construção Automática de Features

Uma árvore de decisão booleana representa uma função booleana. Cada ramo do nó raiz à uma folha rotulada 1 define um termo da função. A função representada por tal árvore é a disjunção de todos os termos assim obtidos, estando, conseqüentemente, na Forma Normal Disjuntiva — FND.

O aprendizado de uma função booleana a partir de sua tabela-verdade parcial consiste em gerar uma árvore de decisão que a represente. Como apontado em [Pagallo 90], a representação em árvore de decisão de uma função booleana com uma descrição FND pequena tem uma estrutura peculiar: a mesma seqüência de testes de decisão que conduz à uma folha positiva é replicada na árvore — problema representacional conhecido como *problema da replicação*.

Devido ao problema da replicação durante o aprendizado de uma árvore de decisão, a estratégia de partição deve dividir os exemplos de um termo em muitos subconjuntos. Isso faz com que o algoritmo exija um número bem grande de exemplos, de maneira a garantir que os subconjuntos sejam numerosos o suficiente para estimativas probabilísticas precisas; se isso não acontecer, o processo de subdivisão irá ramificar incorretamente ou, então, terminará prematuramente. Uma possível maneira de solucionar este problema é através do uso da conjunção de features nos nós de decisão.

Como já mencionado, neste trabalho será abordada a construção automática de features, especificamente, o aprendizado de funções booleanas através de indução com construção automática de features. É importante observar que muitas das estratégias desenvolvidas para aprendizado no domínio das funções booleanas, podem ser reutilizadas e aplicadas para o aprendizado em outros domínios. Isso acontece, particularmente, em situações nas quais a expressão do exemplo tem a informação (booleana) da ausência ou presença de determinado atributo. O conjunto de tais atributos pode ser isolado, métodos construtivos de construção de features booleanas podem ser aplicados e as features assim construídas, incorporadas às expressões dos exemplos e ao conjunto de atributos.

6.4 Definição do Operador Construtivo

Para a pesquisa de aprendizado construtivo de funções booleanas, foi escolhido o operador construtivo

$$and(operando_1, operando_2)$$

onde

$$operando_i = \langle feature \ operator_relacional \ valor_da_feature \rangle \quad (i = 1, 2)$$

são obtidos nos ramos positivos da árvore de decisão. O operador relacional relaciona a feature a seus possíveis valores.

Seja, por exemplo, a função booleana $f(x_1, x_2, x_3, x_4) = \sum min(1, 3, 5, 7)$ ⁸ representada pela árvore de decisão da Figura 6.5.

⁸Esta notação é usada para representar uma função na Forma Normal Disjuntiva. No exemplo, a função $f(x_1, x_2, x_3, x_4) : \{0, 1\}^4 \rightarrow \{0, 1\}$ assume valor 1 para os seguintes elementos do domínio: $\langle 0, 0, 0, 1 \rangle, \langle 0, 0, 1, 1 \rangle, \langle 0, 1, 0, 1 \rangle$ e $\langle 0, 1, 1, 1 \rangle$, ou seja $f(x_1, x_2, x_3, x_4) = \overline{x_1}x_2\overline{x_3}x_4 \vee \overline{x_1}x_2x_3x_4 \vee \overline{x_1}x_2\overline{x_3}x_4 \vee \overline{x_1}x_2x_3x_4$.

Figura 6.5: Árvore de Decisão da Função $f(x_1, x_2, x_3, x_4) = \sum \min(1, 3, 5, 7)$

Convencionou-se que o ramo de um nó que sai à sua esquerda está associado ao valor 0 do atributo que aquele nó representa, e o que sai à sua direita, ao valor 1. Estabelecida a convenção, a árvore da Figura 6.5 é lida como

- quando $x_1 = 0$ e $x_4 = 0$ o valor da função é 0
- quando $x_1 = 0$ e $x_4 = 1$ o valor da função é 1
- quando $x_1 = 1$ o valor da função é 0

A expressão da função pode ser reduzida através da construção da feature que compõe os atributos x_1 e x_4 (estão num ramo de comprimento ≥ 2 cuja folha é 1), e a árvore passa, então, a ser aquela mostrada na Figura 6.6.

Figura 6.6: Árvore de Decisão da Função $f(x_1, x_2, x_3, x_4) = \sum \min(1, 3, 5, 7)$ com a Feature $f_1 = \text{and}(x_1 = 0, x_4 = 1)$

No caso geral, a quais pares de $\langle \text{feature} \text{ operador_relacional } \text{valor_da_feature} \rangle$ o operador contrutivo se aplica vai depender do bias de seleção de operandos escolhido ser *root*, *fringe* ou *root-fringe*, definidos e tratados na próxima seção.

A abordagem de construção de features adotada neste trabalho deixa o algoritmo de aprendizado definir e ampliar adaptativamente suas próprias features durante o processo de aprendizado, característica conhecida como *bias dinâmico*.

6.5 Estratégias para Construção de Features

Existem várias estratégias possíveis para construção de features. A seguir serão discutidas três delas: *root* [Matheus 90], *fringe* [Pagallo 90] e *root-fringe* [Nicoletti 92b], esta última considerada como uma combinação das duas primeiras. As três estratégias de composição são, na realidade, três biases disponíveis para a seleção dos operandos

do operador construtivo *and*, que irão compor as features. Os três biases consideram apenas os ramos positivos — classe 1 — da árvore de decisão.

Figura 6.7: Biases para a Construção de Features

Em cada passo de iteração, após a construção da árvore, se o bias escolhido for

- *root* — são selecionadas as duas primeiras features de cada ramo, a partir da raiz. Compõe a cada passo features com os menores valores de entropia⁹, ou seja, features que mais contribuem para a classificação
- *fringe* — são selecionadas as duas últimas features de cada ramo, a partir das folhas. Compõe a cada passo features com os maiores valores de entropia, ou seja, features que menos contribuem para a classificação
- *root-fringe* — são selecionadas a primeira e a última feature de cada ramo. Compõe a cada passo uma feature com um baixo valor de entropia, com uma outra com alto valor de entropia

A Figura 6.7 mostra, para uma mesma árvore de decisão, a atuação de cada um dos três biases na construção de features. As três estratégias usam o mesmo operador *and* para compor as features e diferem entre si, unicamente, na forma como escolhem os operandos aos quais o *and* vai ser aplicado. A seleção dos operandos é realizada durante o processo de aprendizado, a partir da árvore de decisão obtida. Nenhuma das três estratégias generaliza ou faz uma avaliação dos construtores obtidos. Eles são, compulsoriamente, anexados ao conjunto de atributos e cada exemplo do conjunto de treinamento é expandido com a introdução dos construtores obtidos e seus respectivos valores.

Como o processo de construção de features envolve a construção da árvore de decisão, a seguir é apresentado o algoritmo que constrói iterativamente a árvore de decisão usando

⁹Nos experimentos por nós conduzidos, foi utilizada a entropia como função de avaliação para a construção da árvore. Entretanto, qualquer outra função de avaliação $aval(t)$ (ver Capítulo 5 pg. 33) poderia ter sido utilizada.

Figura 6.8: Construção de Árvore de Decisão com Construção Automática de Features

features, bem como as descrições dos algoritmos que implementam os três biases de construção de features.

6.6 Descrição dos Algoritmos

Dados um conjunto de features primitivas V e um conjunto de exemplos E , expressos em termos de V , o algoritmo começa construindo uma árvore de decisão T , cujos nós são elementos de V . Uma vez construída a árvore, um procedimento iterativo que implementa um dos três biases de construção de features é invocado. Este procedimento pode ser visualizado na Figura 6.8.

6.6.1 Algoritmo Iterativo de Construção de Árvore de Decisão com Construção Automática de Features

```

{  $V$ : conjunto de features primitivas
 $E$ : conjunto de exemplos expressos em função de  $V$  }
begin
  input :  $V, E$ 
   $k \leftarrow 0$  {contador do passo de iteração}
   $V_1 \leftarrow V$ 
  repeat
     $k \leftarrow k + 1$ 
    construir a árvore de decisão  $T_k$ , usando  $E$  expresso em função de  $V_k$ 
     $\text{conjunto\_features} \leftarrow \text{encontra\_feature\_n } \{n = 1, 2, 3\}$ 
     $V_{k+1} \leftarrow V_k \cup \text{conjunto\_features}$ 
  until ( $V_{k+1} = V_k$  or  $|V_{k+1}| \geq M$ )
  output  $T_k, V_k$ 
end

```

O processo de construção das features irá — a cada passo da iteração — compor, através de conjunção, dois nós da árvore. A escolha de quais dos nós é definida pelo bias utilizado. Os procedimentos que correspondem à construção das features são

- `encontra_feature_1`, que implementa o bias *root*
- `encontra_feature_2`, que implementa o bias *fringe*
- `encontra_feature_3`, que implementa o bias *root-fringe*

Cada um dos dois procedimentos: `encontra_feature_2` e `encontra_feature_3` constrói uma feature para cada folha com rótulo 1 da árvore, desde que tal folha se encontre a uma profundidade, na árvore, maior ou igual a 2. Assim, serão construídas tantas features quantas forem tais folhas.

O procedimento `encontra_feature_1` constrói, no máximo, 4 features em cada passo da iteração, uma vez que este bias sempre elege como operandos do operador *and* a raiz da árvore e um filho seu, desde que estejam num ramo com uma folha rotulada 1.

Obtidas as novas features, o conjunto de exemplos é reescrito, ampliando cada exemplo com a introdução das novas features e de seus respectivos valores. O conjunto de features é também atualizado, de maneira a incorporar as features geradas. Com esses novos conjuntos — de exemplos e atributos — é gerada uma nova árvore de decisão e o processo se repete. O processo termina quando não são geradas novas features ou, então, quando o número de features geradas atinge um limite pré-estabelecido — M .

Nos experimentos realizados (ver Capítulo 7, pg. 79) apenas a primeira condição de parada foi considerada, ou seja, o processo de geração de features prossegue até que não sejam mais geradas novas features.

A seguir são apresentados os três procedimentos de construção de features, todos eles acompanhados de um exemplo de utilização no aprendizado de uma mesma função booleana. Para os três procedimentos

- T representa a árvore de decisão construída a partir do conjunto de exemplos, expressos em função das features primitivas e das features obtidas até então, por chamadas anteriores à `encontra_feature.i` ($i = 1, 2, 3$)
- nós de T são representados pelas variáveis: $x_1, \dots, x_n$
- a função a ser aprendida é dada por $f(x_1, x_2, x_3) = \sum \min(0, 1, 2, 5)$, ou seja, $f(x_1, x_2, x_3) = \overline{x_1}x_2x_3 \vee \overline{x_1}\overline{x_2}x_3 \vee \overline{x_1}x_2\overline{x_3} \vee x_1\overline{x_2}x_3$

6.6.2 Algoritmo Root

```

begin {Bias usado: root}
  input  $T$ 
   $\text{conjunto\_features} \leftarrow \emptyset$  {conjunto de features é inicializado vazio}
   $\text{raiz\_T} \leftarrow \text{raiz de } T$  {um dos  $x_1, \dots, x_n$ }
  para cada ramo correspondente a  $\text{valor\_raiz\_T}$  (0 ou 1) de  $\text{raiz\_T}$ ,
  com comprimento  $\geq 2$ 
    begin
       $x_i \leftarrow \text{filho de raiz\_T}$ ,
      para cada ramo correspondente a cada  $\text{valor\_}x_i$  de  $x_i$ ,
      begin
        para a primeira folha rotulada 1 encontrada,
        if
           $\text{valor\_raiz\_T} = 0$  e  $\text{valor\_}x_i = 0$  então
             $\text{nova\_feature} = \text{and}(\text{not raiz\_T}, \text{not } x_i)$  10
           $\text{valor\_raiz\_T} = 0$  e  $\text{valor\_}x_i = 1$  então  $\text{nova\_feature} = \text{and}(\text{not raiz\_T}, x_i)$ 
           $\text{valor\_raiz\_T} = 1$  e  $\text{valor\_}x_i = 0$  então  $\text{nova\_feature} = \text{and}(\text{raiz\_T}, \text{not } x_i)$ 
           $\text{valor\_raiz\_T} = 1$  e  $\text{valor\_}x_i = 1$  então  $\text{nova\_feature} = \text{and}(\text{raiz\_T}, x_i)$ 
        end if
         $\text{conjunto\_features} \leftarrow \text{conjunto\_features} \cup \text{nova\_feature}$ 
        abandona o ramo
      end
    end
  return  $\text{conjunto\_features}$ 
end

```

Seqüência de Construção de Árvores usando o Bias Root

Figura 6.9: Root - Primeira Iteração

¹⁰Notação equivalente a $\text{and}(\text{raiz_T} = 0, x_i = 0)$

Figura 6.10: *Root* - Segunda Iteração

Figura 6.11: *Root* - Terceira e Última Iteração

6.6.3 Algoritmo Fringe

```

begin {Bias usado: fringe}
  input T
  conjunto_features ← ∅ {conjunto de features é inicializado vazio}
  raiz_T ← raiz de T {um dos  $x_1, \dots, x_n$ }
  para cada folha rotulada 1, com profundidade maior ou igual a 2.
    begin
       $x_i \leftarrow$  pai de folha;  $x_j \leftarrow$  avô de folha;
      sejam valor_ $x_i$  e valor_ $x_j$  os valores dos ramos que conduzem à folha
      if
        valor_ $x_i$  = 0 e valor_ $x_j$  = 0 então nova_feature = and(not  $x_i$ , not  $x_j$ )
        valor_ $x_i$  = 0 e valor_ $x_j$  = 1 então nova_feature = and(not  $x_i$ ,  $x_j$ )
        valor_ $x_i$  = 1 e valor_ $x_j$  = 0 então nova_feature = and( $x_i$ , not  $x_j$ )
        valor_ $x_i$  = 1 e valor_ $x_j$  = 1 então nova_feature = and( $x_i$ ,  $x_j$ )
      endif
      conjunto_features ← conjunto_features ∪ nova_feature
    return conjunto_features
  end

```

Sequência de Construção de Árvores usando o Bias *Fringe*

Figura 6.12: *Fringe* - Primeira Iteração

Figura 6.13: *Fringe* - Segunda Iteração

Figura 6.14: *Fringe* - Terceira Iteração

Figura 6.15: *Fringe* - Quarta e Última Iteração

6.6.4 Algoritmo Root-Fringe

```
begin {Bias usado: root-fringe}
  input T
  conjunto_features  $\leftarrow \emptyset$  {conjunto de features é inicializado vazio}
  raiz_T  $\leftarrow$  raiz de T {um dos  $x_1, \dots, x_n$ }
  para cada ramo correspondente a cada valor_raiz_T (0 ou 1) de raiz_T,
    para cada folha rotulada 1, com profundidade maior ou igual a 2,
      begin
        seja  $x_i$  o nó pai dessa folha, e valor_ $x_i$  o ramo que chega em folha.
        if
          valor_raiz_T = 0 e valor_ $x_i$  = 0 então
            nova_feature = and(not raiz_T, not  $x_i$ )
          valor_raiz_T = 0 e valor_ $x_i$  = 1 então nova_feature = and(not raiz_T,  $x_i$ )
          valor_raiz_T = 1 e valor_ $x_i$  = 0 então nova_feature = and(raiz_T, not  $x_i$ )
          valor_raiz_T = 1 e valor_ $x_i$  = 1 então nova_feature = and(raiz_T,  $x_i$ )
        endif
        conjunto_features  $\leftarrow$  conjunto_features  $\cup$  nova_feature
      end
  return conjunto_features
end
```

Sequência de Construção de Árvores usando o Bias Root-Fringe

O desempenho, entre as três estratégias de composição de atributos, foi avaliado através de uma série de experimentos, tratados detalhadamente no próximo capítulo. Como o processo de construção de features, por suas características interativas, tem um alto custo computacional devido, principalmente, ao número de vezes que a árvore de decisão é construída, decidimos investigar métodos que pudessem aliviar tal custo. Um deles, proposto e discutido a seguir, investe no pré-processamento do conjunto de treinamento, extraindo os primos-implicantes da função e incorporando-os como novas features.

Figura 6.16: *Root-Fringe* - Primeira Iteração

Figura 6.17: *Root-Fringe* - Segunda Iteração

Figura 6.18: *Root-Fringe* - Terceira e Última Iteração

6.7 Pré-Processamento do Conjunto de Treinamento

Os chamados métodos de minimização de funções booleanas minimizam as formas normais de funções booleanas de uma maneira típica. Dada uma função booleana expressa na *Forma Normal Disjuntiva*, um método de minimização procura encontrar uma outra FND equivalente à primeira, que contenha o menor número de produtos envolvendo o menor número de literais.

6.7.1 Minimização de Funções Booleanas

Um método de minimização bastante utilizado é o de *Quine-McCluskey* [Quine 52, McCluskey 56]. Este método é diretamente aplicável a circuitos combinatórios com qualquer número de variáveis de entrada e uma única variável de saída. Métodos de minimização são baseados no conceito de *implicantes primos*.

Definição 6.7.1 (*Implicante Primo*) Uma forma booleana¹¹ α cobre outra forma booleana β se $\nu(\alpha) = 1$ para todas as atribuições de valores que tornam $\nu(\beta) = 1$, onde ν é uma função de avaliação binária.

Se α cobre β , e β é um produto simples, então β é um implicante de α . Se β é um implicante de α , e nenhuma outra forma que possa ser obtida a partir de β , pela remoção de literais é um implicante de α , então β é um implicante primo de α .

Numa soma minimal de produtos equivalente à uma FND, os produtos devem todos ser implicantes primos. Desde que uma soma de produtos tem valor 1, sempre que um produto nesta soma tem valor 1, todo produto da soma minimal de produtos deve ser um implicante da forma normal. Além disso, se algum produto não for implicante primo, ele pode ser substituído por um implicante contendo um menor número de literais i.e., a soma de produtos não será minimal. O procedimento Quine-McCluskey gera primeiramente o conjunto de todos os primos implicantes da forma normal e, então, seleciona o subconjunto dos implicantes primos que definem a soma minimal de produtos.

Após os implicantes primos terem sido encontrados, a soma minimal de produtos é gerada. Os *minterms* cobertos apenas por um implicante primo tornam aquele implicante primo um *implicante primo essencial*. Todo implicante primo essencial deve participar do conjunto minimal.

De uma maneira geral, a determinação da soma minimal de produtos não é muito simples. Alguns dos minterms podem, eventualmente, não ser cobertos por nenhum dos implicantes primos essenciais. Quando isso acontece a forma minimal deverá conter, pelo menos, um implicante primo adicional aos implicantes primos essenciais. Pode também acontecer que nenhum dos implicantes primos encontrados seja essencial. Neste caso, o problema de minimização tem mais do que uma solução. No experimento descrito no próximo capítulo, foi usado o conjunto completo de implicantes primos.

¹¹ Expressão de uma função booleana.

O algoritmo iterativo de construção de árvore de decisão com construção automática de features usado foi o descrito na Seção 6.6.1 pg. 67, antecedido por um pré-processamento do conjunto de treinamento, conforme mostrado na Figura 6.19.

Figura 6.19: Pré-Processamento do Conjunto de Treinamento

6.7.2 Algoritmo para o Pré-Processamento do Conjunto Treinamento

begin *início pré-processamento*

$\{V$: conjunto de features primitivas

E : conjunto de exemplos expressos em função de $V\}$

input : V, E

$P \leftarrow$ implicantes primos da função (aplicação do Quine-McCluskey)

$E' \leftarrow$ exemplos de E expandidos com a adição dos elementos de P
e seus respectivos valores

$V' \leftarrow P \cup V$

output V', E'

end *término do pré-processamento*

6.8 Ambiente Experimental Construtivo

Todos os algoritmos descritos neste Capítulo foram também implementados na linguagem de programação lógica Prolog, e fazem parte do ambiente experimental mostrado na Figura 6.20, onde ADF refere-se a Árvore de Decisão com Features. Atualmente, a biblioteca de algoritmos para construção de features possui as implementações dos três algoritmos *root*, *fringe* e *root-fringe*. Pretendemos, no futuro, estender essa biblioteca com implementações de outros algoritmos, com vistas a uma possível análise teórica do problema de construção de features de funções booleanas, que continua ainda um problema em aberto.

Foi também implementado um gerador de exemplos de funções booleanas — ver Figura 6.21 — que, dada a expressão de uma função booleana na FND, e um inteiro n , cria randomicamente um conjunto de n instâncias desta função, no formato requerido pelo ambiente.

Figura 6.20: Visão Geral do Ambiente Experimental Construtivo

Figura 6.21: Módulo Gerador de Exemplos

Como mencionado anteriormente, o ambiente TDIDT implementado permite gerar qualquer tipo de árvore de decisão. Porém, para viabilizar essa generalidade, bem como para facilitar a aplicabilidade da função de avaliação de atributos, foi necessário utilizar uma estrutura de dados altamente complexa na construção da árvore. Geralmente, os algoritmos que manipulam a árvore são bastante complexos. No caso de aprendizado de funções booleanas, entretanto, como os atributos possuem somente dois possíveis valores, a árvore resultante é binária. No caso específico de funções booleanas, é possível utilizar uma estrutura de dados bem mais simples, ou seja, ao invés de se usar uma estrutura geral de árvore de decisão, pode-se usar diretamente uma estrutura binária. Desta forma, as implementações correspondentes à construção e poda da árvore, bem como à construção de features utilizando as diferentes estratégias, podem ser simplificadas. Essa simplificação permite uma maior facilidade para incorporar, no futuro, implementações de outros algoritmos, bem como outras estratégias de construção de operadores construtivos.

Este ambiente TDIDT simplificado, denominado ambiente TDIDT para a construção automática de features de funções booleanas, já está sendo implementado. Os algoritmos para a construção da árvore de decisão binária, mecanismo de pós-poda e procedimentos necessários para a análise estatística da árvore gerada, já foram implementados. A descrição dessas implementações encontram-se em [Martins 94]. Os algoritmos para

a construção automática de features estão em fase de teste e em [Monard 94] estão sendo descritos os detalhes de suas implementações.

6.9 Considerações Finais

Neste capítulo foi discutida a importância da ampliação da linguagem de descrição de exemplos, no aprendizado de conceitos. Tal ampliação foi abordada no domínio de funções booleanas, através de três algoritmos para a construção de novas features. A ampliação da linguagem de descrição, para aquele domínio, foi ainda abordada considerando o tratamento do conjunto de treinamento, através de pré-processamento, para evidenciar os implicantes primos da função, como possíveis novas features. O ambiente experimental construtivo, que se articula ao ambiente experimental TDIDT, foi apresentado e detalhado.

Capítulo 7

Avaliação Empírica dos Algoritmos para Construção de Features

7.1 Considerações Iniciais

Este Capítulo descreve uma série de experimentos realizados relacionados a aprendizado construtivo de funções booleanas, utilizando as três estratégias de construção de features discutidas no capítulo anterior. Os resultados do desempenho das três estratégias são apresentados e analisados. Descreve também experimentos realizados com dois métodos de poda no ambiente de aprendizado construtivo de funções booleanas: os métodos de poda da Redução do Erro e seu Variante são empiricamente avaliados usando resultados dos experimentos de construção de features. Por fim, experimentos relacionados ao pré-processamento do conjunto de treinamento, com o objetivo de reduzir o número de vezes que a árvore de decisão é construída, são descritos e seus resultados analisados.

7.2 Avaliação das Estratégias de Construção de Features

Os três biases foram experimentalmente verificados no aprendizado de funções booleanas do tipo *l-termo kFND*. Uma função booleana *l-termo kFND* é expressa pela disjunção de *l* termos, onde cada um deles tem, no máximo, *k* features. Por exemplo

$$(\bar{x}_1 \ x_5 \ x_4) \vee (x_7 \ \bar{x}_3) \text{ é uma função 2-termo 3FND}$$

Com a variação dos valores de *l* e *k* pode-se automaticamente gerar funções a serem aprendidas que têm diferentes graus de complexidade. Neste trabalho os experimentos envolveram as cinco funções booleanas seguintes

$$\begin{aligned}
f_1(x_2, x_3, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{15}) &= \overline{x_7}x_3x_{10} \vee x_{12}\overline{x_9}x_{11} \vee x_8x_2\overline{x_{15}} \\
f_2(x_1, x_2, x_4, x_5, x_6, x_8, x_9, x_{10}, x_{11}) &= \overline{x_6}x_5\overline{x_4} \vee x_{11}\overline{x_2}x_1 \vee x_9x_8\overline{x_{10}} \\
f_3(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_{10}, x_{11}) &= \overline{x_5}x_7x_3 \vee x_3x_4x_1 \vee x_{11}\overline{x_{10}}x_2
\end{aligned}$$

$$\begin{aligned}
f_{10}(x_1, x_2, x_3, x_4, x_5, x_7, x_8, x_{10}, x_{11}, x_{12}, x_{14}, x_{16}) &= \overline{x_8}x_{11}\overline{x_4} \vee x_{12}\overline{x_3}x_5 \vee \overline{x_1}x_{16}x_7 \vee x_2\overline{x_{10}}x_{14} \\
f_{11}(x_1, x_3, x_5, x_6, x_8, x_9, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}) &= \overline{x_9}x_{11}\overline{x_{14}} \vee x_{15}x_5x_{16} \vee \overline{x_1}x_6\overline{x_{13}} \vee x_{12}\overline{x_8}x_3
\end{aligned}$$

Tais funções têm, respectivamente, a seguinte representação na implementação Prolog realizada

3-termo 3FND
fnd(1,[[n x7, x3, x10], [x12, n x9, n x11], [x8, x2, n x15]])
fnd(2,[[n x6, n x5, n x4], [x11, n x2, x1], [x9, x8, n x10]])
fnd(3,[[x5, n x7, x3], [x6, x4, x1], [x11, n x10, x2]])

4-termo 3FND
fnd(10,[[n x8, n x11, n x4], [x12, n x3, n x5], [n x1, n x16, x7], [x2, n x10, x14]])
fnd(11,[[n x9, x11, n x14], [x15, x5, x16], [n x1, x6, n x13], [x12, n x8, n x3]])

Em cada uma das funções consideradas, as features primitivas comparecem apenas uma vez e cada um dos termos das cinco funções utilizadas tem, exatamente, 3 features primitivas.

Nos experimentos realizados, o conjunto original de features primitivas foi ampliado com a introdução de 25% de features irrelevantes, como sugerido em [Pagallo 90]. Tais features irrelevantes são referenciadas neste trabalho como variáveis espúrias. Com o acréscimo de variáveis espúrias, o total de features primitivas passa a ser $1.33 * k * l^1$. Desta forma, foram introduzidas 3 variáveis espúrias nas funções 3-termo 3FND e 4 variáveis espúrias nas funções 4-termo 3FND.

Nos experimentos com as funções 3-termo 3FND e 4-termo 3FND, cada uma das cinco funções ($f_1, f_2, f_3, f_{10}, f_{11}$) foi usada para criar 15 conjuntos de exemplos de treinamento: 5 com 50 exemplos (Grupo A), 5 com 100 exemplos (Grupo B) e 5 com 150 exemplos (Grupo C). Foram, pois, criados 75 conjuntos de exemplos de treinamento.

Para a avaliação da precisão das árvores obtidas foram gerados randomicamente, para cada função, 3 conjuntos de teste com 20, 40 e 60 exemplos, respectivamente. Tais valores correspondem a aproximadamente 40% de cada um dos três tamanhos de conjuntos de treinamento utilizados para o aprendizado da função. Isto é, as funções aprendidas com um conjunto de treinamento com 50 exemplos, foram validadas com um conjunto de teste com 20 exemplos, as aprendidas com um conjunto de treinamento com 100 exemplos, foram validadas com um conjunto de teste com 40 exemplos e as geradas a partir de 150 exemplos, validadas com um conjunto de 60 exemplos.

¹Do total de features T , as variáveis espúrias representam 25% e as features primitivas contidas nas FND, 75%. Nas funções l -termo k FND utilizadas neste trabalho o número de features primitivas é dado por $k * l$. Assim sendo $T = T/4 + (k * l)$, ou seja, $T = 4/3 * k * l$.

Em cada um dos grupos (A, B e C), para cada um dos 5 conjuntos de treinamento, foi realizado o aprendizado da função a ele associada, através da construção da árvore de decisão, utilizando cada um dos vieses: *root*, *fringe* e *root-fringe*, bem como o aprendizado da função sem a construção de features — aqui referenciado como árvore exata. As 5 árvores obtidas relativas à aplicação de um mesmo bias, bem como as 5 obtidas sem aplicação de qualquer bias, foram medidas com relação a uma série de características relevantes para a avaliação empírica dos algoritmos.

Ao término desta fase foram obtidas as medidas (média de 5) de características das árvores para cada uma das FND, em cada um dos grupos A, B e C. As características medidas estão listadas na Tabela 7.1. Tais medidas foram agrupadas por classe de FND, isto é, 3-termo 3FND e 4-termo 3FND.

NF	Número de Folhas
NN	Número de Nós
H	Altura
CC	Comprimento de Caminho
P(%)	Precisão da Árvore
NI	Número de Iterações
FC	Número de Features Distintas Construídas
FA	Número de Features Distintas na Árvore
FFA	Número de Features na Árvore
PF	Número de Primitivas Distintas nas Features da Árvore
FPF	Número de Primitivas nas Features
Pr	Número de Primitivas Distintas na Árvore
FPr	Número de Primitivas na Árvore
Es	Número de Variáveis Espúrias na Árvore
FES	Número de Variáveis Espúrias nas Features da Árvore

Tabela 7.1: Características Medidas

7.2.1 Análise dos Resultados

Uma vez que as funções booleanas usadas no experimento foram randomicamente extraídas das classes de funções booleanas 3-termo 3FND e 4-termo 4FND, os resultados exibidos nas Tabelas 7.2 e 7.3 — para os conjuntos de treinamento com 50, 100 e 150 exemplos — mostram a atuação dos três vieses naquelas classes de função, e não simplesmente em funções randômicas individualizadas. As entradas em branco nas tabelas indicam que a característica em questão não se aplica. Para uma visualização mais rápida entre os desempenhos dos vieses, com relação às características medidas, optamos por representar várias das medidas obtidas, pelos três vieses, de forma pictórica. Nas Figuras, a abcissa identifica o bias usado, observando a seguinte convenção

E	Exata
F	Fringe
R	Root
RF	Root-Fringe

Os resultados obtidos são discutidos a seguir.

	<i>crata</i>	<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos				
NF	14.8 (0.4)	5.2 (0.4)	12.9 (0.2)	11.1 (0.5)
NN	28.6 (0.9)	9.4 (0.9)	24.9 (0.3)	21.3 (1.0)
H	5.6 (0.2)	4.2 (0.4)	6.2 (0.3)	5.9 (0.4)
CC	100.5 (3.0)	24.0 (5.4)	95.1 (7.1)	76.4 (3.5)
P(%)	82.5 (5.5)	91.5 (3.5)	87.5 (3.5)	84.5 (5.0)
NI		5.7 (0.4)	4.2 (0.4)	3.7 (0.1)
FC		18.9 (1.5)	9.7 (1.5)	14.1 (0.2)
FA		3.7 (0.3)	2.2 (0.3)	3.1 (0.5)
FFA		3.7 (0.3)	2.3 (0.3)	3.1 (0.5)
PF		7.8 (0.7)	3.5 (0.3)	3.8 (0.4)
FPF		11.5 (1.8)	6.2 (1.4)	7.1 (1.5)
Pr	7.6 (0.4)	0.5 (0.6)	6.1 (0.2)	5.3 (0.4)
FPr	12.5 (0.7)	0.5 (0.6)	8.8 (0.6)	6.3 (0.4)
Es	0.4 (0.2)	0.02 (0.03)	0.3 (0.1)	0.2 (0.1)
FEs		0.2 (0.1)	0.2 (0.1)	0.2 (0.1)
100 exemplos				
NF	24.1 (1.8)	4.0 (0.1)	14.5 (1.5)	14.9 (1.1)
NN	47.0 (3.2)	7.0 (0.1)	28.1 (2.9)	28.9 (2.2)
H	6.5 (0.3)	3.0 (0.1)	6.5 (0.1)	6.5 (0.2)
CC	193.9 (17.8)	12.0 (0.1)	112.7 (11.1)	114.8 (10.0)
P(%)	85.8 (3.5)	100.0 (0.0)	92.5 (4.5)	91.75 (4.0)
NI		4.2 (0.2)	3.9 (0.4)	3.8 (0.2)
FC		16.1 (1.0)	9.7 (1.5)	18.7 (0.7)
FA		3.0 (0.1)	1.9 (0.2)	2.6 (0.2)
FFA		3.0 (0.1)	1.9 (0.2)	2.8 (0.2)
PF		9.0 (0.1)	3.7 (0.5)	4.3 (0.2)
FPF		9.0 (0.1)	5.3 (0.9)	7.1 (0.2)
Pr	9.0 (0.1)	0.0 (0.0)	6.7 (0.5)	6.6 (0.2)
FPr	21.6 (1.0)	0.0 (0.0)	11.3 (1.3)	10.4 (0.6)
Es	0.5 (0.2)	0.0 (0.0)	0.1 (0.1)	0.2 (0.1)
FEs		0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
150 exemplos				
NF	27.5 (1.4)	4.0 (0.1)	15.3 (0.5)	14.3 (0.2)
NN	54.1 (2.8)	7.0 (0.1)	29.7 (1.0)	27.7 (0.5)
H	6.8 (0.2)	3.0 (0.1)	6.7 (0.2)	6.8 (0.2)
CC	236.5 (17.0)	12.0 (0.1)	123.5 (4.7)	114.1 (3.1)
P(%)	88.5 (1.7)	99.0 (0.1)	97.2 (1.3)	95.5 (1.3)
NI		4.1 (0.1)	3.9 (0.1)	4.0 (0.1)
FC		15.7 (0.6)	9.7 (0.3)	18.5 (0.6)
FA		3.0 (0.1)	2.1 (0.1)	2.6 (0.3)
FFA		3.0 (0.1)	2.1 (0.1)	2.9 (0.2)
PF		9.0 (0.1)	3.9 (0.2)	4.4 (0.2)
FPF		9.0 (0.1)	5.8 (0.4)	7.8 (0.6)
Pr	8.9 (0.1)	0.0 (0.0)	6.4 (0.3)	6.5 (0.2)
FPr	25.3 (0.8)	0.0 (0.0)	11.8 (0.6)	10.1 (0.4)
Es	0.4 (0.2)	0.0 (0.0)	0.2 (0.1)	0.1 (0.1)
FEs		0.0 (0.0)	0.0 (0.0)	0.0 (0.0)

Tabela 7.2: 3-termo 3FND

	<i>exata</i>	<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos				
NF	14.2 (1.0)	5.6 (0.4)	12.6 (0.1)	13.0 (0.2)
NN	27.4 (2.0)	10.2 (0.8)	24.2 (0.2)	25.0 (0.4)
H	5.5 (0.1)	4.5 (0.3)	5.8 (0.1)	5.8 (0.2)
CC	92.4 (9.6)	25.6 (3.6)	85.8 (3.0)	87.8 (4.2)
P(%)	67.5 (2.5)	84.0 (5.0)	69.0 (1.0)	67.5 (1.5)
NI		6.5 (0.1)	3.5 (0.3)	3.0 (0.2)
FC		21.6 (0.6)	7.9 (0.9)	11.8 (1.8)
FA		4.5 (0.3)	2.0 (0.2)	1.4 (0.6)
FFA		4.5 (0.3)	2.1 (0.1)	1.4 (0.6)
PF		9.1 (0.7)	3.1 (0.3)	2.0 (0.6)
FPF		12.5 (0.5)	5.1 (0.3)	3.3 (1.5)
Pr	8.2 (0.8)	0.1 (0.1)	6.6 (0.8)	6.8 (0.1)
FPr	11.6 (1.2)	0.1 (0.1)	8.5 (0.3)	9.3 (0.3)
Es	0.4 (0.1)	0.0 (0.0)	0.3 (0.1)	0.3 (0.1)
FEs		0.3 (0.1)	0.1 (0.1)	0.0 (0.0)
100 exemplos				
NF	27.7 (1.3)	6.3 (0.7)	22.4 (1.2)	23.0 (1.0)
NN	54.4 (2.6)	11.6 (1.4)	43.8 (2.4)	45.0 (2.0)
H	6.8 (0.4)	5.0 (0.4)	7.8 (0.2)	8.1 (0.1)
CC	235.0 (16.2)	34.2 (7.8)	209.6 (12.8)	217.8 (6.6)
P(%)	69.8 (0.3)	93.3 (0.8)	78.3 (0.8)	79.3 (1.3)
NI		5.9 (0.1)	4.0 (0.1)	3.8 (0.4)
FC		31.2 (3.2)	10.3 (0.1)	21.2 (2.4)
FA		4.8 (0.2)	2.6 (0.2)	3.7 (0.1)
FFA		4.8 (0.2)	2.6 (0.2)	3.7 (0.1)
PF		12.0 (0.1)	3.9 (0.3)	4.2 (0.6)
FPF		16.3 (0.7)	6.8 (0.2)	9.1 (0.9)
Pr	11.7 (0.1)	0.4 (0.4)	4.4 (4.4)	9.8 (0.2)
FPr	24.8 (0.6)	0.5 (0.5)	17.2 (1.0)	16.6 (0.4)
Es	0.5 (0.2)	0.0 (0.0)	0.4 (0.1)	0.4 (0.1)
FEs		0.1 (0.1)	0.1 (0.1)	0.1 (0.1)
150 exemplos				
NF	31.7 (1.3)	5.3 (0.3)	25.9 (0.7)	26.9 (1.7)
NN	62.4 (2.6)	9.6 (0.6)	50.8 (1.4)	52.8 (3.4)
H	7.5 (0.5)	4.3 (0.3)	7.7 (0.1)	7.5 (0.1)
CC	285.8 (20.2)	23.6 (3.6)	257.4 (7.0)	252.4 (17.2)
P(%)	72.5 (7.5)	98.8 (1.2)	81.5 (1.8)	77.0 (1.0)
NI		4.7 (0.1)	4.0 (0.1)	4.1 (0.1)
FC		30.1 (3.9)	10.0 (0.5)	35.2 (3.8)
FA		4.3 (0.3)	2.7 (0.1)	5.5 (0.3)
FFA		4.3 (0.3)	2.8 (0.2)	5.9 (0.3)
PF		12.0 (0.1)	4.1 (0.1)	6.6 (0.1)
FPF		13.7 (1.7)	7.5 (0.3)	15.4 (1.0)
Pr	11.9 (0.1)	0.0 (0.0)	10.2 (0.2)	9.8 (0.4)
FPr	28.1 (0.9)	0.0 (0.0)	20.7 (0.7)	18.3 (1.5)
Es	0.7 (0.1)	0.0 (0.0)	0.4 (0.1)	0.4 (0.1)
FEs		0.1 (0.1)	0.0 (0.0)	0.1 (0.1)

Tabela 7.3: 4-termo 3FND

NF: Número de Folhas — Figuras 7.1 e 7.2

O número de folhas de uma árvore de decisão expressa o número de regras de decisão que representam a árvore em questão. As árvores aprendidas usando *fringe* são as que se expressam com o menor número de regras. As árvores aprendidas com *root* e *root-fringe* são expressas com aproximadamente o mesmo número de regras.

O valor de NF para *fringe* é menor do que para os outros dois biases por um fator de 2 no mínimo e de 5 no máximo.

NN: Número de Nós H: Altura — Figuras 7.1 e 7.2 — **CC: Comprimento de Caminho**

Essas três medidas de uma árvore se relacionam diretamente ao seu tamanho e forma. A altura de uma árvore representa o número máximo de termos que as regras de decisão que expressam a árvore têm. O comprimento de caminho da árvore dá uma idéia mais geral da complexidade da árvore em função desses parâmetros.

As árvores aprendidas com *fringe* são as menores e de melhor forma. A complexidade das árvores aprendidas com *root* e *root-fringe* é semelhante. O valor de CC para *fringe* é menor do que para os outros dois biases por um fator de 3 no mínimo e de 10 no máximo.

Figura 7.1: 3-termo 3FND - o: H •: NF x:NN

P: Precisão da Árvore — Figura 7.3

A melhor precisão na classificação de novos exemplos é obtida pelas árvores geradas por *fringe*; para os outros dois biases é semelhante.

Figura 7.2: 4-termo 3FND - o: H •: NF x:NN

NI: Número de Iterações — Figura 7.4

Por número de iterações entende-se o número de vezes que os dois processos — o de construção da árvore e de construção de features — se repetem. O NI é maior para *fringe*. Pode se notar, entretanto, a tendência dos números de iterações dos três biases se tornarem os mesmos, à medida que o conjunto de treinamento aumenta de tamanho.

Figura 7.3: (P%)Precisão - ◇: 3-termo 3 FND ×: 4-termo 3 FND

Figura 7.4: (NI)Número de Iterações - ○: 3-termo 3FND ×: 4-termo 3FND

FC: Número de Features Distintas Construídas — Figuras 7.5 e 7.6

Pode ser notado que esse número é bem menor para *root*. Isso se deve ao fato que este bias constrói 4 features no máximo por iteração. Sob esse ponto de vista, o *root* pode ser considerado um método econômico (é importante lembrar que a cada iteração as novas features são incorporadas ao conjunto de treinamento). Em geral, o FC é maior para *fringe*, chegando, em determinadas situações, a ser duas vezes maior que o número de features do conjunto original usadas na descrição inicial do conjunto de treinamento.

FA: Número de Features Distintas na Árvore — Figuras 7.5 e 7.6

Em geral é maior para *fringe*. Este fato não se verifica apenas para a classe das 4-termo 3FND aprendidas a partir de um conjunto com 150 elementos.

Figura 7.5: 3-termo 3FND - $\diamond$: (FA)Features Distintas na Árvore $\bullet$: (FC)Features Distintas Construídas

Figura 7.6: 4-termo 3FND - $\diamond$: (FA)Features Distintas na Árvore $\bullet$: (FC)Features Distintas Construídas

FFA: Número de Features na Árvore

Este número dá uma idéia da distribuição das features na árvore. Em todos os biases a relação $FFA/FA \approx 1$ é verificada. Isto indica que as features que entram na árvore aprendida, o fazem apenas uma vez.

PF: Número de Primitivas Distintas nas Features da Árvore

Este número se mostrou sempre bem maior nas árvores geradas por *fringe*. É um indicativo que *fringe* consegue eleger muito bem quais operandos vão compor as features que constrói. O valor do PF vai aumentando à medida que aumenta o número de exemplos de treinamento, até chegar ao limite dado pelo número total de primitivas da FND. Este fato não é observado nos outros dois biases.

FPF: Número de Primitivas nas Features da Árvore

A relação $FPF/PF \approx 1$ é verificada para as árvores aprendidas com *fringe*. Para os outros dois biases o que se verifica é $FPF/PF > 1$, o que indica que as mesmas primitivas são utilizadas por mais de uma feature.

Pr: Número de Primitivas Distintas na Árvore — Figuras 7.7 e 7.8

Este número é praticamente zero para *fringe*, indicando que na árvore final somente comparecem features. Para os outros dois biases isto não acontece.

FPr: Número de Primitivas na Árvore — Figuras 7.7 e 7.8

Este número dá uma idéia da distribuição das primitivas na árvore. Como comentado anteriormente, as primitivas praticamente não comparecem nas árvores geradas por *fringe*. O valor do FPr, para os outros dois biases, é semelhante. A relação $FPr/Pr = 1$ indica que as primitivas aparecem apenas uma vez na árvore. Este valor aumenta à medida que aumenta a repetição das primitivas na árvore. Esta última situação acontece quando a representatividade das primitivas para a expressão do conceito é pequena. Assim, pode-se dizer que as primitivas aparecem mais próximas das folhas da árvore quanto maior for a relação FPr/Pr . Os resultados obtidos não permitem realizar uma comparação entre os biases *root* e *root-fringe*.

Figura 7.7: 3-term 3FND - x: (Pr)Primitivas Distintas na Árvore •: (FPr)Primitivas na Árvore

Es: Número de Variáveis Espúrias na Árvore

O melhor caso é quando $Es = 0$, indicando que nenhuma variável espúria comparece sozinha em um nó de decisão da árvore. Isto é praticamente o que acontece nas árvores criadas por *fringe*. O pior comportamento acontece na construção das árvores exatas, piorando à medida que cresce o conjunto de treinamento. O comportamento de *root* e *root-fringe* é médio, já que algumas variáveis espúrias aparecem como nós de decisão

Figura 7.8: 4-termo 3FND - x: (Pr)Primitivas Distintas na Árvore •: (FPr)Primitivas na Árvore

das árvores.

FEs: Número de Variáveis Espúrias nas Features da Árvore

De maneira análoga à Es, o melhor caso acontece quando FEs = 0, indicando que nenhuma variável espúria comparece nas features que fazem parte da árvore final. O comportamento dos três algoritmos com relação ao FEs é semelhante ao descrito para Es.

7.2.2 Avaliação dos Biases Root, Fringe e Root-Fringe

Os resultados obtidos mostram que *fringe* tem um comportamento muito superior aos outros dois biases, enquanto que os comportamentos de *root* e de *root-fringe* são semelhantes.

Os biases *root* e *fringe* podem ser considerados como opostos, no sentido que o primeiro começa a construção das features pela raiz da árvore — onde estão as features que mais contribuem para a classificação — enquanto que o segundo começa a construção das features pelas folhas. O bias *root-fringe* pode ser considerado como uma composição desses dois biases, já que a construção das features é realizada escolhendo os nós do início e fim dos ramos positivos da árvore.

Deve ser observado que eram esperados resultados distintos para os três biases, o que efetivamente não aconteceu. Além dos experimentos comprovarem a grande superioridade do *fringe*, não foi possível detectar qualquer diferença essencial entre os desempenhos de *root* e *root-fringe*.

Os resultados mostraram que a introdução de variáveis espúrias teve pouco impacto no aprendizado das funções, indicando que os algoritmos realmente ignoram features não essenciais.

Como já mencionado, um aspecto negativo da construção iterativa de features é o seu alto custo computacional. A árvore deve ser construída várias vezes até que o critério

de parada seja satisfeito. Este aspecto negativo foi o que nos motivou a pesquisar formas alternativas de reduzir o custo computacional de *fringe* através de um pré-processamento dos exemplos. A solução por nós proposta é a discutida na Seção 6.7, pg. 75, que foi avaliada através dos experimentos descritos na Seção 7.4, pg. 97.

7.3 Descrição dos Experimentos com Métodos de Poda e Análise dos Resultados

Como discutido na Seção 5.7, pg. 45, métodos de poda são usados para melhorar a precisão e compreensibilidade de árvores de decisão. Geralmente, a aplicação de tais métodos torna as árvores de decisão mais precisas e concisas. Os experimentos descritos e discutidos a seguir foram conduzidos objetivando examinar o quanto a estratégia de aprendizado construtivo, utilizando os biases *root*, *fringe* e *root-fringe* — que faz uso de árvores de decisão — pode ser afetada com o uso de mecanismos de poda.

Os métodos de poda de Redução do Erro [Quinlan 87] e seu Variante [Castiñeira 90a] foram escolhidos com vistas a

- investigar a contribuição do mecanismo de poda quando usado em uma situação de aprendizado construtivo
- comparar empiricamente o desempenho dos dois métodos

A experimentação com os dois métodos de poda, bem como a análise comparativa do desempenho de ambos em uma situação de aprendizado construtivo, teve por base dados obtidos do experimento anterior (ver Seção 7.2, pg. 79), relativos à avaliação experimental do desempenho dos biases *root*, *fringe* e *root-fringe*.

A Tabela 7.4, extraída das Tabelas 7.2 e 7.3, mostra o número de iterações necessárias à obtenção das árvores de decisão para cada um dos biases avaliados.

Para as árvores de decisão aprendidas no experimento anterior, ambos os métodos de poda foram aplicados. Para viabilizar a aplicação dos mecanismos de poda foram criados 15 conjuntos de testes, formados por exemplos gerados randomicamente. Para cada FND foram criados 3 conjuntos de testes (com 20, 40 e 60 exemplos, que correspondem a 40% do número de exemplos — 50, 100 e 150 respectivamente — utilizados para o aprendizado da FND). As árvores podadas foram analisadas com relação a

NF	Número de Folhas
NN	Número de Nós
H	Altura
CC	Comprimento de Caminho
P(%)	Precisão da Árvore

Para a medida da precisão da árvore podada, os mesmos conjuntos de teste do primeiro experimento foram usados. Com o objetivo de facilitar a leitura, alguns dos resultados

3-termo 3FND		
<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos		
5.7 (0.4)	4.2 (0.4)	3.7 (0.1)
100 exemplos		
4.2 (0.2)	3.9 (0.4)	3.8 (0.2)
150 exemplos		
4.1 (0.1)	3.9 (0.1)	4.0 (0.1)

4-termo 3FND		
<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos		
6.5 (0.1)	3.5 (0.3)	3.0 (0.2)
100 exemplos		
5.9 (0.1)	4.0 (0.1)	3.8 (0.4)
150 exemplos		
4.7 (0.1)	4.0 (0.1)	4.1 (0.1)

Tabela 7.4: Número de Iterações Necessárias para a Construção da Árvore de Decisão

do primeiro experimento são novamente apresentados nas Tabelas 7.5 e 7.6 sob o título **sem poda**.

Os dois métodos de poda tiveram um desempenho semelhante; o método Variante, entretanto, se mostrou um pouco mais drástico: a árvore podada pelo método da Redução do Erro é, geralmente, maior em tamanho do que a árvore podada utilizando o método Variante — Figuras 7.9, 7.10, 7.11 e 7.12.

Figura 7.9: (H)Altura 3-termo 3FND - o: sem poda ◊: reduced ×: variante

Pode se notar que tanto a precisão — Figuras 7.13 e 7.14 — quanto o formato da árvore obtida usando poda são sempre melhores quando *fringe* é utilizado; a diferença, entretanto, não é tão significativa quando tais valores são comparados com os correspondentes das árvores obtidas sem poda. Quando o número de exemplos de treinamento aumenta, a precisão das árvores geradas usando cada um dos três biases é aproximadamente a mesma.

Figura 7.10: (H)Altura 4-termo 3FND - o: sem poda ◇: reduced ×: variante

Os resultados obtidos indicam que se a árvore de decisão for aprendida usando o bias *fringe* e um conjunto de treinamento com 100 ou 150 exemplos, os métodos de poda não interferem na concisão da árvore obtida e contribuem muito pouco para a melhoria de sua precisão — Figuras 7.13 e 7.14. Este resultado enfatiza a adequação da estratégia usada pelo *fringe*, de combinar pares de features com valores altos de entropia (features que não contribuem muito para a discriminação entre as classes). Nos experimentos conduzidos, o bias *fringe* mostrou-se tão eficiente que mesmo métodos de poda tornam-se supérfluos quando o conjunto de treinamento tem um número representativo de exemplos.

O mecanismo de poda mostrou-se conveniente quando a árvore de decisão é aprendida usando um dos dois biases *root* ou *root-fringe* e um conjunto de treinamento com 100 ou

Figura 7.11: (NF)Número Folhas 3-termo 3FND - o: sem poda ◇: reduced ×: variante

Figura 7.12: (NF)Número Folhas 4-termo 3FND - o: sem poda ◊: reduced x: variante

150 exemplos. Talvez para conjuntos de treinamento com maior número de exemplos as melhorias obtidas com o uso dos métodos de poda poderão diminuir. Pelos resultados dos experimentos evidenciou-se que o uso de poda promove a melhoria de árvores geradas usando *root* e *root-fringe*. Pode também ser observado, a partir dos resultados obtidos, que o processo de construção de features melhora a precisão e a concisão da árvore, mesmo sem o uso de poda.

Figura 7.13: (P%)Precisão 3-termo 3FND - o: sem poda ◊: reduced x: variante

Figura 7.14: (P%)Precisão 4-termo 3FND - o: sem poda ◊: reduced x: variante

	<i>sem bias</i>	<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos				
sem poda				
NF	14.8 (0.4)	5.2 (0.4)	12.9 (0.2)	11.1 (0.5)
NN	28.6 (0.9)	9.4 (0.9)	24.9 (0.3)	21.3 (1.0)
H	5.6 (0.2)	4.2 (0.4)	6.2 (0.3)	5.9 (0.4)
CC	100.5 (3.0)	24.0 (5.4)	95.1 (7.1)	76.4 (3.5)
P(%)	82.5 (5.5)	91.5 (3.5)	87.5 (3.5)	84.5 (5.0)
poda redução do erro				
NF	9.5 (1.5)	4.7 (0.1)	8.8 (1.2)	8.0 (0.9)
NN	17.9 (2.9)	8.3 (0.2)	16.6 (2.4)	15.0 (1.7)
H	4.7 (0.2)	3.7 (0.1)	5.4 (0.7)	5.2 (0.3)
CC	51.7 (11.9)	19.5 (1.8)	55.6 (12.7)	46.3 (7.5)
P(%)	92.3 (2.8)	96.4 (1.5)	94.3 (2.2)	93.5 (3.0)
poda variante				
NF	7.5 (2.4)	4.2 (0.3)	8.3 (1.6)	6.0 (1.3)
NN	14.0 (4.8)	7.4 (0.7)	15.5 (3.1)	11.0 (2.2)
H	3.7 (0.9)	3.2 (0.3)	5.1 (1.2)	4.1 (0.0)
CC	38.3 (15.8)	15.6 (4.5)	52.4 (14.5)	29.7 (9.1)
P(%)	90.7 (3.2)	96.1 (1.2)	90.3 (2.5)	92.4 (2.7)
100 exemplos				
sem poda				
NF	24.1 (1.8)	4.0 (0.1)	14.5 (1.5)	14.9 (1.1)
NN	47.0 (3.2)	7.0 (0.1)	28.1 (2.9)	28.9 (2.2)
H	6.5 (0.3)	3.0 (0.1)	6.5 (0.1)	6.5 (0.2)
CC	193.9 (17.8)	12.0 (0.1)	112.7 (11.1)	114.8 (10.0)
P(%)	85.8 (3.5)	100.0 (0.0)	92.5 (4.5)	91.75 (4.0)
poda redução do erro				
NF	14.1 (0.8)	4.0 (0.0)	11.0 (0.7)	9.9 (0.8)
NN	27.1 (1.5)	7.0 (0.0)	21.0 (1.4)	18.7 (1.5)
H	5.6 (0.2)	3.0 (0.0)	6.2 (0.3)	5.5 (0.6)
CC	92.7 (9.0)	12.0 (0.0)	78.1 (10.4)	63.9 (9.0)
P(%)	94.6 (0.5)	100.0 (0.0)	98.1 (1.5)	97.8 (1.0)
poda variante				
NF	11.5 (0.6)	4.0 (0.0)	10.8 (0.8)	9.6 (0.7)
NN	22.1 (1.2)	7.0 (0.0)	20.6 (1.7)	18.2 (1.4)
H	5.1 (0.2)	3.0 (0.0)	6.2 (0.3)	5.5 (0.6)
CC	71.2 (7.4)	12.0 (0.0)	76.8 (10.6)	61.6 (8.7)
P(%)	94.4 (0.6)	100.0 (0.0)	97.9 (1.5)	97.5 (0.4)
150 exemplos				
sem poda				
NF	27.5 (1.4)	4.0 (0.1)	15.3 (0.5)	14.3 (0.2)
NN	54.1 (2.8)	7.0 (0.1)	29.7 (1.0)	27.7 (0.5)
H	6.8 (0.2)	3.0 (0.1)	6.7 (0.2)	6.8 (0.2)
CC	236.5 (17.0)	12.0 (0.1)	123.5 (4.7)	114.1 (3.1)
P(%)	88.5 (1.7)	99.0 (0.1)	97.2 (1.3)	95.5 (1.3)
poda redução do erro				
NF	19.2 (1.1)	4.0 (0.0)	12.5 (0.2)	11.9 (0.8)
NN	37.4 (2.1)	7.0 (0.0)	24.0 (0.4)	22.7 (1.5)
H	6.1 (0.1)	3.0 (0.0)	6.5 (0.1)	6.2 (0.0)
CC	142.3 (10.8)	12.0 (0.0)	91.9 (1.5)	85.9 (8.5)
P(%)	96.6 (0.5)	100.0 (0.0)	99.2 (0.6)	98.6 (0.7)
poda variante				
NF	16.0 (1.3)	4.0 (0.0)	12.2 (0.3)	11.6 (0.8)
NN	31.0 (2.6)	7.0 (0.0)	23.4 (0.7)	22.2 (1.7)
H	5.9 (0.2)	3.0 (0.0)	6.3 (0.1)	6.1 (0.2)
CC	112.1 (11.7)	12.0 (0.0)	88.8 (2.0)	82.7 (9.8)
P(%)	96.4 (0.7)	100.0 (0.0)	99.0 (0.7)	98.6 (0.7)

Tabela 7.5: 3-termo 3FND

	<i>sem bias</i>	<i>fringe</i>	<i>root</i>	<i>root-fringe</i>
50 exemplos				
sem poda				
NF	14.2 (1.0)	5.6 (0.4)	12.6 (0.1)	13.0 (0.2)
NN	27.4 (2.0)	10.2 (0.8)	24.2 (0.2)	25.0 (0.4)
H	5.5 (0.1)	4.5 (0.3)	5.8 (0.1)	5.8 (0.2)
CC	92.4 (9.6)	25.6 (3.6)	85.8 (3.0)	87.8 (4.2)
P(%)	67.5 (2.5)	84.0 (5.0)	69.0 (1.0)	67.5 (1.5)
poda redução do erro				
NF	9.6 (1.8)	5.5 (0.5)	9.2 (0.6)	8.9 (0.5)
NN	18.2 (3.6)	10.0 (1.0)	17.4 (1.2)	16.8 (1.0)
H	4.7 (0.7)	4.4 (0.4)	5.2 (0.4)	4.8 (0.4)
CC	54.2 (13.4)	24.6 (4.6)	56.0 (6.0)	51.2 (4.0)
P(%)	87.8 (0.2)	93.8 (2.2)	88.2 (2.20)	89.0 (0.2)
poda variante				
NF	7.9 (1.5)	5.0 (0.0)	7.9 (0.5)	8.1 (0.5)
NN	14.8 (3.0)	9.0 (0.0)	14.8 (1.0)	15.2 (1.0)
H	4.0 (1.2)	3.9 (0.1)	4.4 (0.8)	4.4 (0.8)
CC	44.0 (10.0)	20.8 (0.8)	47.2 (3.2)	46.2 (3.8)
P(%)	87.2 (0.4)	93.2 (2.8)	87.2 (2.0)	88.2 (0.2)
100 exemplos				
sem poda				
NF	27.7 (1.3)	6.3 (0.7)	22.4 (1.2)	23.0 (1.0)
NN	54.4 (2.6)	11.6 (1.4)	43.8 (2.4)	45.0 (2.0)
H	6.8 (0.4)	5.0 (0.4)	7.8 (0.2)	8.1 (0.1)
CC	235.0 (16.2)	34.2 (7.8)	209.6 (12.8)	217.8 (6.6)
P(%)	69.8 (0.3)	93.3 (0.8)	78.3 (0.8)	79.3 (1.3)
poda redução do erro				
NF	14.2 (0.4)	5.0 (0.2)	11.3 (0.1)	12.5 (0.7)
NN	27.4 (0.8)	9.0 (0.4)	21.6 (0.2)	24.0 (1.4)
H	5.9 (0.1)	4.0 (0.2)	5.7 (0.3)	6.5 (0.1)
CC	94.6 (3.8)	20.2 (1.8)	77.6 (2.8)	93.4 (8.2)
P(%)	89.3 (2.1)	97.4 (0.2)	92.3 (0.3)	92.0 (0.4)
poda variante				
NF	10.5 (1.1)	4.6 (0.2)	8.5 (0.5)	10.0 (0.2)
NN	20.0 (2.2)	8.2 (0.4)	16.0 (1.0)	19.0 (0.4)
H	5.2 (0.2)	3.6 (0.2)	4.6 (0.2)	6.0 (0.0)
CC	63.6 (8.4)	17.0 (1.4)	53.0 (4.2)	68.4 (4.8)
P(%)	88.5 (2.5)	97.3 (0.3)	91.9 (0.3)	91.5 (0.3)
150 exemplos				
sem poda				
NF	31.7 (1.3)	5.3 (0.3)	25.9 (0.7)	26.9 (1.7)
NN	62.4 (2.6)	9.6 (0.6)	50.8 (1.4)	52.8 (3.4)
H	7.5 (0.5)	4.3 (0.3)	7.7 (0.1)	7.5 (0.1)
CC	285.8 (20.2)	23.6 (3.6)	257.4 (7.0)	252.4 (17.2)
P(%)	72.5 (7.5)	98.8 (1.2)	81.5 (1.8)	77.0 (1.0)
poda redução do erro				
NF	19.6 (0.6)	5.0 (0.0)	15.1 (2.3)	14.6 (0.6)
NN	38.2 (1.2)	9.0 (0.0)	29.2 (4.6)	27.5 (2.0)
H	6.4 (0.2)	4.0 (0.0)	6.6 (0.4)	6.6 (0.1)
CC	148.8 (7.2)	20.0 (0.0)	123.0 (28.2)	105.3 (9.8)
P(%)	91.4 (1.0)	100.0 (0.0)	93.5 (1.1)	91.3 (0.6)
poda variante				
NF	13.3 (0.7)	5.0 (0.0)	11.5 (1.7)	8.8 (1.2)
NN	25.6 (1.4)	9.0 (0.0)	22.0 (3.4)	16.6 (2.4)
H	6.2 (0.0)	4.0 (0.0)	6.3 (0.5)	5.2 (0.4)
CC	92.0 (6.8)	20.0 (0.0)	87.2 (19.6)	51.5 (11.1)
P(%)	89.7 (0.1)	100.0 (0.0)	92.8 (1.5)	90.3 (0.1)

Tabela 7.6: 4-termo 3FND

7.4 Avaliação da Estratégia de Pré-Processamento do Conjunto de Treinamento

Como visto na seção anterior, o *fringe* foi o algoritmo de construção de features que apresentou o melhor desempenho; em contrapartida, o número de iterações realizadas pelo *fringe* é sempre maior. A técnica de pré-processar o conjunto de treinamento foi utilizada com o intuito de minimizar o número de iterações realizadas pelo algoritmo *fringe*.

Nos experimentos realizados, o conjunto de treinamento foi submetido a um pré-processamento e expandido, com a introdução dos implicantes primos da função, considerados como novas features. Após a expansão do conjunto de treinamento, o algoritmo *fringe* foi utilizado. A composição dos algoritmos de pré-processamento e do *fringe* é chamada, neste trabalho, de *quine-fringe*.

Os algoritmos *fringe* e *quine-fringe* foram avaliados no aprendizado de 21 funções booleanas. O conjunto das 21 funções foi dividido em 3 grupos com 7 funções cada, usando como critério de agrupamento o número de atributos relevantes em cada função, a saber: 3, 4 e 5. Ambos algoritmos adotam como critério de parada do processo iterativo de construção de features o fato $V_{k+1} = V_k$ (ver algoritmo na pg. 67).

A Tabela 7.7 apresenta os resultados obtidos com o uso de *fringe* e *quine-fringe* no aprendizado de funções booleanas totais². Nesta tabela, *NI* representa o número de iterações realizadas, *NF* representa a média do número de folhas das árvores obtidas e *CC/N* representa a média dos comprimentos de caminhos das árvores divididos pelos respectivos números de nós. Ela mostra, também, o desvio padrão obtido.

Número de atributos	fringe (F)			quine-fringe (Q-F)		
	NI	NF	CC/N	NI	NF	CC/N
3	4.0(0.75)	4.0(0.75)	1.71(0.38)	2.00(0.00)	3.86(0.83)	1.64(0.42)
4	4.71(1.38)	4.57(0.73)	2.0(0.37)	2.00(0.00)	4.28(0.45)	1.85(0.23)
5	4.14(0.99)	6.0(2.33)	3.05(1.85)	2.00(0.00)	5.43(1.68)	2.35(0.75)

Tabela 7.7: Desempenho de *fringe* e *quine-fringe* no Aprendizado de Funções Completas

Como esperado neste caso particular — em que a função a ser aprendida é completamente especificada — o *quine-fringe* tem um desempenho muito melhor do que *fringe*, uma vez que os implicantes primos determinados pelo algoritmo *quine-maccluskey* são precisamente os implicantes primos da função, e o *fringe*, mesmo tentando, não consegue encontrar *features* mais representativas para a expressão da função. Este fato é evidenciado no número constante e igual a 2 de iterações realizadas pelo *quine-fringe*.

O algoritmo *fringe*, após um determinado número de iterações, também evidencia como features os implicantes primos da função. Os implicantes primos (não aqueles essenciais) usados por ambos algoritmos podem, entretanto, diferir. Uma razão possível é que quando da construção da árvore de decisão, o teste aplicado para determinar o melhor atributo para particionar o conjunto de exemplos, nem sempre encontra apenas um

²Funções especificadas por sua tabela-verdade completa.

atributo, i.e., a função de avaliação de vários atributos é idêntica. Ainda assim, ambos algoritmos constroem a árvore de decisão final usando features que correspondem aos implicantes primos essenciais (quando eles existem).

Em todos os casos, a média do número de folhas e do comprimento de caminho dividido pelo número de nós da árvore é maior para *fringe*.

O próximo teste foi realizado usando conjuntos não totais de exemplos para construir a árvore. Para cada função com 4 e 5 atributos relevantes, o experimento foi repetido 3 vezes (com diferentes conjuntos de exemplos de treinamento). Foram usados 3/4 e 7/8 dos exemplos gerados para construir a árvore e o restante (1/4 e 1/8 respectivamente) para a verificação da precisão da árvore construída.

É importante notar que muito embora os exemplos usados para construir a árvore não cobrissem a FND a ser aprendida, os exemplos reservados para testar a precisão da árvore apenas continham atributos relevantes para o aprendizado da fórmula — isto explica a alta porcentagem de erro obtido. Tal experimento foi realizado objetivando medir a real utilidade de encontrar e usar os implicantes primos, tendo informação incompleta. A Tabela 7.8 e 7.9 mostram os resultados obtidos. Nelas, a porcentagem de erro é obtida pela divisão do número de erros de classificação pelo tamanho da amostra de teste.

Número de atributos: 4	3/4		7/8	
	Q-F	F	Q-F	F
NF	4.62(0.45)	4.14(0.43)	4.62(0.70)	4.24(0.75)
CC/N	2.09(0.24)	1.83(0.23)	2.04(0.29)	1.87(0.40)
% Erro	31(14)	28(6)	47(26)	26(19)
NI	2.19(0.69)	4.24(0.59)	2.14(0.16)	4.38(0.60)

Tabela 7.8: Desempenho de *fringe* e *quine-fringe* no Aprendizado de Funções Booleanas com 4 Atributos, a Partir do Conjunto de Treinamento Incompleto

Número de atributos: 5	3/4		7/8	
	Q-F	F	Q-F	F
NF	5.51(1.22)	5.19(1.46)	5.76(1.11)	5.19(1.59)
CC/N	2.48(0.54)	2.28(0.67)	2.65(0.52)	2.27(0.73)
% Erro	31(14)	21(12)	29(9)	18(2)
NI	2.52(0.35)	4.38(0.58)	2.24(0.13)	4.71(1.0)

Tabela 7.9: Desempenho de *fringe* e *quine-fringe* no Aprendizado de Funções Booleanas com 5 Atributos, a Partir do Conjunto de Treinamento Incompleto

Pode ser observado que, em ambos os casos, ainda que o número de iterações do *quine-fringe* seja bem menor que o do *fringe*, tanto a precisão quanto a complexidade das árvores geradas, apresentam os melhores resultados com o uso do *fringe*.

7.5 Considerações Finais

Muitas vezes a linguagem utilizada para a descrição do conjunto de treinamento não é rica o suficiente para permitir uma descrição concisa do conceito a ser aprendido. Essa situação se torna ainda mais drástica quando o conceito existe na forma de múltiplas regiões disjuntas no espaço de instâncias³.

Além disso, geralmente os métodos de aprendizado indutivo não consideram informações que correlacionam mais do que uma feature. Essa situação torna o aprendizado do conceito disjunto difícil de ser conduzido pelos métodos tradicionais de aprendizado indutivo: a presença de múltiplas regiões disjuntas implica a distribuição da informação através de várias features. Construção de features pode ser adotada como uma solução potencial a este problema; a informação que correlaciona features é combinada formando uma nova feature. Através da introdução de descritores apropriados, construção de features pode melhorar a precisão e concisão da hipótese. Os experimentos conduzidos e descritos neste capítulo evidenciaram esse aspecto de construção de features e comprovaram o bias *fringe* como o que produz as melhores hipóteses.

Dois mecanismos de poda foram experimentados em um ambiente de indução construtiva e, muito embora tenham promovido o nível da precisão das árvores geradas pelo *root* e *root-fringe*, não foram muito efetivos quando usados em árvores geradas pelo *fringe*. Nas situações experimentadas, os dois métodos de poda tiveram desempenho semelhante, sendo que o Variante mostrou-se ligeiramente mais drástico que o da Redução do Erro.

Os experimentos de introdução dos implicantes primos da função, como features, para um posterior processamento pelo *fringe* mostraram que o pré-processamento em questão não foi muito conveniente numa situação típica de aprendizado: aquela em que o conjunto de treinamento está incompleto.

Apesar do elevado custo computacional, existem vários benefícios potenciais no uso de indução construtiva, entre eles: construção de novas features pode permitir uma maior tolerância a ruídos [Pagallo 89a]; viabilização da construção de hipóteses mais concisas e, quando a construção de features é acompanhada de avaliação e poda, o tempo necessário ao aprendizado do conceito pode ser reduzido [Drastal 89]. É importante ressaltar, mais uma vez, que os resultados obtidos no aprendizado de funções booleanas são diretamente aplicáveis a situações reais de aprendizado de conceitos, em domínios onde a informação disponível no conjunto de treinamento assinala apenas a presença ou ausência de atributos.

³Referenciado como conceito disjunto.

Capítulo 8

Programação Lógica Indutiva — PLI

8.1 Considerações Iniciais

Programação Lógica Indutiva é uma abordagem relativamente recente de aprendizado indutivo de máquina, que busca contornar algumas das limitações de abordagens anteriores através do uso

- de teoria do domínio (ou conhecimento de fundo) na construção da expressão do conceito a ser aprendido
- de linguagens de descrição mais poderosas, no caso, a linguagem de programas lógicos

Se por um lado a adoção de linguagens de programas lógicos como linguagens de descrição de instâncias, conceitos e teoria do domínio, viabiliza o aprendizado de um espectro muito mais amplo de conceitos e promove PLI a uma técnica indutiva extremamente poderosa, traz por outro lado, uma série de problemas que ainda estão começando a ser contornados e/ou solucionados.

Tais problemas estão relacionados, principalmente, ao controle da busca pela expressão do conceito, uma vez que o aprendizado de definições lógicas requer a exploração de vastos espaços de descrições de hipóteses. Tais espaços devem, de alguma forma, ser confinados, de maneira a tornar o aprendizado uma tarefa exeqüível. A pesquisa na área de PLI tem investido principalmente em confinar o espaço de busca, impondo restrições às linguagens de descrição utilizadas pelo sistema.

Neste Capítulo são introduzidos a terminologia e os conceitos básicos necessários à compreensão de PLI e de seus problemas. Busca é caracterizada como um problema específico de PLI e algumas estratégias são apresentadas.

8.2 Conceitos Básicos

Definição 8.2.1 (*Cláusula Definida*) Uma cláusula definida é uma cláusula que contém exatamente uma literal positiva. Tem a forma

$$A \leftarrow B_1, B_2, \dots, B_n$$

onde $A, B_1, B_2, \dots, B_n$ são átomos. A literal positiva A é chamada de cabeça e $B_1, \dots, B_n$ é chamado de corpo da cláusula definida.

Apenas átomos são permitidos no corpo de cláusulas definidas. Em Prolog, entretanto, literais da forma *not* L , onde L é um átomo, são permitidas, onde *not* é interpretado como *negação-como-falha*.

Definição 8.2.2 (*Cláusula de Programa*) Uma cláusula de programa é uma cláusula da forma

$$A \leftarrow L_1, L_2, \dots, L_n$$

onde A é um átomo (i.e., uma literal positiva) e cada $L_1, \dots, L_n$ é da forma B ou $\neg B$, onde B é um átomo.

Definição 8.2.3 (*Cláusula Unitária*) Uma cláusula unitária é uma cláusula da forma

$$A \leftarrow$$

ou seja, uma cláusula de programa (ou cláusula definida) com o corpo vazio.

Na terminologia Prolog uma cláusula unitária é chamada de *fato* — e algumas vezes *cláusula definida incondicional* — e é denotada simplesmente por

$$A.$$

Definição 8.2.4 (*Cláusula Meta*) Uma cláusula meta é uma cláusula da forma

$$\leftarrow L_1, \dots, L_m$$

i.e., uma cláusula sem cabeça. Cada L_i ($i = 1, \dots, m$) é chamada de submeta da cláusula meta. É a negação pura de $L_1, \dots, L_m$

Definição 8.2.5 (*Cláusula Tipada*) Uma cláusula é tipada se cada variável que comparece como argumento das literais da cláusula, está associada com um conjunto de valores que pode assumir.

Definição 8.2.6 (*Cláusula de Base de Dados*) Uma cláusula de base de dados é uma cláusula de programa tipada da forma

$$A \leftarrow L_1, \dots, L_n$$

onde A é um átomo e $L_1, \dots, L_n$ são literais.

Definição 8.2.7 (*Fórmula ou Termo Ground*) Seja E uma wff ou um termo. Seja $\text{vars}(E)$ o conjunto de variáveis em E . Diz-se que E é ground se e somente se $\text{vars}(E) = \emptyset$.

Definição 8.2.8 (*Programa Lógico*) Um conjunto finito de cláusulas de programa é chamado de programa lógico.

Definição 8.2.9 Em um programa lógico, o conjunto de todas as cláusulas de programa com o mesmo predicado p na cabeça (e a mesma aridade) é chamado de definição de p .

Definição 8.2.10 (*Cláusula de Horn*) Uma cláusula de Horn é uma cláusula que é uma cláusula de programa ou uma cláusula meta. Isto significa que uma cláusula de Horn contém no máximo uma literal positiva.

8.3 O Processo de Aprendizado em PLI

O aprendizado empírico de um único conceito — *predicado* — em PLI pode ser formulado como [Muggleton 91, Lavrač 94]

Dados

- um conjunto de exemplos de treinamento $\mathcal{E}$ descritos na linguagem $\mathcal{L}_{\mathcal{E}}$ e consistindo de
 - exemplos positivos, $\mathcal{E}^+$
 - exemplos negativos, $\mathcal{E}^-$
- de um predicado desconhecido p — relação a ser aprendida (*relação alvo*)
- uma linguagem de descrição $\mathcal{L}_{\mathcal{H}}$, especificando as restrições sintáticas na definição do predicado p
- teoria do domínio $\mathcal{K}$, descrita na linguagem $\mathcal{L}_{\mathcal{K}}$, definindo predicados q_i (outros que p) os quais podem ser usados na definição de p e que podem fornecer informação adicional sobre argumentos dos exemplos do predicado p
- um operador entre $\mathcal{L}_{\mathcal{E}}$ and $\mathcal{L}_{\mathcal{H}}$ com relação a $\mathcal{L}_{\mathcal{K}}$ que determina se um exemplo é coberto por uma cláusula expressa em $\mathcal{L}_{\mathcal{H}}$

Encontrar

- uma definição $\mathcal{H}$ para p , expressa em $\mathcal{L}_{\mathcal{H}}$, tal que
 - $\mathcal{H}$ é completa, i.e., $\mathcal{K} \wedge \mathcal{H} \models \mathcal{E}^+$,
 - $\mathcal{H}$ é consistente, i.e., $\mathcal{K} \wedge \mathcal{H} \not\models \mathcal{E}^-$

com relação aos exemplos

8.4 Buscas no Espaço de Hipóteses

Como já discutido, aprendizado de conceitos a partir de exemplos pode ser abordado como um processo de busca no espaço de hipóteses, o qual é ordenado por uma relação de generalidade entre elas. O tamanho do espaço e as propriedades da relação de generalização¹ determinam o grau de dificuldade em encontrar a expressão do conceito.

Objetivando lidar com espaços de hipóteses complexos, alguns aspectos representacionais da tarefa de aprendizado devem ser analisados e escolhas devem ser feitas. De acordo com [Kietz 92], escolhas podem ser feitas relacionadas a

- *formalismo de representação* — está relacionado com a linguagem empregada para a representação dos exemplos, hipóteses e teoria do domínio. Entre os formalismos mais usados estão cláusulas de Horn (Prolog puro) e cláusulas de Horn sem símbolos funcionais (programas Datalog)
- *subespaços de busca restritos* — a restrição frequentemente imposta ao formalismo de representação não é suficiente para evitar que problemas de busca continuem ainda muito complexos. Assim, restrições adicionais são necessárias — isto pode ser conseguido através de uma nova restrição no formalismo representacional ou, então, através do uso de operadores de generalização/especialização que direcionam o processo de busca
- *subespaços restritos da teoria do domínio* — a forma como a teoria do domínio é utilizada tem uma importante influência na complexidade da tarefa de aprendizado. A teoria do domínio, geralmente, é fornecida ao sistema na forma *ground*.
- *controle na busca* — está relacionado ao uso de heurísticas que guiam o processo de busca através do espaço de hipóteses. Pode melhorar a eficiência do sistema, em detrimento da completeza

Como já mencionado, a maioria das técnicas disponíveis de aprendizado de conceitos pode ser vista como técnica de busca [Mitchell 82]. É importante lembrar que para uma busca ser efetiva, é fundamental que ela seja conduzida de maneira sistemática. No caso particular de PLI, para sistematizar a busca no espaço de hipóteses, é imprescindível

¹Tal como a θ — *subsoma* [Plotkin 71].

que esse espaço seja estruturado através do estabelecimento de uma ordenação de seus elementos. A estruturação do espaço de hipóteses é obtida através da introdução de uma ordem parcial no conjunto de hipóteses², baseada na operação de θ - subsoma.

Definição 8.4.1 (θ -subsoma) Uma cláusula C_1 é mais geral (θ -subsoma) do que uma cláusula C_2 se e somente se existir uma substituição θ tal que $C_1\theta \subseteq C_2$. O fato de C_1 θ -subsoma C_2 é notado por $C_1 \preceq C_2$.

No contexto da Definição 8.4, uma cláusula é tratada como um conjunto de literais, relacionadas entre si via disjunção. Por exemplo, a cláusula

$$f(X, Y) \leftarrow g(X) \wedge h(X, Y)$$

corresponde ao conjunto

$$\{f(X, Y), \neg g(X), \neg h(X, Y)\}$$

Exemplo 8.4.1 Sejam as cláusulas

$$C_1 : \text{cena}(X) \leftarrow \text{sobre}(X, Y, Z), \text{circulo}(Y)$$

$$C_2 : \text{cena}(\text{quadro1}) \leftarrow \text{sobre}(\text{quadro1}, \text{anel}, E), \text{circulo}(\text{anel}), \text{plano}(E)$$

C_1 θ -subsoma C_2 , com $\theta = \{\text{quadro1}/X, \text{anel}/Y, E/Z\}$.

As seguintes propriedades são válidas para θ -subsoma

1. se C_1 θ -subsoma C_2 então $C_1 \models C_2$. O inverso nem sempre é verdade.
2. a relação $\preceq$ induz um reticulado no conjunto de todas as cláusulas. Isto significa que para quaisquer duas cláusulas existe um *menor limite superior* e um *maior limite inferior* e que ambos são únicos.

A operação de θ -subsoma introduz uma noção sintática de generalidade.

- a cláusula C_1 é pelo menos tão geral quanto a cláusula C_2 ($C_1 \preceq C_2$) se C_1 θ -subsoma C_2
- a cláusula C_1 é mais geral que a cláusula C_2 ($C_1 \prec C_2$) se $C_1 \preceq C_2$ e $C_2 \not\preceq C_1$.

Neste último caso, diz-se que C_2 é uma *especialização* de C_1 e que C_1 é uma *generalização* de C_2 .

²Cláusulas.

Com o espaço de hipóteses estruturado pela introdução de uma ordem parcial, buscas nesse espaço tornam-se factíveis e podem ser, sistematicamente, realizadas de maneira *bottom-up* ou *top-down*.

Técnicas que implementam busca *top-down* são conhecidas como *técnicas de especialização* e consistem basicamente na aplicação de operadores, de maneira a obter, a partir de uma descrição geral do conceito, descrições mais especializadas. Por outro lado, técnicas de generalização buscam o espaço de hipóteses de uma maneira *bottom-up*, inicializando a busca a partir dos exemplos de treinamento — hipóteses mais específicas — usando operadores que as generalizam. A seguir são discutidas algumas das operações mais utilizadas em buscas que utilizam generalizações e especializações, respectivamente.

8.4.1 Através de Técnicas de Generalização

LGG — Menor Generalização Geral

Definição 8.4.2 A menor generalização geral — lgg^3 — de duas cláusulas C_1 e C_2 , denotada por $lgg(C_1, C_2)$, é o menor limite superior de C_1 e C_2 , no reticulado de cláusulas induzido pela relação $\preceq$.

Exemplo 8.4.2 $lgg(p(g(a), a), p(g(b), b)) = p(g(X), X)$

Deve ser observado que com a substituição $\{X/a\}$, $p(g(X), X)$ subsoma $p(g(a), a)$ e que com a substituição $\{X/b\}$, $p(g(X), X)$ subsoma $p(g(b), b)$.

Os próximos três algoritmos [Lavrač 94] calculam, respectivamente, a lgg de termos, lgg de literais e lgg de cláusulas.

Algoritmo 8.4.1 lgg de termos

```

procedure  $lgg(T_1 : termo, T_2 : termo)$ 
  if  $T_1 = T_2$ 
    then  $T_1$ 
  else if  $T_1 = f(u_1, \dots, u_k)$  e  $T_2 = f(v_1, \dots, v_k)$ 
    then  $f(lgg(u_1, v_1), \dots, lgg(u_k, v_k))$ 
    else variável  $V$ , onde a mesma  $V$  é usada
      sempre como a  $lgg$  desses termos
    endif
  endif
endproc

```

³Optamos por manter as iniciais da operação *least general generalization* — lgg — por ser universalmente aceita.

Algoritmo 8.4.2 lgg de literais

```

procedure lgg( $L_1$  : literal,  $L_2$  : literal)
  if  $L_1 = \text{sign } p(t_1, \dots, t_n)$  e  $L_2 = \text{sign } p(u_1, \dots, u_n)$ 
    then  $\text{sign } p(\text{lgg}(t_1, u_1), \dots, \text{lgg}(t_n, u_n))$ 
    else indefinido
  endif
endproc

```

Algoritmo 8.4.3 lgg de cláusulas

```

procedure lgg( $C_1$  : cláusula,  $C_2$  : cláusula)
  if  $C_1 = \{L_1, \dots, L_n\}$  and  $C_2 = \{K_1, \dots, K_m\}$ 
    then  $\{U_i \mid U_i = \text{lgg}(L_j, K_k) \text{ for } 1 \leq j \leq n \text{ and } 1 \leq k \leq m \text{ and } \text{lgg}(L_j, K_k) \neq \text{indefinido}\}$ 
  endif
endproc

```

$Literal_1$	$Literal_2$	$\text{lgg}(Literal_1, Literal_2)$
$g(X, Y).$	$g(Y, X).$	$g(A, B).$
$g(X, Y).$	$g(X, Y).$	$g(A, B).$
$\text{concat}([], M, N).$	$\text{concat}(X, [], N).$	$\text{concat}(A, B, C).$
$\text{concat}([], [N M], Z).$	$\text{concat}([2], [3 Y], [2, 3 Y]).$	$\text{concat}(A, [B C], D).$
$q([a, b, c, d], X, Y).$	$q([a, c, d], X, Z).$	$q([a, A, B C], D, E).$
$q([a, c, d], X, V).$	$q([a, b, c, d, e], X, V).$	$q([a, A, B C], D, E).$
$q([1, 2], [3, 4], [1, 2, 3, 4]).$	$q([a], [b], [a, b]).$	$q([A B], [C D], [A, E F]).$
$q([1, 1, 2, 2, 3],$ $[a, b, a], [1, 1, 2, 2, 3, a, b, a]).$	$q([a, b],$ $[c], [a, b, c]).$	$q([A, B C],$ $[D E], [A, B, F G]).$
$a([a, b], X, X).$	$a([a, c, d], V, V).$	$a([a, A B], C, C).$
$a([a], X, X).$	$a([b, c], V, V).$	$a([A B], C, C).$
$p(\text{tree}(\text{nil}, a, \text{nil})).$	$p(\text{tree}(\text{tree}(\text{nil}, a, \text{nil}), a, \text{nil})).$	$p(\text{tree}(A, a, \text{nil})).$
$p([1, 2, 3]).$	$p([4, 5, 6]).$	$p([A, B, C]).$
$p([1, 2, 3, 4, 5, 6, 7]).$	$p([a]).$	$p([A B]).$
$p([]).$	$p([1, 2, 3, 4, 5, 6, 7, 8]).$	$p(A).$
$p([1, 2, 3]).$	$p([a, b, c, d, e]).$	$p([A, B, C D]).$
$p([1, 1]).$	$p([a, a, b, c, d]).$	$p([A, A B]).$
$p([1, 1, 1, 1]).$	$p([2, 2, 2, 3, 4, 5]).$	$p([A, A, A, B C]).$
$p([2, 2, 2, 3, 4, 5]).$	$p([1, 1, 1, 1]).$	$p([A, A, A, B C]).$
$p(f(X, a), b, c, g(a, b, M)).$	$p(f(X, e), b, d, g(b, a, M)).$	$p(f(A, B), b, C, g(D, E, F)).$
$p(f(X, a), b, c, g(a, b, M)).$	$p(f(a, X), e, c, g(X, e, N)).$	$p(f(A, B), C, c, g(B, C, D)).$
$p(A, b, c, D, f(g, h)).$	$p(A, b, v, F, p).$	$p(A, b, B, C, D).$
$p(X, Y, Z, p(X, Y, Z)).$	$p(X, Y, Z, p(X, Y, a)).$	$p(A, B, C, p(A, B, D)).$
$p(X, X, Y, p(X, X, Y, a)).$	$p(a, b, m, p(b, a, n, X)).$	$p(A, B, C, p(B, A, D, E)).$
$p([], [], M, [N I], [1, 2, 3, 4 K],$ $g(M, N, K)).$	$p([], a, j, i, [1, 2, 4, 3, 7 K],$ $g(j, 1, u)).$	$p([], A, B, C, [1, 2, D, E F],$ $g(B, G, H)).$
$p([1, 2], [3, 4], [1, 2, 3, 4]).$	$p([a], [b], [a, b]).$	$p([A B], [C D], [A, E F]).$
$p(X, Y, Z).$	$p(p(X, Y, Z), Y, Y).$	$p(A, B, C).$
$p(h(a, b, v, c), h(5, 8), M).$	$p(h(x, y), h(5, 8, 7, 6), B).$	$p(A, B, C).$
$p([], [], [], []).$	$p(a, b, a, b).$	$p(A, B, A, B).$
$p(1, 2, 3, 4).$	$p([1, 2, 3, 4], [], 3, [M N]).$	$p(A, B, 3, C).$
$p(1, 2, 3).$	$p(2, 2, 2, 2).$	$[Incompatible predicate symbols]$

Tabela 8.1: Lggs entre Pares de Literais Calculadas pelo Sistema *GOLEM*

A operação de lgg, como definida anteriormente, é uma operação puramente sintática. A utilização da teoria do domínio expressa extensionalmente na operação lgg introduz

a noção semântica na generalização e dá origem à operação de *menor generalização geral relativa* — *rlgg*⁴

A determinação da *rlgg* é abordada mais detalhadamente no contexto do sistema *GOLEM* — ver Capítulo 11, pg. 146 — que a utiliza para direcionar a busca bottom-up que realiza. Resumidamente, se a teoria do domínio consiste de fatos ground e $\mathcal{K}$ denota a conjunção de todos esses fatos, a *rlgg* de dois átomos ground, A_1 e A_2 , relativa à teoria do domínio ground $\mathcal{K}$ é definida como

Definição 8.4.3 *A menor generalização geral relativa — rlgg — de dois exemplos A_1 e A_2 é a sua menor generalização geral $lgg(A_1, A_2)$ relativa à teoria do domínio $\mathcal{K}$, expressa por*

$$rlgg(A_1, A_2) = lgg((A_1 \leftarrow \mathcal{K}), (A_2 \leftarrow \mathcal{K}))$$

Modelos de uma teoria nem sempre são finitos. Se a teoria contém cláusulas arbitrárias, o modelo a ela associado pode ser infinito (por exemplo, se a teoria contém cláusulas recursivas). Modelos infinitos resultam em cláusulas infinitas e, conseqüentemente, em uma *rlgg* infinita entre essas cláusulas. Em sistemas que usam a teoria do domínio em forma extensional, a restrição do modelo associado à teoria do domínio é vital para a exeqüibilidade do aprendizado. Formas de restrição do modelo associado à teoria do domínio são tratadas mais detalhadamente nos Capítulos 9 e 11.

Inversão de Resolução

A inversão da resolução foi introduzida como uma técnica de generalização em PLI por Muggleton e Buntine [Muggleton 88], e consiste na inversão da regra de resolução da inferência dedutiva [Robinson 65].

O sistema *CIGOL* [Muggleton 92a], que emprega a técnica de inversão de resolução na generalização de suas hipóteses, inverte o processo de resolução usando o operador de generalização baseado na inversão da substituição [Buntine 88]. Dada uma wff W , uma *substituição inversa* θ^{-1} da substituição θ é uma função que associa termos de $W\theta$ a variáveis, tal que $W\theta\theta^{-1} = W$.

O sistema *CIGOL* processa exemplos incrementalmente e depende de um oráculo para guiar o processo de indução. *CIGOL* é uma extensão do sistema *DUCE* [Muggleton 87], cujo algoritmo é não incremental, é restrito à geração de cláusulas de Horn proposicionais e não cria novos descritores.

CIGOL é um algoritmo relativamente ineficiente porque existem muitos resolventes em cada passo da inversão da resolução; isso, na maioria das vezes, implica um número muito grande de passos necessários para a construção da descrição do conceito a ser aprendido. Tentativas de melhoria do *CIGOL* resultaram no sistema *IRES* [Rouveirol 90] (e seu descendente *ITOU* [Rouveirol 91]), ainda dependentes de um oráculo, de maneira a garantir a corretividade, quando da busca em imensos espaços.

⁴Optamos por manter as iniciais da operação *least general generalization* — *rlgg* — por ser universalmente aceita.

A maioria dos algoritmos que usa a inversão de resolução permite a ampliação da linguagem de descrição de conceitos, através da construção de novos descritores — predicados. Eles são, entretanto, limitados ou pela dedução, ou por requererem um oráculo para manter a eficiência a um nível razoável [Aha 92]. Algoritmos que usam inversão de resolução não são abordados neste trabalho.

8.4.2 Através de Técnicas de Especialização

Técnicas de especialização são usadas para a realização de busca top-down no espaço de hipóteses (cláusulas). A busca se inicia a partir de hipóteses gerais, e se direciona a hipóteses mais específicas, através da aplicação de operadores de refinamento.

Um operador de refinamento, quando aplicado a uma cláusula, determina tipicamente apenas o conjunto de especializações minimais (mais gerais) dessa cláusula, com relação à θ -subsoma. Uma operação de refinamento se traduz sintaticamente na

1. aplicação de uma substituição à cláusula e/ou
2. adição de uma literal ao corpo da cláusula

A técnica básica de especialização em PLI é a *busca top-down de grafos de refinamento*. Como mencionado anteriormente, o espaço de hipóteses pode ser associado a um reticulado, estruturado através da ordenação de generalidade entre cláusulas, como decorrência da aplicação de θ -subsoma. Neste reticulado, o grafo de refinamento pode ser definido e usado para direcionar a busca de hipóteses mais específicas, a partir das mais gerais.

Sistemas que utilizam essa técnica inicializam o aprendizado a partir das cláusulas mais gerais possíveis e, repetidamente, especializam-nas até que não cubram mais exemplos negativos; durante a busca os sistemas garantem que as cláusulas consideradas cobrem, pelo menos, um exemplo positivo [Lavrač 94].

8.5 Considerações Finais

Como discutido nos capítulos anteriores, a utilização de linguagens proposicionais para a representação de exemplos e conceitos, têm uma série de limitações, entre elas, a dificuldade no uso de teoria do domínio e a impossibilidade do aprendizado de conceitos relacionais.

Neste capítulo, a fim de contornar aquelas limitações, é discutido a ampliação do poder representacional de sistemas de aprendizado, através da adoção de linguagens mais poderosas, baseadas em lógica de primeira ordem. Programação Lógica Indutiva combina técnicas de programação lógica, com processos indutivos e viabiliza o aprendizado indutivo de conceitos relacionais — aprendidos e expressos como programas lógicos — a partir de exemplos de treinamento e teoria do domínio. Por um lado o uso de linguagens mais ricas, no caso, subconjuntos da lógica de primeira ordem, bem como a

possibilidade de incorporação da teoria do domínio no processo de aprendizado, ampliam os limites do espaço de conceitos passíveis de serem aprendidos. Por outro lado, dão origem a uma série de problemas que necessitam ser tratados, de maneira a viabilizar o aprendizado. A maioria dos problemas no uso de PLI está relacionada à busca no espaço de possíveis hipóteses: como controlar, direcionar e garantir o término de tal busca. A relação de θ — *subsoma* foi introduzida como uma ordenação parcial no espaço de hipóteses — apenas com o espaço estruturado é possível realizar uma busca sistemática. Com o espaço estruturado, vários operadores podem ser aplicados, de maneira a direcionar a busca — este capítulo abordou alguns dos mais utilizados.

Capítulo 9

A Teoria do Domínio em PLI

9.1 Considerações Iniciais

Como já comentado anteriormente, a adoção de linguagens de descrição mais ricas, bem como o uso da teoria do domínio, muito embora ampliem os limites do espaço de hipóteses, dão origem a muitas dificuldades que necessitam ser contornadas, para que o aprendizado se torne exeqüível.

Este capítulo discute o uso e as restrições impostas à teoria do domínio, em um ambiente de Programação Lógica Indutiva. Os dois sistemas PLI mais relevantes — *GOLEM* e *FOIL*, tratados no Capítulo 11, pg. 146 — são operacionais apenas quando a teoria do domínio está expressa extensionalmente.

O uso de cláusula generativa, uma forma restrita de cláusula de Horn, que permite a construção de um modelo finito associado a um programa, é discutido em detalhes. Um algoritmo que verifica se uma dada teoria do domínio consiste apenas de cláusulas generativas, é por nós proposto assim como sua implementação apresentada.

A “tradução” da teoria do domínio, da forma intencional para extensional, é abordada através de dois algoritmos, por nós propostos. Os algoritmos diferem entre si apenas com relação à estratégia adotada. Enquanto o primeiro constrói o modelo associado à teoria do domínio usando uma estratégia de encadeamento para frente, o segundo constrói o modelo usando uma estratégia de encadeamento para trás. A implementação do segundo algoritmo é apresentada e discutida.

9.2 Restringindo a Teoria do Domínio

A teoria do domínio pode ser expressa em forma *intencional* ou *extensional*. Ela é descrita extensionalmente através da identificação de todas as suas instâncias. O conceito de *par* em um jogo de pôquer, por exemplo, pode ser extensionalmente descrito pelo conjunto de todos os pares de cartas que têm o mesmo valor, independentemente de qual seja o seu naipe, ou seja, $par = \{ \langle espada, 7, copas, 7 \rangle, \langle espada, 7, ouros, 7 \rangle, \dots \}$.

$\langle \text{espada}, 7, \text{paus}, 7 \rangle, \dots, \langle \text{copas}, \text{rei}, \text{espada}, \text{rei} \rangle$. Descrições extensionais podem ser inconvenientes por uma série de razões, sendo a principal delas a possibilidade de permitir um número infinito de instâncias. Em muitas situações, conseqüentemente, é preferível apresentar a descrição em sua forma intencional, isto é, em uma linguagem de descrição que permita representações mais compactas e concisas, por exemplo, na forma de regras de decisão.

A teoria do domínio — usada por sistemas PLI — deve sempre ser restrita antes de sua incorporação ao processo de aprendizado, caso contrário, a busca pela expressão do conceito herda a indecidibilidade¹ do processo dedutivo.

A maioria dos sistemas PLI exige que a teoria do domínio esteja expressa extensionalmente. O número de instâncias que a descrevem deve, conseqüentemente, ser limitado. No que se segue, serão apresentadas e discutidas maneiras de restringir a teoria do domínio.

9.2.1 Conceitos Básicos

A terminologia utilizada nesse capítulo segue a definição de aprendizado de um único conceito em PLI, apresentada na pg. 100.

Definição 9.2.1 (*Teoria Ground do Domínio*) Uma teoria de domínio K é ground se consistir apenas de cláusulas unitárias ground.

Definição 9.2.2 (*Teoria do Domínio com Aridade Limitada*) Uma teoria do domínio K tem aridade limitada, se a aridade máxima dos predicados em K for limitada por alguma constante j .

Definição 9.2.3 (*Cláusula Restrita*) Uma cláusula é restrita se todas as variáveis que comparecem no corpo da cláusula também ocorrem na sua cabeça.

Uma restrição comum, que se aplica tanto à linguagem $\mathcal{L}_H$, de descrição de hipóteses, quanto à linguagem $\mathcal{L}_K$, de descrição da teoria do domínio, é a de permitir o uso apenas de cláusulas livres de símbolos funcionais.

Definição 9.2.4 (*Cláusulas Livres de Símbolos Funcionais*) Uma cláusula é livre de símbolos funcionais se nela não comparecerem símbolos funcionais, i.e., todos os seus argumentos são variáveis ou constantes (símbolos funcionais de aridade 0).

A restrição da expressão da teoria do domínio a cláusulas unitárias, livres de símbolos funcionais, (bem como a restrição dos exemplos a cláusulas ground livres de símbolos funcionais) permite que a operação de θ — *subsoma* (ver Definição 8.4.1, pg. 104)

¹Um sistema lógico é considerado *decidível* se sempre existir um procedimento que pode ser usado para determinar se uma dada fórmula é válida ou não.

[Plotkin 71], possa ser usada como um procedimento completo de inferência. Além disso, se a teoria do domínio $\mathcal{K}$ consistir apenas de cláusulas de Horn generativas livres de símbolos funcionais, com um valor máximo de aridade de seus predicados fixo, a inferência de uma teoria ground do domínio pode ser feita em tempo polinomial, proporcional ao tamanho da teoria do domínio $\mathcal{K}$ [Wrobel 87, Kietz 93].

Os principais sistemas que implementam programação lógica indutiva, *GOLEM* e *FOIL*, assumem que a teoria do domínio está em sua forma extensional, expressa como fatos ground. Se a teoria do domínio estiver expressa intencionalmente, ela deve ser convertida em sua forma extensional, para poder ser incorporada ao processo de aprendizado conduzido por esses sistemas.

9.2.2 O Modelo H-easy

Se o programa lógico P , que expressa a teoria do domínio $\mathcal{K}$, consistir de cláusulas arbitrárias — teoria do domínio expressa intencionalmente — o modelo M de P pode ser infinito. A fim de tratar situações como esta, visando restringir modelos infinitos a finitos, a noção de *h-easiness* foi estabelecida e é apresentada a seguir [Muggleton 90].

Definição 9.2.5 (*Átomo h-easy*) Dado um programa lógico P , um átomo a é *h-easy* com relação a P , para um dado número natural h , se e somente se existir uma derivação de a a partir de P , envolvendo no máximo h resoluções binárias².

Definição 9.2.6 (*Modelo H-easy de Herbrand de P*) O modelo *h-easy* de Herbrand de P — M_h — é o conjunto de todas as instanciações de Herbrand³ de átomos *h-easy* de P .

Teorema 9.2.1 Dado um programa lógico P , para qualquer número finito h , o número de átomos *h-easy* de P é finito.

Prova: Segue do fato que existe apenas um número finito de tais derivações.

Exemplo 9.2.1 Para um dado programa não ground P , representando uma certa teoria do domínio $\mathcal{K}$, o cálculo de M_h para $h = 5$ é mostrado a seguir. Cláusulas factuais do programa estão mostradas pictoricamente na Figura 9.1.

Seja P o programa Prolog a seguir, onde o predicado *pais*(X, Y) representa a relação de X ser pai ou mãe de Y e *predessor*(X, Y) representa a relação de X ser um ancestral de Y .

²Deve ser observado que esta definição implica que fatos constantes do programa lógico P são 1-easy com relação a P , uma vez que 1 resolução binária foi necessária para derivar cada um deles.

³Ver [Nicoletti 93, Nicoletti 93a].

Figura 9.1: Relação de Família

- (1) *pais(beth, tim).*
- (2) *pais(fred, tim).*
- (3) *pais(jackie, vicky).*
- (4) *pais(tim, vicky).*
- (5) *pais(vicky, anna).*
- (6) *pais(vicky, rachel).*
- (7) *pais(martin, anna).*
- (8) *pais(martin, rachel).*
- (9) *pais(rachel, vera).*
- (10) *pais(rachel, sean).*
- (11) *pais(ed, vera).*
- (12) *pais(ed, sean).*
- (13) *predecessor(X, Z) :- pais(X, Z).*
- (14) *predecessor(X, Z) :- pais(X, Y),
predecessor(Y, Z).*

O predicado *predecessor/2* é expresso na forma clausal por

- (13) *predecessor(X, Z) ∨ ¬pais(X, Z)*
- (14) *predecessor(X, Z) ∨ ¬pais(X, Y) ∨ ¬predecessor(Y, Z)*

O conjunto dos átomos 1-easy com relação a P coincide com o Modelo de Herbrand 1-easy de P — M_1 . Isso se deve ao fato de nenhuma instanciação de átomos 1-easy de P ter sido realizada. Todos os átomos 1-easy obtidos foram ground.

$$M_1 = \{ \text{pais(beth, tim), pais(fred, tim), pais(jackie, vicky),} \\ \text{pais(tim, vicky), pais(vicky, anna), pais(vicky, rachel),} \\ \text{pais(martin, anna), pais(martin, rachel), pais(rachel, vera),} \\ \text{pais(rachel, sean), pais(ed, vera), pais(ed, sean)} \}$$

M_2 é o conjunto de todas as instanciações de Herbrand de átomos 2-easy de P . Deve ser lembrado que um átomo a é 2-easy com relação a P , se e somente se existir uma derivação de a , a partir de P , envolvendo no máximo duas resoluções binárias. A

próxima tabela mostra todas as possíveis derivações do programa P , envolvendo 2 resoluções binárias.

<i>Clausula</i> ₁	<i>Clausula</i> ₂	Resolvente (2 resoluções binárias)	
(1)	(13)	<i>predecessor</i> (beth, tim)	(15)
(2)	(13)	<i>predecessor</i> (fred, tim)	(16)
(3)	(13)	<i>predecessor</i> (jackie, vicky)	(17)
(4)	(13)	<i>predecessor</i> (tim, vicky)	(18)
(5)	(13)	<i>predecessor</i> (vicky, anna)	(19)
(6)	(13)	<i>predecessor</i> (vicky, rachel)	(20)
(7)	(13)	<i>predecessor</i> (martin, anna)	(21)
(8)	(13)	<i>predecessor</i> (martin, rachel)	(22)
(9)	(13)	<i>predecessor</i> (rachel, vera)	(23)
(10)	(13)	<i>predecessor</i> (rachel, sean)	(24)
(11)	(13)	<i>predecessor</i> (ed, vera)	(25)
(12)	(13)	<i>predecessor</i> (ed, sean)	(26)
(1)	(14)	<i>predecessor</i> (beth, Z) ∨ ¬ <i>predecessor</i> (tim, Z)	(27)
(2)	(14)	<i>predecessor</i> (fred, Z) ∨ ¬ <i>predecessor</i> (tim, Z)	(28)
(3)	(14)	<i>predecessor</i> (jackie, Z) ∨ ¬ <i>predecessor</i> (vicky, Z)	(29)
(4)	(14)	<i>predecessor</i> (tim, Z) ∨ ¬ <i>predecessor</i> (vicky, Z)	(30)
(5)	(14)	<i>predecessor</i> (vicky, Z) ∨ ¬ <i>predecessor</i> (anna, Z)	(31)
(6)	(14)	<i>predecessor</i> (vicky, Z) ∨ ¬ <i>predecessor</i> (rachel, Z)	(32)
(7)	(14)	<i>predecessor</i> (martin, Z) ∨ ¬ <i>predecessor</i> (anna, Z)	(33)
(8)	(14)	<i>predecessor</i> (martin, Z) ∨ ¬ <i>predecessor</i> (rachel, Z)	(34)
(9)	(14)	<i>predecessor</i> (rachel, Z) ∨ ¬ <i>predecessor</i> (vera, Z)	(35)
(10)	(14)	<i>predecessor</i> (rachel, Z) ∨ ¬ <i>predecessor</i> (sean, Z)	(36)
(11)	(14)	<i>predecessor</i> (ed, Z) ∨ ¬ <i>predecessor</i> (vera, Z)	(37)
(12)	(14)	<i>predecessor</i> (ed, Z) ∨ ¬ <i>predecessor</i> (sean, Z)	(38)

Pode ser visto que apenas as 12 primeiras derivações, envolvendo 2 resoluções binárias (numeradas de 15 a 26), derivam um átomo (todos eles são átomos ground neste exemplo). O conjunto dos átomos 2-easy com relação a P (que inclui os átomos 1-easy também) coincide com o Modelo de Herbrand 2-easy de $P - M_2$.

$$M_2 = M_1 \cup \{predecessor(beth, tim), predecessor(fred, tim), predecessor(jackie, vicky), predecessor(tim, vicky), predecessor(vicky, anna), predecessor(vicky, rachel), predecessor(martin, anna), predecessor(martin, rachel), predecessor(rachel, vera), predecessor(rachel, sean), predecessor(ed, vera), predecessor(ed, sean)\}$$

As próximas tabelas mostram a construção dos conjuntos de átomos 3-easy, 4-easy e 5-easy com relação a P , os quais coincidem com seus respectivos modelos h-easy M_3 , M_4 and M_5 (a coincidência se deve, novamente, ao fato dos conjuntos de átomos h-easy serem ground, o que previne que a instanciação de Herbrand aconteça na geração do M_h).

<i>Clausula</i> ₁	<i>Clausula</i> ₂	Resolvente(3 resoluções binárias)	
(18)	(27)	<i>predecessor</i> (beth, vicky)	(39)
(18)	(28)	<i>predecessor</i> (fred, vicky)	(40)
(19)	(29)	<i>predecessor</i> (jackie, anna)	(41)
(20)	(29)	<i>predecessor</i> (jackie, rachel)	(42)
(19)	(30)	<i>predecessor</i> (tim, anna)	(43)
(20)	(30)	<i>predecessor</i> (tim, rachel)	(44)
(23)	(32)	<i>predecessor</i> (vicky, vera)	(45)
(24)	(32)	<i>predecessor</i> (vicky, sean)	(46)
(23)	(27)	<i>predecessor</i> (martin, vera)	(47)
(24)	(27)	<i>predecessor</i> (martin, sean)	(48)

$$M_3 = M_2 \cup \{predecessor(beth, vicky), predecessor(fred, vicky), predecessor(jackie, anna), predecessor(jackie, rachel), predecessor(tim, anna), predecessor(tim, rachel), predecessor(vicky, vera), predecessor(vicky, sean), predecessor(martin, vera), predecessor(martin, sean)\}$$

<i>Clausula</i> ₁	<i>Clausula</i> ₂	Resolvente (4 resoluções binárias)	
(27)	(43)	<i>predecessor(beth, anna)</i>	(49)
(27)	(44)	<i>predecessor(beth, rachel)</i>	(50)
(28)	(43)	<i>predecessor(fred, anna)</i>	(51)
(28)	(44)	<i>predecessor(fred, rachel)</i>	(52)
(29)	(45)	<i>predecessor(jackie, vera)</i>	(53)
(29)	(46)	<i>predecessor(jackie, sean)</i>	(54)
(30)	(45)	<i>predecessor(tim, vera)</i>	(55)
(30)	(45)	<i>predecessor(tim, sean)</i>	(56)

$$M_4 = M_3 \cup \{predecessor(beth, anna), predecessor(beth, rachel), predecessor(fred, anna), predecessor(fred, rachel), predecessor(jackie, vera), predecessor(jackie, sean), predecessor(tim, vera), predecessor(tim, sean)\}$$

<i>Clausula</i> ₁	<i>Clausula</i> ₂	Resolvente (5 resoluções binárias)	
(27)	(55)	<i>predecessor(beth, vera)</i>	(57)
(27)	(56)	<i>predecessor(beth, sean)</i>	(58)
(28)	(41)	<i>predecessor(fred, vera)</i>	(59)
(28)	(42)	<i>predecessor(fred, sean)</i>	(60)

$$M_5 = M_4 \cup \{predecessor(beth, vera), predecessor(beth, sean), predecessor(fred, vera), predecessor(fred, sean)\}$$

Exemplo 9.2.2 *Seja P o programa Prolog*

$$\begin{aligned} p(X) : - & \quad q(X). \\ & \quad q(Z). \end{aligned}$$

Este programa tem como Universo e Base de Herbrand, respectivamente, os conjuntos

$$\begin{aligned} H &= \{a\} \\ H_B &= \{p(a), q(a)\} \end{aligned}$$

e os seguintes conjuntos

- conjunto de átomos 1-easy de P : $\{q(Z)\}$
- conjunto de todas as instanciações de Herbrand de átomos 1-easy de P

$$M_1 = \{q(a)\}$$

- conjunto de átomos 2-easy de P : $\{p(Z), q(Z)\}$
- conjunto de todas as instanciações de Herbrand de átomos 2-easy de P

$$M_2 = \{p(a), q(a)\}$$

Exemplo 9.2.3 *Seja P o programa Prolog*

$$\begin{array}{l} p(X) : - \quad q(f(X)). \\ q(Z). \end{array}$$

Este programa tem como Universo e Base de Herbrand, respectivamente, os conjuntos

$$\begin{array}{lcl} H & = & \{a, f(a), f(f(a)), f(f(f(a))), \dots\} \\ H_B & = & \{a, p(a), q(a), p(f(a)), q(f(a)), p(f(f(a))), q(f(f(a))), \dots\} \end{array}$$

e os seguintes conjuntos

- conjunto de átomos 1-easy de P : $\{q(Z)\}$
- conjunto de todas as instanciações de Herbrand de átomos 1-easy de P

$$M_1 = \{q(a), q(f(a)), q(f(f(a))), \dots\}$$

- conjunto de átomos 2-easy de P : $\{p(Z), q(Z)\}$
- conjunto de todas as instanciações de Herbrand de átomos 2-easy de P

$$M_2 = \{p(a), q(a), p(f(a)), q(f(a)), \dots\}$$

O Exemplo 9.2.3 ilustra o fato de um programa P , que tem um conjunto finito de átomos h -easy, ter o seu correspondente Modelo de Herbrand M_h infinito.

Desde que existe um interesse especial na geração de modelos finitos de programas, a seguir será definida a noção de cláusulas generativas [Muggleton 90], uma forma restrita de cláusulas de Horn. Se um programa for expresso como um conjunto de tais cláusulas, existe a garantia da existência de um modelo finito a ele associado. Em outras palavras, se a teoria do domínio for expressa como cláusulas generativas, é possível construir um modelo finito a ela associado.

9.2.3 Cláusulas Generativas

Segue a definição de uma forma restrita de cláusula de Horn P para a qual o Modelo de Herbrand h -easy é finito.

Definição 9.2.7 (*N-Derivação-Atômica*)(*Fechamento-da-Derivação-Atômica*) O conjunto n -derivação-atômica do programa P , $D^n(P)$, é definido recursivamente como⁴

⁴Observe que com as definições aqui apresentadas, $M_h = D^{h-1}(P)$.

- $D^0(P) =$ conjunto de cláusulas unitárias de P
- $D^n(P) = D^{n-1}(P) \cup \{A\theta_1 \dots \theta_n : A \leftarrow B_1, \dots, B_m \in P \text{ e para cada } B_i \exists B'_i \in D^{n-1}(P) \text{ tal que } \theta_i \text{ é o mgu}^5 \text{ de } B_i \text{ e } B'_i\}$

O fechamento-da-derivação-atômica é definido como

$$D^*(P) = D^0(P) \cup D^1(P) \cup \dots$$

Definição 9.2.8 (Programa Semanticamente Generativo) O programa lógico P é semanticamente generativo se e somente se todo elemento de $D^*(P)$ for ground.

Esta definição semântica corresponde à restrição sintática em programas lógicos, definida por

Definição 9.2.9 (Cláusula Sintaticamente Generativa) A cláusula $A \leftarrow B_1, \dots, B_m$ é sintaticamente generativa⁶ sempre que as variáveis que aparecem na cabeça A forem um subconjunto das variáveis que aparecem no corpo $B_1, \dots, B_m$. O programa lógico P é sintaticamente generativo⁶ se e somente se toda cláusula em P for sintaticamente generativa.

Teorema 9.2.2 Todo programa lógico sintaticamente generativo P é semanticamente generativo.

Prova:

- Para $n = 0$, $D^0(P)$ contém as cláusulas unitárias de P . Essas cláusulas unitárias são ground uma vez que o programa lógico P é sintaticamente generativo
- Suponha que todo elemento de $D^{n-1}(P)$ seja ground
- Seja $A \leftarrow B_1, \dots, B_m$ uma cláusula em P e $A\theta_1\theta_2 \dots \theta_m$ um elemento de $D^n(P) - D^{n-1}(P)$ tal que, para cada B_i , existe $B'_i \in D^{n-1}(P)$, para o qual θ_i é o mgu de B_i e B'_i .

Pela hipótese indutiva cada $B'_i \in D^{n-1}(P)$ é ground e, assim, cada $B_i\theta_i$ é ground.

Desde que cada θ_i é uma substituição ground e toda variável em A é encontrada no domínio de $\theta_1\theta_2 \dots \theta_m$, segue que $A\theta_1\theta_2 \dots \theta_m$ é ground, o que completa a prova

É importante notar que o Menor Modelo de Herbrand e o fechamento-da-derivação-atômica de um programa lógico semanticamente generativo são iguais. O exemplo que segue ilustra este fato [Hogger 90].

⁵most general unifier (unificador mais geral).

⁶Ou *range-restricted*.

Exemplo 9.2.4 *Seja o conjunto de cláusulas P*

gosta(chris, X) if gosta(X, logica)
gosta(bob, logica)

O Universo de Herbrand para P é o conjunto $H = \{chris, bob, logica\}$ e a instanciação ground de $P - G_P -$ é

gosta(chris, chris) if gosta(chris, logica)
gosta(chris, bob) if gosta(bob, logica)
gosta(chris, logica) if gosta(logica, logica)
gosta(bob, logica)

que pode ser reescrito simplifcadamente como

CC if CL
CB if BL
CL if LL
BL

A Base de Herbrand é

$$H_B = \{CC, CL, CB, LC, LL, LB, BC, BL, BB\}$$

O conjunto $\{BL, CB\} \subset B_P$ é assim uma interpretação de Herbrand. Para essa interpretação a instanciação ground G_P é reproduzida, a seguir, com os valores-verdade associados escritos em lugar de seus átomos.

falso if falso
verdade if verdade
falso if falso
verdade

Com esta atribuição, todas as cláusulas de G_P são *verdade*. Esta interpretação é, portanto, um Modelo de Herbrand para G_P e, portanto, um Modelo de Herbrand para P . Além disso, é o Menor Modelo de Herbrand de P . Este último programa é um programa lógico semanticamente generativo porque é um programa lógico sintaticamente generativo. Seu fechamento-da-derivação-atômica é

$$D^*(P) = D^0(P) \cup D^1(P) = \{BL\} \cup \{CB\} = \{BL, CB\}$$

Teorema 9.2.3 *Para todo programa lógico sintaticamente generativo P , e todo h , $M_h(P)$ é finito.*

Corolario 9.2.1 *Para todo programa lógico sintaticamente generativo P e conjunto de exemplos $e_1, \dots, e_n$, a $rlgg^7$ de $e_1, \dots, e_n$ com relação a $M_h(P)$ é finita e tem tamanho no máximo $|M_h(P)|^n + 1$.*

⁷Ver Definição 8.4.3, pg. 107.

9.3 Transformação de Teoria de Domínio Intencional em Extensional

Como estabelecido na Definição 8.2.9, pg. 102, a definição de um predicado p é o conjunto de todas as cláusulas com o mesmo predicado na cabeça (e mesma aridade).

Como já mencionado, a única maneira de usar uma teoria de domínio P — especificada intencionalmente — tanto em *GOLEM* quanto em *FOIL*, é através do uso de uma versão extensional desta teoria. Esta versão pode ser obtida através da derivação dos átomos que são relevantes aos exemplos, como um pré-processamento da tarefa de aprendizado.

Após ter verificado que uma dada teoria de domínio — expressa como um programa lógico P — consiste apenas de cláusulas generativas (ver Seção 9.2.9, pg. 117), é possível construir o modelo finito associado a essa teoria, através da construção do fechamento-da-derivação-atômica. Para isso, é necessário um algoritmo que encontre todas as possíveis conclusões do programa P . Isto pode ser feito de duas maneiras distintas. Uma delas, é através de um raciocínio que parte dos fatos e se direciona às metas, conhecido como *raciocínio com encadeamento para frente* ou *raciocínio direcionado por dados* ou, ainda, *raciocínio modus ponens*. Por exemplo, dado o programa P

$$\begin{aligned} p(X, Y) : & -q(X), p(Y). \\ q(a). \\ p(b). \end{aligned}$$

pode ser concluído por *modus ponens* que $p(a, b)$ é também um fato.

A outra maneira de condução do raciocínio é partir das metas, direcionando-se aos fatos, conhecida como *raciocínio com encadeamento para trás* ou *raciocínio direcionado por metas*.

Em geral, para usar *encadeamento para frente* — como base da estrutura de controle — é necessário considerar os fatos em uma determinada ordem e, para cada fato F , é necessário encontrar todas as cláusulas

$$A_i \leftarrow L_{i_1}, L_{i_2}, \dots, L_{i_{n_i}}.$$

em que alguma de suas literais L_{i_j} possa ser instanciada com o fato F . Para essas cláusulas, uma nova cláusula é gerada a partir da antiga, mas sem o fato em questão. Numa cláusula cujo corpo é constituído por apenas uma literal, sempre que um fato instancia com essa literal, a cabeça da cláusula é provada e, conseqüentemente, um novo fato é obtido. Um aspecto importante a ser considerado é a ordem na qual fatos estão dispostos. Novos fatos podem ser adicionados à frente daqueles ainda não considerados, de maneira que o último fato provado será o próximo a ser considerado. Embora essa abordagem seja conveniente quando o objetivo for chegar a uma conclusão o mais rápido possível, ela não é apropriada quando for necessário encontrar, sistematicamente, toda conclusão possível a partir dos fatos. Para este último caso, é recomendado colocar os

últimos fatos provados no final do conjunto de fatos. Tal lista de fatos assemelha-se a uma fila.

Um outro aspecto a ser considerado é a presença de átomos negados no corpo da cláusula. Os algoritmos apresentados a seguir [Nicoletti 94], adotam a *hipótese do mundo-fechado*⁸, de maneira que $\neg L$ (negação de L) significa que L não pode ser provado. Devido a isto, o algoritmo de encadeamento para frente assume primeiro que todas as negações são falsas, prova todos os possíveis fatos e apenas considera como verdade aquelas negações que ainda não são fatos. Deve ser observado que aquelas negações — sendo verdadeiras — podem, possivelmente, provar novos fatos, com novas consequências.

9.3.1 Algoritmo para Verificar se um Programa Lógico é Generativo

O algoritmo descrito a seguir tem por entrada um programa P , expresso como um conjunto de cláusulas e verifica se cada uma delas é generativa. O fato de uma das cláusulas não ser generativa implica o conjunto de cláusulas não ser generativo e, conseqüentemente, não haver necessidade do exame das cláusulas restantes.

Seja um programa lógico P definido como um conjunto de cláusulas $\{C_1, \dots, C_m\}$ da forma

$$C_i : A_i \leftarrow L_{i_1}, L_{i_2}, \dots, L_{i_{n_i}} \quad \forall i = 1, \dots, m \text{ e } n_i \geq 0$$

Seja $\text{variaveis}(L_{ij})$ o conjunto das variáveis que ocorrem na literal L_{ij} . Ao término da execução do procedimento

```
function generativo( $P$ ): boolean
  generativo  $\leftarrow$  verdade
  begin
    for  $k = 1$  to  $m$  do
      begin
         $B = \emptyset$ 
         $H = \text{variaveis}(A_k)$ 
        for  $p = 1$  to  $n_k$  do
           $B = B \cup \text{variaveis}(L_{kp})$ 
        if  $H \not\subseteq B$  then
          begin
            generativo = false
            exit
          end
        end
      end
    end
  end
```

⁸Closed-World Assumption — CWA.

se *generativo(P)* for *true*, então o programa lógico *P* consiste apenas de cláusulas generativas. Os principais predicados Prolog, que implementam o meta-programa *generativo(P)* são descritos e comentados a seguir.

Este procedimento foi implementado em Prolog. Na implementação realizada, o predicado de mais alto nível é *verifies_generative(<arquivo_entrada>,Flag)*, que cuida da abertura e do fechamento do arquivo onde o programa *P* está gravado. Cada cláusula deste programa *P*, armazenada no *<arquivo_entrada>* na forma padrão

$$A:-L_1,L_2,\dots,L_n.$$

é lida e transformada na a estrutura

$$(A,[L_1,L_2,\dots,L_n])$$

A cláusula unitária — que representa um fato — é transformada na estrutura $(A,[])$. Após a transformação, a cláusula é passada para o procedimento *generative_clause/2*, que é o que realmente verifica se a cláusula é ou não generativa. O fim do arquivo é reconhecido pelo *generative_clause/2*, através do átomo *empty*, que funciona como critério de parada, quando o programa é composto apenas por cláusulas generativas. A cláusula que trata essa situação é

```
generative_clause(empty,true):-
    !.                                % corte verde
```

A segunda cláusula do programa *generative_clauses/2* trata as cláusulas unitárias do programa *P*. Uma cláusula unitária é generativa se e somente se não tiver variáveis livres.

```
generative_clause((Head,[]),Flag):-
    !,
    copy(Head,Head1),
    ifthenelse(numbevars(Head1,0,0),Flag = true,Flag=false).
```

A terceira cláusula daquele programa, que trata de cláusulas gerais do programa *P*, é mostrada a seguir.

```
generative_clause((Head,ListLiterals),Flag):-
    freeze_vars((Head,ListLiterals),(FVarsHead,FVarsListLiterals)),
    ifthenelse(subset(FVarsHead,FVarsListLiterals),Flag=true,Flag=false).
```

Deve ser observado que, neste caso, é necessário um tratamento explícito das possíveis variáveis na cláusula. Este é um problema delicado, que mostra uma deficiência no poder de expressão da linguagem Prolog [O'Keefe 90, Sterling 86].

É importante lembrar que `generative_clause/2` é um meta-programa⁹ e, como tal, precisa tratar as variáveis que fazem parte das cláusulas do programa *P* como constantes, de maneira a poder identificá-las pelo seus respectivos nomes. Este problema é solucionado pelo predicado meta-lógico `freeze_vars/2`, cuja ação é a de transformar variáveis instanciadas de termos, em constantes únicas — *variáveis frozen*. O comportamento das variáveis, após a transformação realizada por `freeze_vars/2`, é análogo ao comportamento de constantes. O predicado `freeze_vars/2`, onde `FVarsHead` é a lista das frozen variáveis da cabeça da cláusula (`Head`) e `FVarsListLiterals` é a lista das frozen variáveis do corpo da cláusula, está definido da seguinte forma

```
freeze_vars((Head,ListLiterals),(FVarsHead,FVarsListLiterals)):-
    freeze_oneliteral(Head,0,N1,FHead),
    freeze_listliterals(ListLiterals,N1,N2,FListLiterals),
    collect_frozenvars_oneliteral(FHead,FVarsHead),
    collect_frozenvars_listliterals(FListLiterals,FVarsListLiterals1),
    flatten(FVarsListLiterals1,FVarsListLiterals).

freeze_oneliteral(H,N1,N2,FH):-
    copy(H,FH),
    numbervars(FH,N1,N2).

freeze_listliterals([],N,N,[]).
freeze_listliterals([H|T],N1,N2,[H1|T1]):-
    freeze_oneliteral(H,N1,N3,H1),
    freeze_listliterals(T,N3,N2,T1).

collect_frozenvars_oneliteral(FTerm,FVarsTerm):-
    list_frozenvars(FTerm,FVarsTerm).

collect_frozenvars_listliterals([],[]).
collect_frozenvars_listliterals([FTerm|LFTerms],[FVarsTerm|FLVarsTerm]):-
    collect_frozenvars_oneliteral(FTerm,FVarsTerm),
    collect_frozenvars_listliterals(LFTerms,FLVarsTerm).

list_frozenvars('$VAR'(X),['$VAR'(X)]):-
    !.                                     % corte vermelho
list_frozenvars(V,[]):-
    functor(V,_,0),
    !.
list_frozenvars(Term,LVars):-
    functor(Term,Func,N),
    list_frozenvars(N,Term,LLVars),
    flatten(LLVars,LVars).

list_frozenvars(0,Term,[]):-
    !.                                     % corte verde

list_frozenvars(N,Term,[Vargs|Vs]):-
```

⁹ Programa que trata outros programas como dados.

```

    N > 0,
    arg(N,Term,Arg),
    list_frozenvars(Arg,Vargs),
    N1 is N - 1,
    list_frozenvars(N1,Term,Vs).

numbervars('$VAR'(N),N,N1) :-
    !,
    N1 is N + 1.

numbervars(Term,N1,N2):-
    nonvar(Term),
    functor(Term,Name,N),
    numbervars(0,N,Term,N1,N2).

numbervars(N,N,Term,N1,N1).
numbervars(I,N,Term,N1,N3):-
    I < N,
    I1 is I + 1,
    arg(I1,Term,Arg),
    numbervars(Arg,N1,N2),
    numbervars(I1,N,Term,N2,N3).

```

onde o predicado `numbervars(Term,N1,N2)` instancia, sucessivamente, as diferentes variáveis em `Term`, com `$VAR(N)` e com `N` variando de `N1` a `N2 - 1`. A fim de exemplificar, suponha que a cláusula do programa *P* sob investigação seja

```
p(f(X,Y,a),X,g(Z,Y)):- m(Z,g(Z,a)),q(b,f(Y,X,Z),Z),p(b).
```

Primeiramente, ela será transformada para a estrutura

```
(p(f(X,Y,a),X,g(Z,Y)), [m(Z,g(Z,a)), q(b,f(Y,X,Z),Z), p(b)])
```

e `generative_clause/2` será ativado, que por sua vez ativará `freeze_vars/2`, como mostrado a seguir.

```
?- freeze_vars((p(f(X,Y,a),X,g(Z,Y)), [m(Z,g(Z,a)), q(b,f(Y,X,Z),Z), p(b)]), (L1,L2)).
```

```

X = '$VAR'(0)
Y = '$VAR'(1)
Z = '$VAR'(2)
L1 = ['$VAR'(1), '$VAR'(2), '$VAR'(0), '$VAR'(1), '$VAR'(0)]
L2 = ['$VAR'(2), '$VAR'(2), '$VAR'(2), '$VAR'(2), '$VAR'(0), '$VAR'(1)]->

```

Assim, se o programa `verifies_generative/2` é interrogado com um arquivo que contém o seguinte programa

```

parent(beth,tim).                parent(martin,anna).
parent(fred,tim).                parent(martin,rachel).

```

```

parent(jackie,vicky).           parent(rachel,vera).
parent(tim,vicky).             parent(rachel,sean).
parent(vicky,anna).            parent(ed,vera).
parent(vicky,rachel).           parent(ed,sean).

predecessor(X,Z):- parent(X,Z).
predecessor(X,Z):- parent(X,Y),predecessor(Y,Z).

```

ele será bem sucedido, com a variável **Flag** — que indica se um programa é generativo ou não — instanciada com a constante **true**.

Os predicados auxiliares **subset/2**, **list/1**, **flatten/2**, **append/3** etc., têm implementações simples (ver [Monard 93a, Monard 93b]) e não serão tratados aqui.

9.3.2 Algoritmos para a Geração do Fechamento da Derivação Atômica de um Programa Lógico Generativo

A seguir são apresentadas duas versões do algoritmo para a geração do fechamento-da-derivação-atômica (Definição 9.2.7, pg. 116) de um programa lógico generativo P . A primeira usa raciocínio de encadeamento para frente, e a segunda, raciocínio de encadeamento para trás. Em ambas as versões, P é um programa lógico definido como o conjunto de cláusulas generativas $\{C_1, \dots, C_m\}$, sendo cada cláusula da forma

$$C_i : A_i \leftarrow L_{i_1}, L_{i_2}, \dots, L_{i_{n_i}} \quad \forall i = 1, \dots, m \text{ e } n_i \geq 0$$

Versão I - Encadeamento para Frente

```

begin
input  P
fila_fato_naousado ← fila de todos os fatos
fila_fato_usado ← ∅
fila_clausula ← fila de todas as cláusulas C (que não são cláusulas unitária)
repeat
  while fila_fato_naousado ≠ ∅ do
    begin
      let F a primeira literal em fila_fato_naousado
      for cada C em fila_clausula que possui em seu corpo uma literal10 que
        “casa” com F do
        begin
          criar a nova cláusula C' a partir de C removendo a literal F após as
            necessárias instanciações11 das variáveis em C terem sido feitas
          if C' for uma cláusula unitária
            then adicionar C' como último elemento de fila_fato_naousado

```

¹⁰Se existirem átomos negados no corpo da cláusula, após a primeira iteração *fila_fato_naousado* pode conter a negação de um fato provado considerando a hipótese do mundo fechado.

¹¹*Bindings*.


```

        else adicionar  $C'$  como primeiro elemento de fila_clausula
      end
    end
  for cada átomo negado  $\neg B$  de cláusulas em fila_clausula tal que  $B$  não “casa”
    com nenhum dos átomos em fila_fato_usado adicionar  $\neg B$  como o primeiro
    elemento de fila_fato_naousado
  until fila_fato_naousado =  $\emptyset$ 
output fila_fato_usado
end

```

Versão II - Encadeamento para Trás

```

begin
input  $P, HDepth$ 
  begin Model  $\leftarrow \emptyset$ 
    repeat considere uma cláusula  $C$  de  $P$ 
      if  $C$  for uma cláusula unitária
        then Model  $\leftarrow Model \cup C$ 
      else
        begin
           $HDepth = HDepth - 1$ 
          if  $HDepth \geq 0$  then
            repetir o procedimento para cada  $L_i$  de  $C$  e incrementar o Model
          end
        until todas as cláusulas de  $C$  tenham sido consideradas
      output Model
    end
  end
end

```

Discussão da Implementação da Versão II

Na implementação realizada, o predicado de mais alto nível é

`heasy_model(Hdepth, <arquivo_entrada>, <arquivo_saida>)`

que cria o modelo heasy de profundidade $Hdepth$ do programa P , gravado no arquivo `<arquivo_entrada>`, gravando esse modelo no `<arquivo_saida>`.

De maneira análoga à implementação anterior, este predicado cuida da transformação de cada cláusula do programa P , armazenada em `<arquivo_entrada>` na forma padrão, para a estrutura mostrada na Seção 9.3.1 e grava esta estrutura na memória principal em uma forma conveniente, facilmente acessada pelo predicado `clause_considered/2`.

O predicado Prolog que faz a geração do modelo é o `run_heasy/2`, através do meta-programa `heasy/2`, usando como parâmetros: cada cláusula do programa que representa a teoria do domínio, e o número inteiro fornecido como a profundidade desejada para o modelo.

`run_heasy(Prog, HDepth) :-`

```

        clause_considered(Prog,_),
        heasy(Prog,HDepth),
        fail.

run_heasy(_,_).

```

O meta-programa `heasy/2` (através do meta-programa auxiliar `heasy/3` que controla a profundidade da prova) executa basicamente cada uma das metas — cabeças — das cláusulas do programa *P*, com respeito ao programa *P*. O meta-programa tenta, pois, provar que cada meta segue logicamente do programa *P*. Isto é realizado utilizando o Motor de Inferência do Prolog, que realiza a prova dos programas utilizando estratégia de encadeamento para trás.

Em casos reais, o programa *P* — que especifica a teoria do domínio em forma intencional — não é constituído somente de cláusulas Prolog puro. É de se esperar que ele possua relações extralógicas, tais como operações aritméticas, relações de ordem, etc., expressas através dos vários predicados pré-definidos da linguagem Prolog. Estes predicados não são definidos por cláusulas no programa *P* e, portanto, precisam ser tratados de forma diferenciada, uma vez que devem ser executados diretamente pelo Prolog. O predicado `system_literal/1` é responsável por reconhecer esses predicados pré-definidos do Prolog. Finalmente, o predicado `herbrand_instantiation/1` é responsável por lembrar a(s) meta(s) provada(s). Segue a listagem dos procedimentos mais importantes.

```

heasy(Prog,HDepth):- heasy(Prog,-1,HDepth).

heasy([],_,HDepth):-!.

heasy(Prog,HDepth,HDepth):- !.           %limite da prova foi alcançado

heasy([Prog|Progs],Depth,HDepth):-
    heasy(Prog,Depth,HDepth),
    heasy(Progs,Depth,HDepth).

heasy(Prog,Depth,HDepth):-
    literal(Prog),
    clause_considered(Prog,Body),
    Depth1 is Depth + 1,
    Depth1 < HDepth,
    heasy(Body,Depth1,HDepth),
    herbrand_instantiation(Prog).

heasy(Prog,Depth,HDepth):-
    system_literal(Prog),                % predicado pre-definido
    Prog.                                % nao conta qualquer resolucao binaria

heasy(not not Prog,Depth,HDepth):-
    !,
    heasy(Prog,Depth,HDepth).

```

```

heavy(not Prog,Depth,HDepth):-
    not heavy(Prog,Depth,HDepth).

herbrand_instantiation(Literal):-
    not already_in(Literal),
    insert(Literal).

herbrand_instantiation(_).

```

As implementações realizadas para transformar a teoria do domínio da forma intencional para a forma extensional fazem parte do *Ambiente de Aprendizado Relacional PLI*, descrito a seguir.

9.4 O Ambiente de Aprendizado Relacional — PLI

O ambiente de aprendizado relacional, mostrado na Figura 9.2, é formado, até o momento, pelos dois sistemas de programação lógica indutiva mais representativos, difundidos e utilizados — *GOLEM* e *FOIL* — tratados detalhadamente no Capítulo 11.

Figura 9.2: Ambiente de Aprendizado Relacional — PLI

Neste ambiente, o Módulo que transforma a teoria do domínio da forma intencional em extensional, indicado por [-----], pode ser utilizado. A expressão extensional é gerada apenas para teoria do domínio expressa como cláusulas generativas. O papel do customizador de primeira ordem é o de adequar as entradas ao formato exigido por cada um dos softwares.

Entre todos os sistemas PLI desenvolvidos, os sistemas *GOLEM* e *FOIL* são os que apresentam um maior número de situações de aprendizado aplicadas a problemas reais. Ambos têm implementações razoavelmente eficientes e estão disponíveis via ftp. Problemas e novas versões são anunciados via lista de usuários e listas de AM em geral. No ambiente já estão instaladas as versões *GOLEM 1.0 Alpha (1)* de outubro de 1990 e a versão *GOLEM* de agosto de 1992. Na última versão o *trace* de execução está mais inteligível, mas a maneira de operação e os recursos que o *GOLEM* oferece são, praticamente, os mesmos em ambas as versões.

O *FOIL* está disponível na versão 5.1, 6.0 e 6.1 (esta última ainda em fase de testes). As três últimas versões do *FOIL* diferem entre si quanto a detalhes operacionais, algumas opções disponíveis e valores *defaults*.

Apesar dos sistemas *GOLEM* e *FOIL* realizarem a busca no espaço de hipóteses em sentidos opostos, bottom-up e top-down, respectivamente, eles pertencem à uma mesma classe de sistemas PLI, conhecida como classe dos sistemas empíricos. Sistemas PLI empíricos tipicamente aprendem a definição de um único predicado a partir de um número razoável de exemplos. Nesta classe de sistemas empíricos encontram-se, também, os sistemas *mFOIL* [Džeroski 91], *LINUS* [Lavrač 90] e *MARKUS* [Grobelnik 92], que pretendemos obter e instalar futuramente como parte deste ambiente.

O sistema PLI *mFOIL* implementa o algoritmo *FOIL* com algumas alterações na forma de condução da busca e critério de parada, objetivando melhorar o tratamento dado a ruídos. Enquanto o *FOIL* usa uma heurística de ganho de informação, baseada em entropia, o *mFOIL* usa uma estimativa de precisão como heurística de busca. Essa estimativa de precisão pode ser a estimativa de Laplace ou uma mais sofisticada, chamada de m-estimativa [Cestnik 90, Cestnik 91].

O *mFOIL*, assim como uma versão simplificada do *GOLEM* — identificada por *GOLEM-SI* — estão implementados em Quintus Prolog para ambiente de WS. As versões disponíveis de ambos não estão operacionais, uma vez que não dispomos ainda do ambiente de desenvolvimento Quintus Prolog.

O sistema PLI *LINUS* adota uma filosofia um pouco diferente dos sistemas PLI tradicionais. O *LINUS* pode ser visto mais como um ambiente PLI de integração de sistemas proposicionais, que induz hipóteses na forma de cláusulas DHDB¹². Tais cláusulas são tipadas (cada variável está associada a um possível conjunto de valores) e têm a forma

$$T \leftarrow L_1, \dots, L_m$$

onde *T* é um átomo e *L*₁, ..., *L*_{*m*} são literais, restritas a definições não recursivas de predicados e a tipos não recursivos. A principal idéia em *LINUS* é transformar um

¹²Deductive Hierarchical DataBase.

problema de aprendizado relacional descrito na representação DHDB, em uma tarefa de aprendizado proposicional. Para tanto, *LINUS* incorpora os sistemas de aprendizado proposicional *Assistant* [Cestnik 87], *NEWGEM* [Mozetic 85] e *CN2* [Clark 87, Clark 91].

O sistema *PLI MARKUS* [Grobelnik 92] é um derivado do sistema *MIS* [Shapiro 83], que melhora alguns dos aspectos considerados ineficientes no *MIS*. Os principais melhoramentos que o *MARKUS* oferece são: uso de todos os fatos positivos e negativos de uma vez, remoção de cláusulas incorretas e redundantes, uso de negação, geração ótima do grafo de refinamento e controle do espaço de busca através de vários parâmetros.

9.5 Considerações Finais

Neste capítulo foram discutidas maneiras de incorporar a teoria do domínio, relativa a um certo problema, na tarefa de aprendizado. Foram abordadas, também, formas de restringir a linguagem de descrição da teoria do domínio, objetivando limitar o espaço de busca de maneira a tornar a tarefa de aprendizado exequível.

Uma forma restrita de cláusula de Horn — cláusula generativa — foi discutida em detalhes, uma vez que se a teoria do domínio $\mathcal{K}$ for expressa como um programa generativo, existe a garantia da existência de um modelo finito a ela associada.

Um algoritmo que verifica se uma dada teoria do domínio $\mathcal{K}$ consiste de cláusulas generativas foi por nós proposto, e os principais predicados que fazem parte de sua implementação, foram apresentados e discutidos.

Dois algoritmos, que geram um modelo associado a um programa utilizando, respectivamente, as estratégias de encadeamento para frente e encadeamento para trás foram propostos. Os principais predicados que participam da implementação do segundo deles, que é mais eficiente, foram também apresentados e discutidos.

Capítulo 10

Linguagem de Descrição de Hipóteses em PLI

10.1 Considerações Iniciais

O uso de lógica de primeira ordem — como linguagem de expressão de instâncias, teoria do domínio e hipóteses — expande os limites do espaço de conceitos passíveis de serem aprendidos e, conseqüentemente, se mostra como uma alternativa às restrições impostas pelo uso de uma lógica proposicional. Foi provado, entretanto, que um problema de PLI que usa cláusulas de primeira ordem como linguagem de descrição de instâncias, teoria do domínio e hipóteses induzidas é, no caso geral, indecidível [Plotkin 71a]. Se, por um lado, o uso de lógica de primeira ordem expande os limites do aprendizado, por outro, tal uso deve ser controlado e restrito, caso contrário o processo se torna indecidível. Com o objetivo de tornar o problema do aprendizado em PLI decidível, várias restrições podem ser aplicadas à própria linguagem, bem como a cláusulas nela expressas. Na prática, sistemas PLI adotam várias restrições sintáticas e semânticas que objetivam, também, limitar a complexidade do problema de aprendizado.

A restrição mais frequentemente usada por sistemas PLI é a que confina a expressão do conceito apenas a cláusulas de Horn ordenadas. Neste capítulo, as restrições a cláusulas de Horn serão abordadas considerando cláusulas de Horn livres de símbolos funcionais e cláusulas de Horn com símbolos funcionais.

O conceito de *ij-determinação* é usado em PLI com o objetivo de restringir a linguagem de descrição de hipóteses. Na literatura disponível relacionada à *ij-determinação* podem ser evidenciadas diferentes interpretações associadas a esse conceito. Isso acontece, também, com conceitos que subsidiam a definição de *ij-determinação*.

O objetivo deste capítulo é apresentar, discutir e analisar criticamente as diferentes interpretações dadas a *ij-determinação* e conceitos associados, procurando evidenciar o que precisamente cada uma delas objetiva quando usada como um bias na restrição da linguagem de descrição de hipóteses, em um ambiente de programação lógica indutiva. Subjacente a esta discussão existe a intenção de buscar uma concordância

entre as diferentes interpretações, visando uma unificação da conceituação referente a ij-determinação.

No que segue, uma cláusula de Horn será sempre considerada como uma cláusula de Horn ordenada, definida como

Definição 10.1.1 *Uma cláusula de Horn é ordenada se existe uma relação de ordem total sobre as literais que fazem parte do corpo da cláusula.*

10.2 Linguagem de Descrição de Hipóteses sem Símbolos Funcionais

Considerando uma linguagem de descrição de hipóteses $\mathcal{L}_H$ livre de símbolos funcionais, os próximos conceitos podem ser usados para restringir as hipóteses aprendidas (cláusulas de Horn livres de símbolos funcionais), expressas em $\mathcal{L}_H$.

Definição 10.2.1 *(Cláusula Livre de Símbolos Funcionais) Uma cláusula é livre de símbolos funcionais quando os termos que nela comparecem são apenas variáveis e/ou constantes (símbolos funcionais de aridade 0).*

Definição 10.2.2 *(Variáveis de Entrada e de Saída) [Cohen 93] Se*

$$A \leftarrow L_1, \dots, L_{i-1}, L_i, \dots, L_m$$

é uma cláusula de Horn ordenada, então as variáveis de entrada da literal L_i são aquelas variáveis aparecendo em L_i que também aparecem em alguma literal da cláusula $A \leftarrow L_1, \dots, L_{i-1}$; todas as outras variáveis que aparecem em L_i são chamadas de variáveis de saída.

Tabela 10.1 mostra alguns exemplos de variáveis de entrada e de saída em literais de uma cláusula.

10.2.1 Profundidade de Variáveis e Cláusulas

Alguns sistemas PLI restringem a linguagem de descrição de hipóteses permitindo cláusulas de Horn com variáveis até uma determinada profundidade. Seguem duas definições equivalentes de *profundidade de variáveis* em uma cláusula de Horn.

Definição 10.2.3 *(Profundidade de Variável) [Cohen 93] Seja a cláusula*

$$A \leftarrow L_1, L_2, \dots, L_i, \dots, L_m$$

A	L_1	L_2	L_3	L_4
$p(X_1, X_2) \leftarrow$	$q(X_1, X_2).$			
	$I = X_1 \text{ e } X_2$			
$p(X_1, X_2) \leftarrow$	$q(X_1, X_3).$	$p(X_3, X_2).$	$r(X_2).$	
	$I = X_1$ $O = X_3$	$I = X_2 \text{ e } X_3$	$I = X_2$	
$p(X_1, X_2, X_3) \leftarrow$	$q(X_1, Y_1, Z_3).$	$r(X_2, Y_2, Z_3).$	$s(X_3, Y_2, Z_2).$	$t(Z_2, Z_3, Z_1).$
	$I = X_1$ $O = Y_1 \text{ e } Z_3$	$I = X_2 \text{ e } Z_3$ $O = Y_2$	$I = X_3 \text{ e } Y_2$ $O = Z_2$	$I = Z_2 \text{ e } Z_3$ $O = Z_1$
$p(X_1, X_2) \leftarrow$	$q(X_1, Y_1).$	$r(Z_1, Z_2).$	$s(Z_1, X_2, Z).$	$t(Z, Z_2, W).$
	$I = X_1$ $O = Y_1$	$O = Z_1 \text{ e } Z_2$	$I = Z_1 \text{ e } X_2$ $O = Z$	$I = Z \text{ e } Z_2$ $O = W$

Tabela 10.1: Variáveis de Entrada/Saída de Literais em uma Cláusula

Variáveis que aparecem na cabeça da cláusula têm profundidade zero. Caso contrário, seja L_i a primeira literal que contém a variável V e seja d a máxima profundidade das variáveis de entrada de L_i ; então a profundidade de V é $d + 1$.

Definição 10.2.4 (Profundidade de Variável) [Lavrač 94] Seja a cláusula

$$p(X_1, X_2, \dots, X_n) \leftarrow L_1, L_2, \dots, L_i, \dots, L_m$$

As variáveis $X_1, \dots, X_n$ que aparecem na cabeça da cláusula têm profundidade zero. Seja V uma variável que aparece pela primeira vez na literal L_i . Seja d a máxima profundidade de qualquer outra variável em L_i que aparece na cláusula

$$p(X_1, X_2, \dots, X_n) \leftarrow L_1, L_2, \dots, L_{i-1}$$

Então a profundidade da variável V é $d + 1$.

O próximo exemplo justifica a necessidade de ampliar as Definições 10.2.3 e 10.2.4, de maneira a englobarem o tratamento de uma literal sem qualquer variável de entrada.

Exemplo 10.2.1 Seja a cláusula

$$p(X, Y) \leftarrow q1(X, Z), q2(X1, Y1), q3(X1, Z)$$

Nesta cláusula, as variáveis $X1$ e $Y1$ aparecem pela primeira vez na literal $q2(X1, Y1)$. Desde que não existem variáveis em $q2(X1, Y1)$ que tenham aparecido previamente em $p(X, Y) \leftarrow q1(X, Z)$ (i.e. $q2(X1, Y1)$ não tem variáveis de entrada), de acordo com ambas as definições, $\text{profundidade}(X1)$ e $\text{profundidade}(Y1)$ são indefinidas.

Ainda com relação às duas definições anteriores de profundidade de variáveis, é importante notar que sistemas PLI que usam uma abordagem top-down de refinamento de cláusulas, não *especializariam* uma cláusula adicionando a ela uma literal sem variáveis de entrada — as variáveis desta literal ficariam *não ligadas (ou livres)*¹ e, consequentemente, sem qualquer utilidade para o sistema [Lavrač 93]. Entretanto, sistemas PLI que adotam uma abordagem bottom-up de generalização de hipóteses, eventualmente podem construir literais sem variáveis de entrada, como a literal $q2(X1, Y1)$ do Exemplo 10.2.1. Uma sugestão nossa [Nicoletti 94a, Nicoletti 94c], para evitar esse problema, é acrescentar à Definição 10.2.3

Se uma literal não tem variáveis de entrada, todas as variáveis aparecendo na literal têm profundidade 1

e à Definição 10.2.4

Se qualquer outra variável de L_i não aparecer na cláusula

$$p(X_1, X_2, \dots, X_n) \leftarrow L_1, L_2, \dots, L_{i-1}$$

então a profundidade de V é 1

Definição 10.2.5 (*Profundidade de Cláusula*) *A profundidade de uma cláusula é a máxima profundidade entre as profundidades de variável da cláusula.*

A Tabela 10.2 mostra alguns exemplos de profundidade de variáveis em uma cláusula. As dependências entre variáveis, estabelecidas pelas suas profundidades, impõem uma ordem parcial entre as literais de uma cláusula, como pode ser visto nas Figuras 10.1 e 10.2.

10.2.2 ij-Determinação

Através da imposição de um limite à profundidade máxima de variáveis i , a complexidade sintática de cláusulas expressas em linguagens livres de símbolos funcionais pode ser restrita. A próxima definição de *determinação* foi adaptada para o caso de cláusulas livres de símbolos funcionais.

Definição 10.2.6 (*Determinação*) [Džeroski 92]

- *um predicado é determinado se todas as suas cláusulas forem determinadas*
- *uma cláusula é determinada se cada uma de suas literais for determinada*
- *uma literal em uma cláusula é determinada se cada uma de suas variáveis que não aparece nas literais que precedem a literal em questão, tem apenas uma ligação possível, dadas as ligações das variáveis que aparecem nas literais prévias*

¹Uma variável está ligada se estiver instanciada.

cláusula (C)	X ₁	X ₂	X ₃	Y ₁	Y ₂	Z ₁	Z ₂	Z ₃	profundidade(C)
$p(X_1, X_2) \leftarrow q(X_1, X_2).$	0	0							0
$p(X_1, X_2) \leftarrow q(X_1, X_3),$ $p(X_3, X_2),$ $r(X_2).$	0	0	1						1
$p(X_1, X_2, X_3) \leftarrow q(X_1, Y_1, Z_3),$ $r(X_2, Y_2, Z_3),$ $s(X_3, Y_2, Z_2),$ $t(Z_2, Z_3, Z_1).$	0	0	0	1	2	4	3	1	4
$p(X_1, X_2, X_3) \leftarrow q(Y_1, X_2),$ $r(X_3, Z_1, X_2, Y_1),$ $s(Y_2, X_1, Z_3),$ $t(Y_2, Z_3, Z_2, Z_1).$	0	0	0	1	1	2	3	1	3
$p(X_1, X_2, X_3) \leftarrow q(X_1, Z_2),$ $r(Z_2, X_3, Y_1),$ $s(Y_1, Z_1, X_1),$ $t(X_2, Z_1, Z_2, Y_2),$ $u(Z_1, Y_1, Y_2, Z_3).$	0	0	0	2	4	3	1	5	5
$p(X_1, X_2, X_3) \leftarrow q(Y_1, X_2, Y_2),$ $r(X_1, Z_3, Y_1, Z_2),$ $s(X_3, Z_1).$	0	0	0	1	1	1	3	2	3

Tabela 10.2: Profundidade de Variáveis e Cláusulas

Definição 10.2.7 (*i-determinação*) [Lavrač 94] A restrição de determinação é chamada *i-determinação* para um dado valor máximo *i* de profundidade de variável.

Definição 10.2.8 (*ij-determinação*) [Lavrač 94] Seja *j* uma constante inteira tal que apenas a classe de problemas de PLI onde todos os predicados da teoria do domínio *K* têm aridade no máximo *j* serão considerados (*j* é referenciado como limitante de aridade). Dado um limitante de aridade *j* então *i-determinação* implica *ij-determinação*.

É importante observar que enquanto *i* e *j* restringem a complexidade sintática de cláusulas, a noção de *determinação* é essencialmente uma restrição semântica, uma vez que depende de um dado conjunto de exemplos de treinamento, bem como da teoria ground do domínio.

Deve ser observado que em linguagens de descrição de hipóteses livre de símbolos funcionais, restringir a tarefa de aprendizado ao aprendizado apenas de cláusulas *ij-determinadas*, é sinônimo de construção de cláusulas de Horn onde

- a máxima profundidade entre as profundidades de variáveis não é maior do que *i*
- a aridade máxima de predicados da teoria do domínio não é maior do que *j*

10.3 Linguagem de Descrição de Hipóteses com Símbolos Funcionais

A utilização de linguagens de descrição de hipóteses com símbolos funcionais permite a definição de termos outros que apenas variáveis e constantes e dá origem ao conceito de *termo determinado*. Este conceito é, posteriormente, usado na definição de

Figura 10.1: Ordem Parcial Imposta pela Profundidade de Variáveis em $p(X1, X2, X3) \leftarrow q(X1, X2), s(Y2, X1, Z3), r(X3, Z1, X2, Y1), t(X2, Z3, Z2, Z1)$

ij-determinação para linguagens de descrição de hipóteses que permitem símbolos funcionais.

10.3.1 Termo Determinado

A seguir são apresentadas três definições de termo determinado, encontradas na literatura. Algumas imprecisões nessas definições são discutidas e, com base nessa discussão, sugerimos algumas possíveis alterações nas definições em questão [Nicoletti 94a, Nicoletti 94c].

Definição 10.3.1 (*termo determinado*) [Muggleton 90] *Seja*

- $\mathcal{K}$ programa lógico
- $\mathcal{E}$ conjunto de exemplos expressos como átomos ground
- $A \leftarrow L_1, \dots, L_m, L_{m+1}, \dots, L_n$ cláusula de Horn ordenada
- t termo encontrado em L_{m+1}

O termo t é determinado com relação a literal L_{m+1} se e somente se para toda substituição θ tal que

- $A\theta \in \mathcal{E}$
- $\{L_1, L_2, \dots, L_m\} \subset \mathcal{M}(\mathcal{K})$

Figura 10.2: Ordem Parcial Imposta pela Profundidade de Variáveis em $p(X1, X2, X3) \leftarrow q(X1, Z2), r(Z2, X3, Y1), s(Y1, Z1, X1), t(X2, Z1, Z2, Y2), u(Z1, Y1, Y2, Z3)$

existe um único átomo $L_{m+1}\theta\delta \in \mathcal{M}(K)$, i.e., para qualquer θ existe uma única substituição ground válida δ cujo domínio é constituído pelas variáveis em t

Note que as substituições válidas para um termo determinado t em qualquer literal L_{m+1} são funções das substituições aplicadas a um certo número de outros termos $s_1, \dots, s_d$ da literal L_{m+1} . Neste caso, diz-se que a literal L_{m+1} tem grau d com relação a t . Pode-se restringir uma linguagem de descrição de hipóteses estabelecendo um limite j ao máximo grau de qualquer literal com relação a qualquer termo. De maneira análoga, pode-se restringir uma linguagem de descrição de hipóteses estabelecendo um limite i à máxima profundidade de qualquer literal² na cláusula construída [Muggleton 90].

Existem alguns pontos a serem considerados com relação à Definição 10.3.1. Nela, não foi estabelecido como a “determinação de um termo” deve ser examinada. O critério de determinação de termos deve ser aplicado a termos em literais, na ordem em que tais termos comparecem, ou a posição que o termo a ser examinado ocupa na literal, é irrelevante e seu exame independe do exame prévio de qualquer outro termo?

De acordo com a Definição 10.3.1, os pré-requisitos para um termo t ser determinado com relação a uma literal L_{m+1} é que para toda substituição θ tal que

²Em [Muggleton 90] o conceito de profundidade de literal não está formalmente definido; é introduzido através de exemplos, e considera como termos apenas variáveis.

- $A\theta \in \mathcal{E}$
- $\{L_1, L_2, \dots, L_m\} \subset \mathcal{M}(K)$

existe um único átomo $L_{m+1}\theta\delta \in \mathcal{M}(K)$, ou seja, para tal θ existe uma única substituição ground válida δ que tem por domínio as variáveis em t . Algumas situações a serem consideradas são discutidas a seguir.

- quando o termo t na literal L_{m+1} tem alguma variável que também comparece em outros termos em qualquer literal anterior da cláusula, então, desde que uma das pré-condições para t ser determinado com relação a L_{m+1} estabelece que a substituição θ deve ser tal que $\{L_1, L_2, \dots, L_m\} \subset \mathcal{M}(K)$, o domínio de t deve ser *um subconjunto de variáveis de t* ao invés de *as variáveis em t* . Em outras palavras, o domínio de θ deve ser o conjunto das variáveis livres de t
- se as variáveis que aparecem em t também aparecem em outros termos de L_{m+1} , a Definição 10.3.1 não estabelece claramente qual substituição deve ser aplicada àquelas variáveis: θ or δ (ver Exemplo 10.3.1)
- é evidente que algum θ que verifica as condições $A\theta \in \mathcal{E}$ e $\{L_1, L_2, \dots, L_m\} \subset \mathcal{M}(K)$, pode especificar apenas variáveis que estão em $\{L_1, L_2, \dots, L_m\}$. Assim sendo, como verificar se $L_{m+1}\theta\delta \in \mathcal{M}(K)$, se δ é apenas aplicada a variáveis em t e se L_{m+1} tem outros termos, com variáveis que não comparecem nem em t e nem em $L_1, L_2, \dots, L_m$? Uma maneira, por nós sugerida, de lidar com essa situação, é a de considerar θ como um θ *estendido*, de maneira a incluir as variáveis de L_{m+1} (exceto aquelas que aparecem em t). Isto poderia estar implicitamente indicado na definição, se ela especificasse *um único átomo ground* $L_{m+1}\theta\delta$. Especificando esse fato, a definição estaria implicitamente estabelecendo que a substituição θ teria em seu domínio as variáveis de L_{m+1} (exceto aquelas que aparecem em t), de maneira a obter um átomo *ground*.

Se uma *extensão* de θ , como a proposta, não puder ser considerada, eventualmente existirão variáveis em L_{m+1} (outras que aquelas em t), após a aplicação de θ . Neste caso, não é possível dizer nada com relação à unicidade de δ , que é o que importa para que um termo seja determinado com relação a uma literal (ver Exemplo 10.3.2)

- outro ponto a considerar na Definição 10.3.1 está relacionado ao fato de permitir (ou não) que a substituição δ possa ser uma substituição vazia. Este será o caso quando, após a aplicação de θ , $L_{m+1}\theta \in \mathcal{M}(K)$; este fato, obviamente, torna a única substituição δ a substituição vazia ϵ

Exemplo 10.3.1 *Seja a cláusula*

$$p(X, Y) : -q(X), r(Y), s(M, f(M, N, X))$$

e suponha que o termo $f(M, N, X)$ deva ser examinado para verificar sua determinação, com relação a literal $s(M, f(M, N, X))$. Não está claramente estabelecido qual substituição deve tornar ground a variável M .

Exemplo 10.3.2 *Seja a cláusula*

$$p(X) \leftarrow q(X, Y), r(Z, f(X, W))$$

onde o termo $f(X, W)$ deve ser examinado para verificar a sua determinação com relação a literal $r(Z, f(X, W))$. Seja ainda $\mathcal{E} = p(1) \in \mathcal{M}(\mathcal{K})$ dado por

$$\begin{aligned} q(1, 2) \\ r(a, f(1, 3)) \end{aligned}$$

Para $\theta = \{X/1, Y/2\}$,

$$p(1) \leftarrow q(1, 2), r(Z, f(1, W))$$

que verifica

1. $p(X)\theta \in \mathcal{E}$
2. $q(X, Y)\theta \in \mathcal{M}(\mathcal{K})$

Pela definição original, a substituição δ deve considerar apenas W (uma vez que X foi já substituído usando θ), e portanto Z permanece não ligada; assim sendo $r(Z, f(X, W))\theta\delta$ nunca estará num modelo ground. Se, entretanto, θ for estendida como

$$\theta = \{X/1, Y/2, Z/a\}$$

então (1.) and (2.) são válidos e, para $\delta = \{W/3\}$, $r(Z, f(X, W))\theta\delta \in \mathcal{M}(\mathcal{K})$.

Como nada foi dito a respeito do domínio da substituição θ , cada uma daquelas θ que especifica apenas variáveis em A e $L_1, L_2, \dots, L_m$, pode eventualmente ser estendida.

Assim, a nossa sugestão é a de modificar a Definição 10.3.1, de maneira a atender os casos discutidos, substituindo

existe um único átomo $L_{m+1}\theta\delta \in \mathcal{M}(\mathcal{K})$, i.e., para qualquer tal θ existe uma única substituição ground válida δ , cujo domínio são as variáveis em t .

por

existe um único átomo ground $L_{m+1}\theta\delta \in \mathcal{M}(\mathcal{K})$, i.e., para qualquer tal θ existe no máximo uma substituição ground válida δ , cujo domínio são as variáveis em t .

Exemplo 10.3.3 *Seja a cláusula*

$$p(X, Y) \leftarrow q(X), r(Y), s(W, f(M, X))$$

e suponha que se deseja examinar a determinação do termo $f(M, X)$ com relação a literal $s(W, f(M, X))$. Esta cláusula tem a forma

$$A \leftarrow L_1, L_2, L_3$$

onde o termo $f(M, X)$ aparece em L_3 .

Parafraseando a Definição 10.3.1, podemos dizer informalmente que, para verificar se o termo $f(M, X)$ na literal L_3 é determinado com relação àquela literal, as variáveis no termo, a saber — M, X — devem ter uma única ligação, dadas as ligações dos termos em A, L_1 e L_2 . Desde que determinação é uma noção semântica, deve ser verificada com relação a exemplos e teoria do domínio (ground). Seja

$\mathcal{E}^+$	$\mathcal{K}$
$p(1, 2)$	$q(1)$
	$r(2)$
	$s(6, f(0, 1))$
	$s(7, f(0, 1))$

Pela Definição 10.3.1, $f(M, X)$ é determinado com relação a $L_3 = s(W, f(M, X))$, se e somente se para toda substituição θ tal que $p(X, Y)\theta \in \mathcal{E}$ e $\{L_1, L_2\}\theta \subseteq M(K)$, existe um único átomo $s(W, f(M, X))\theta\delta \in M(K)$. Deve ser observado que

- δ é uma substituição cujo domínio são as variáveis em $f(M, X)$
- a fim de ter $p(X, Y)\theta \in \mathcal{E}$, X deve ser considerado pela substituição θ
- a fim de ter $s(W, f(M, X))\theta\delta \in M(K)$, W deve ser considerado pela substituição θ

Seja a substituição $\theta = \{X/1, Y/2, W/6\}$. Tem-se

1. $p(X, Y)\theta = p(1, 2) \in \mathcal{E}$
2. $L_1\theta = q(1) \in M(K)$
3. $L_2\theta = r(2) \in M(K)$ e
4. $L_3\theta = s(6, f(M, 1))$

Assim, para $\theta = \{X/1, Y/2, W/6\}$, existe uma única substituição ground $\delta = \{M/0\}$, tal que $s(W, f(M, X))\theta\delta \in M(K)$ de maneira que o termo $f(M, X)$ é determinado com relação a L_3 . Se, por outro lado, os exemplos e teoria do domínio forem

$\mathcal{E}^+$	$\mathcal{K}$
$p(1, 2)$	$q(1)$
	$r(2)$
	$s(6, f(0, 1))$
	$s(6, f(2, 1))$

então o termo $f(M, X)$ não será determinado com relação a L_3 , desde que para a mesma substituição $\theta = \{X/1, Y/2, W/6\}$, existem duas substituições δ verificando $s(W, f(M, X))\theta\delta \in \mathcal{M}(\mathcal{K})$, a saber $\delta = \{M/0\}$ e $\delta = \{M/2\}$.

Termo determinado foi também definido por J. R. Quinlan como segue

Definição 10.3.2 (*Termo Determinado*) [Quinlan-91a] *Seja a cláusula*

$$A \leftarrow L_1, \dots, L_m$$

e seja t um termo em L_m . Sejam

- $\{Y\}$ *conjunto das variáveis que ocorrem apenas em t e*
- $\{Z\}$ *conjunto das outras variáveis³ na cláusula*

O termo t é determinado com relação a L_m se para toda substituição ground para variáveis em $\{Z\}$ existir, no máximo, uma substituição ground para as variáveis em $\{Y\}$, de maneira que a cláusula seja satisfeita. Intuitivamente, os valores de variáveis em $\{Y\}$ são determinados pelos valores de variáveis que aparecem previamente na cláusula.

Deve ser notado que a última sentença da definição não é consequência das anteriores. Os valores das variáveis em $\{Y\}$ podem ser determinados não apenas pelos valores de variáveis que aparecem previamente na cláusula, mas por valores de variáveis que aparecem na mesma literal em que aparece t .

Desde que $\{Z\}$ é o conjunto de variáveis que ocorrem na cláusula (exceto por aquelas variáveis ocorrendo apenas em t), eventualmente variáveis de termos que aparecem após o termo t na literal L_m estão no conjunto $\{Z\}$. Como não existe restrição a respeito de t ser o último termo da literal L_m , os valores das variáveis em $\{Y\}$, eventualmente, podem ser determinados por valores de variáveis de termos que aparecem na literal L_m após t , e não apenas por valores de variáveis que aparecem previamente na cláusula. O próximo exemplo ilustra essa situação.

Exemplo 10.3.4 *Seja a cláusula*

$$p(M, N) \leftarrow q1(f(M, Q), g(M)), q2(h(M), i(R, S), j(M, T))$$

Considerando a literal $q2(h(M), i(R, S), j(M, T))$ e o termo $t = i(R, S)$, os conjuntos

$$\begin{aligned} \{Y\} &= \{R, S\} \\ \{Z\} &= \{M, N, Q, T\} \end{aligned}$$

³Conjunto das variáveis que ocorrem em outros termos da cláusula

O termo t é determinado com relação a $q2(h(M), i(R, S), j(M, T))$ se para toda substituição ground para variáveis em $\{Z\}$ existir no máximo uma substituição ground para variáveis em $\{Y\}$ tal que a cláusula é satisfeita.

A próxima definição de termo determinado é uma versão ligeiramente modificada da Definição 10.3.2.

Definição 10.3.3 (Termo Determinado) [Quinlan-91] *Seja a cláusula*

$$A \leftarrow L_1, \dots, L_n$$

e seja t um termo em L_n . Sejam

- $\{Y\}$ o conjunto das variáveis que ocorrem pela primeira vez em t e
- $\{Z\}$ o conjunto das variáveis que ocorrem antes na cláusula

O termo t é determinado com relação a L_m se para toda substituição ground de variáveis em $\{Z\}$, existe no máximo uma substituição ground para variáveis em $\{Y\}$, de maneira que a cláusula seja satisfeita. Intuitivamente, os valores das variáveis em $\{Y\}$ são determinados pelos valores de variáveis que aparecem antes na cláusula.

Nesta definição, a literal considerada não é necessariamente a última, e o conjunto $\{Z\}$ é considerado como o conjunto de variáveis que aparecem na cláusula antes do que t . Contudo, o problema discutido no Exemplo 10.3.4 ainda persiste.

10.3.2 ij-Determinação

Combinando os parâmetros de profundidade de literal e de grau de literal, e usando como base a Definição 10.3.1, pg. 135, obtém-se um critério para restringir a linguagem de descrição de hipóteses, dado por

Definição 10.3.4 (ij-determinação) [Muggleton 90] *Seja K um programa lógico e sejam os exemplos $\mathcal{E}$ um conjunto de fatos ground.*

1. *toda cláusula unitária é 0j-determinada*
2. *uma cláusula ordenada $A \leftarrow L_1, \dots, L_m, L_{m+1}, \dots, L_n$ é ij-determinada se e somente se*
 - (a) *$A \leftarrow L_1, \dots, L_m$ é (i-1)j-determinada*
 - (b) *toda literal L_k em $L_{m+1}, \dots, L_n$ contém apenas termos determinados e tem grau no máximo j*

Embora nesta definição j esteja relacionado ao grau máximo de literais em uma cláusula, alguns autores dão outra interpretação a este parâmetro. Em [Rouveirol 92] encontra-se que

ij-determinação usa a propriedade que para alguns predicados no corpo de uma cláusula generalizada, dada uma instanciación de seus argumentos de entrada, existe apenas uma única possível instanciación de seus argumentos de saída. ij-determinação também inclui um limite na profundidade da quantificação existencial de uma variável, bem como um limite no número de literais necessárias para que uma literal, no corpo da cláusula generalizada, seja determinada.

Nesta interpretação, o parâmetro j é associado com o número de literais (que satisfazem uma certa condição) o que, de alguma forma, difere do conceito original, que estabelece o parâmetro j como um limite no número de termos de uma literal, os quais contribuem para que outro termo, na mesma literal, seja determinado com relação àquela literal em questão.

10.3.3 Outra Noção de Profundidade e Nível de Termo

A noção de *profundidade de termo*, introduzida em [Muggleton 93], é uma noção inteiramente diferente daquelas introduzidas inicialmente neste capítulo, via Definições 10.2.3 e 10.2.4, pg. 132, uma vez que não considera o termo como parte de uma literal que, por sua vez, é parte de uma cláusula.

Definição 10.3.5 (*Profundidade de Termo*) [Muggleton 93] A *profundidade*

- $d(V)$ de uma variável V é 0
- $d(c)$ de uma constante c é 1
- $d(f(t_1, \dots, t_n))$ de um termo $f(t_1, \dots, t_n)$ é $1 + \max d(t_i)$

Em hipóteses, limitar a profundidade de termos a 1, corresponde a trabalhar com cláusulas livres de símbolos funcionais. Esta definição de profundidade de termo trata um termo isoladamente, não o contextualizando em uma literal, que por sua vez é parte de uma cláusula.

Exemplo 10.3.5 *Considerando o termo*

$$f(g(X, Y), h(i(M, X, N), K), j(r(s(K)), M), O)$$

sua profundidade é dada por

$$\begin{aligned} d(f(g(X, Y), h(i(M, X, N), K), j(r(s(K)), M), O)) = \\ 1 + \max(d(g(X, Y)), d(h(i(M, X, N), K, L)), d(j(r(s(K)), M)), d(O)) = \\ 1 + \max(1, 3, 3) = 4 \end{aligned}$$

Figura 10.3: Profundidade de Termo

desde que

$d(g(X, Y)) = 1 + \max(d(X), d(Y)) = 1 + 0 = 1$
$d(h(i(M, X, N), K, L)) = 1 + \max(d(h(i(M, X, N))), d(K), d(L)) =$ $1 + \max(1 + \max(d(i(M, X, N))), 0, 0) = 1 + \max(1 + (1 + \max(0, 0, 0)), 0, 0) =$ $1 + \max(1 + 1, 0, 0) = 3$
$d(j(r(s(K)), M)) = 1 + \max(d(r(s(K))), d(M)) = 1 + \max(1 + \max(d(s(K))), 0) =$ $1 + \max(1 + 1, 0) = 3$

A representação pictórica do nível do termo do Exemplo 10.3.5 é mostrada na Figura 10.3.

É importante notar que existem, na literatura, duas abordagens distintas de profundidade de termo. A primeira delas, dada pelas Definições 10.2.3 e 10.2.4, se restringe a variáveis e considera a primeira ocorrência da variável na cláusula. A segunda, dada pela Definição 10.3.5, define profundidade de um termo considerando o termo isoladamente. Embora tenham o mesmo nome, o conceito de *profundidade de termo*, como estabelecido nas Definições 10.2.3 e 10.2.4,⁴ tem um significado inteiramente diferente daquele estabelecido pela Definição 10.3.5.

Muggleton define nível de um termo como segue

Definição 10.3.6 (*Nível de um Termo*) [Muggleton 93] O nível $l(t)$ de um termo t em uma cláusula C é 0 se t ocorre como argumento na cabeça de C ; e $1 + \min l(s)$ onde s e t ocorrem como argumentos na mesma literal de C .

A definição de nível de um termo não impõe qualquer ordem entre os termos em uma literal, para determinar o seu nível. Por exemplo, considerando a cláusula

⁴Mesmo sendo restritos a variáveis.

$$p(X_1, X_2, X_3) \leftarrow q(X_1, Z_1, Z_2)$$

se escolhermos calcular o nível do termo Z_2 , o qual é definido como $1 + \min l(s)$ onde $s \in \{X_1, Z_1, Z_2\}$, então $l(Z_1)$ é indefinido. Entretanto, se nível de Z_1 for calculado primeiro, então $l(Z_1) = l(Z_2) = 1$. Para contornar a indefinição, sugerimos que as seguintes condições sejam adicionadas

1. considerar a ordem dada pela ordem de aparecimento das literais na cláusula, bem como de termos nas literais
2. s e t ocorrem como argumentos na mesma literal de C e s ocorreu em qualquer literal prévia

Se essas duas condições são consideradas e a operação escolhida for *max*, então *profundidade de variável* (Definições 10.2.4 e 10.2.3) será um caso particular de *nível de um termo*.

Em [Muggleton 93] encontra-se o seguinte comentário

O nível de um termo corresponde ao parâmetro i de Muggleton e Feng e ao nível de quantificação existencial de De Raedt

embora o parâmetro i seja definido em [Muggleton 90] como *profundidade*.

Se a definição de nível corresponde ao parâmetro i , existem problemas de interpretação uma vez que é inviável que o conceito de *profundidade de termo*, como proposto na Definição 10.3.5, possa ser o mesmo que define o parâmetro j , de maneira que o seguinte comentário (relacionado a GOLEM), também encontrado em [Muggleton 93], possa ter respaldo

Ambos, o nível e profundidade de termos, são freqüentemente empregados por sistemas de aprendizado PLI para definir restrições em linguagens. As duas noções são especialmente importantes no contexto de sistemas PLI que trabalham do específico para o geral, tais como ITOU, GOLEM, CLINT and PGA.

10.4 Considerações Finais

Neste capítulo foram discutidos alguns conceitos básicos que respaldam a definição da restrição ij-determinação, bem como o próprio conceito de ij-determinação, tanto para linguagens de descrição de hipóteses sem símbolos funcionais, quanto para aquelas com símbolos funcionais.

Foi mostrado que na literatura existe um consenso com relação à ij-determinação, quando tal conceito é utilizado para restringir linguagens de descrição de hipóteses

sem símbolos funcionais. A extensão do conceito de ij-determinação a linguagens com símbolos funcionais dá origem a diferentes interpretações entre alguns autores, as quais foram apresentadas e discutidas. Deve ser ressaltado que é importante ter uma unificação do conceito de ij-determinação, uma vez que tal conceito é utilizado frequentemente para comparar o comportamento de diferentes sistemas PLI, que nele se baseiam para restringir suas hipóteses induzidas. Algumas sugestões foram por nós propostas, objetivando alcançar uma concordância entre as várias interpretações.

Capítulo 11

Os Sistemas de PLI GOLEM e FOIL

11.1 Considerações Iniciais

Este capítulo apresenta e discute dois dos sistemas de PLI mais relevantes e difundidos: o *GOLEM* [Feng 90, Muggleton 90a, Muggleton 91], que adota uma estratégia bottom-up de busca no espaço de hipóteses, e o *FOIL* [Quinlan 90, Quinlan-91, Quinlan 93, Quinlan 93a, Cameron-Jones 93], que realiza a busca no espaço de hipóteses de maneira top-down.

Ambos constroem programas lógicos — restritos à cláusulas de Horn — a partir de exemplos positivos e negativos da relação a ser aprendida usando, no aprendizado, informação fornecida pela teoria do domínio. Tanto os exemplos quanto a teoria do domínio devem ser expressos extensionalmente, na forma de fatos atômicos ground. Os dois sistemas têm um nível de eficiência considerado bom, e são capazes de aprender predicados tais como inversão de lista, quicksort, multiplicação inteira etc. Na literatura podem ser encontradas aplicações bem sucedidas de ambos os sistemas, em problemas do mundo real [Dolšak 92, Dolšak 92a, King 92, Morales 92, Muggleton 92, Quinlan 93a].

O objetivo deste capítulo é apresentar os principais conceitos usados por *GOLEM* e *FOIL*, mostrar alguns de seus detalhes operacionais, discutir a qualidade dos exemplos positivos e negativos na indução da expressão apropriada do conceito, discutir suas limitações e propor uma extensão a esses sistemas, de maneira a viabilizar a sua utilização em situações nas quais a teoria do domínio possa ser fornecida intensionalmente.

11.2 O Sistema *GOLEM*

O sistema *GOLEM* usa o conceito de ij-determinação (ver Capítulo 10) na construção da expressão do conceito. Como já visto, as restrições impostas à linguagem de des-

crição de hipóteses pelos dois limites — o i e o j da ij -determinação — controlam, respectivamente, a introdução de novas variáveis na cláusula sendo construída e a aridade dos predicados utilizados na sua construção. A quantificação existencial permitida em *GOLEM* é restrita a *existe exatamente uma (variável)*. Esta restrição é também conhecida como quantificação Hilbert $\epsilon*$. Relações expressas por cláusulas que adotam uma quantificação não- $\epsilon*$ não podem ser aprendidas usando *GOLEM* [Dolšák 92].

Em um sistema de aprendizado incremental, as cláusulas aprendidas podem ser adicionadas à teoria do domínio. Muito embora o *GOLEM* não seja incremental, é conveniente que a descrição das hipóteses se restrinja a cláusulas sintaticamente generativas, uma vez que o conceito aprendido pode ser reutilizado como teoria do domínio.

Apesar do *GOLEM* ter sido projetado para a construção de cláusulas sintaticamente generativas, durante a sua utilização, foi por nós comprovada uma situação em que tal fato não se verifica¹ [Nicoletti 94b].

GOLEM constrói hipóteses restritas, usando uma abordagem *bottom-up*, incrementalmente encontrando os *rlggs* dos exemplos fornecidos, com relação à teoria ground do domínio fornecida. Para gerar uma única cláusula, *GOLEM* seleciona randomicamente um subconjunto de pares de exemplos positivos², gera regras-candidatas usando *rlgg*, verifica consistência com os exemplos negativos disponíveis (i.e., exemplos negativos não devem ser cobertos) e escolhe o par de exemplos que tem a melhor cobertura (o maior número de exemplos positivos cobertos).

Este par é então generalizado, de maneira a incorporar novos exemplos: uma amostra randômica de exemplos positivos é selecionada e o *rlgg* entre o par e cada um dos exemplos da amostra (num máximo de vezes determinado pelo *default* do parâmetro *settings*) é calculado. Os exemplos envolvidos na obtenção do *rlgg* com a melhor cobertura (i.e., o par inicial e mais um exemplo positivo) dão continuidade ao processo, que prossegue até que a cobertura dos exemplos positivos não aumenta, com relação ao aumento da amostra. Quando o processo de generalização termina, a cláusula obtida é, então, reduzida. O processo de redução que o *GOLEM* usa objetiva eliminar literais redundantes da cláusula e está baseado em

- *redução funcional*, que usa a idéia de estabelecer o *status* dos argumentos de entrada e de saída de um predicado como uma restrição adicional na construção de suas cláusulas
- *redução baseada em negativos*, que usa a noção de que uma cláusula deve cobrir o máximo número possível de exemplos positivos, e nenhum exemplo negativo

Na construção de definições que consistem de mais do que uma cláusula, o *GOLEM* usa a abordagem de cobertura: constrói uma cláusula cobrindo exemplos positivos, remove aqueles exemplos cobertos do conjunto de treinamento e repete o processo novamente. O *GOLEM* não cria automaticamente novos descritores de predicado, ou

¹Aquela na qual exemplos negativos não são fornecidos — ver Tabela 11.4, pg. 155.

²O número de pares pode ser estabelecido pelo usuário, através do parâmetro *settings*. *GOLEM* usa 8 pares por *default*.

seja, não amplia a linguagem de descrição de hipóteses. A descrição do algoritmo, para a construção de uma cláusula, é mostrada a seguir.

11.2.1 O Algoritmo *GOLEM*

begin

$\mathcal{E}^+$: conjunto de exemplos positivos

$\mathcal{E}^-$: conjunto de exemplos negativos

$\mathcal{M}_h(\mathcal{K})$: modelo h-easy da teoria de domínio $\mathcal{K}$

n : um dado limite para o número de amostras

$Pares_n = \{ \{e_i, e_j\} \mid e_i, e_j \in \mathcal{E}^+, |Pares_n| = n \}$

$Lggs = \{ C \text{ consistente com relação a } \mathcal{E}^- \mid C = rlgg(\{e, e^*\}) \text{ com relação a } \mathcal{M}_h(\mathcal{K}), \{e, e^*\} \in Pares_n \}$

let $S = \{e, e^*\} \in Lggs$ o par cuja *rlgg* tem a maior cobertura

do

let a amostra randômica $\mathcal{E}_n^+ \subset \mathcal{E}^+, |\mathcal{E}_n^+| = n$

$Lggs = \{ C \text{ consistente com relação a } \mathcal{E}^- \mid C = rlgg(S \cup \{e'\}) \text{ com relação a } \mathcal{M}_h(\mathcal{K}), e' \in \mathcal{E}_n^+ \}$

encontrar e' que produz a maior cobertura em *Lggs*

$S = S \cup \{e'\}$

$\mathcal{E}^+ = \mathcal{E}^+ - cover(rlgg(S))$

while cobertura aumenta

end

Após a construção de uma cláusula, os exemplos positivos do conjunto $\mathcal{E}^+$, que são cobertos por ela, são removidos, e *GOLEM* repete o processo de construção de cláusula, considerando os exemplos positivos restantes. Nas próximas seções são apresentados alguns exemplos de uso do *GOLEM* no aprendizado de conceitos. Os exemplos foram propositadamente escolhidos, de maneira a mostrar a atuação do sistema em situações nas quais: exemplos negativos não são fornecidos, a expressão do conceito depende da “qualidade” dos exemplos e o conceito a ser aprendido é recursivo.

11.2.2 O Aprendizado do Conceito de *cena* a Partir de Dois Exemplos Positivos

Segue uma descrição da aplicação do *GOLEM*, em uma situação em que apenas dois exemplos positivos estão disponíveis, e nenhum exemplo negativo é fornecido. Essa situação foi por nós considerada, para evidenciar a importância dos exemplos negativos, na técnica de aprendizado utilizada pelo *GOLEM*. Neste caso simples, a maior cobertura será dada pela *rlgg* reduzida desses dois exemplos. A Figura 11.1 mostra uma representação pictórica dos exemplos positivos considerados, extraídos de [Quinlan-91].

Os dois exemplos, bem como a teoria do domínio, expressos no formato requerido pelo *GOLEM*, são mostrados na Tabela 11.1.

Segue a determinação da *rlgg* de ambos exemplos.

Figura 11.1: Dois Exemplos Positivos de *cena*

Exemplos	Teoria de Domínio ($\mathcal{K}$)
$cena(s1) \oplus$ $cena(s2) \oplus$	$sobre(s1,a,b)$ $sobre(s2,f,e)$ $esquerda(s1,b,c)$ $esquerda(s2,d,e)$ $retangulo(a)$ $retangulo(f)$ $quadrado(b)$ $quadrado(d)$ $triangulo(c)$ $triangulo(e)$

Tabela 11.1: Exemplos Positivos e Teoria do Domínio para *cena*

$$rlgg(s1,s2) = lgg((cena(s1) \leftarrow \mathcal{K}), (cena(s2) \leftarrow \mathcal{K}))$$

onde

$$\begin{aligned}
 cena(s1) \leftarrow \mathcal{K} \equiv & \neg sobre(s1,a,b) \vee \neg sobre(s2,f,e) \\
 & \vee \neg esquerda(s1,b,c) \vee \neg esquerda(s2,d,e) \\
 & \vee \neg retangulo(a) \vee \neg retangulo(f) \\
 & \vee \neg quadrado(b) \vee \neg quadrado(d) \\
 & \vee \neg triangulo(c) \vee \neg triangulo(e) \\
 & \vee cena(s1)
 \end{aligned}$$

$$\begin{aligned}
 cena(s2) \leftarrow \mathcal{K} \equiv & \neg sobre(s1,a,b) \vee \neg sobre(s2,f,e) \\
 & \vee \neg esquerda(s1,b,c) \vee \neg esquerda(s2,d,e) \\
 & \vee \neg retangulo(a) \vee \neg retangulo(f) \\
 & \vee \neg quadrado(b) \vee \neg quadrado(d) \\
 & \vee \neg triangulo(c) \vee \neg triangulo(e) \\
 & \vee cena(s2)
 \end{aligned}$$

Como visto no Algoritmo 8.4.3, pg. 106, a operação de lgg entre duas cláusulas C_1 e C_2 consiste em obter o conjunto das lggs entre as literais que compõem ambas as cláusulas. As próximas tabelas mostram o cálculo de algumas das operações de lgg entre literais.

—lgg entre—	—resultado—
$\neg \text{sobre}(s1,a,b) , \neg \text{sobre}(s1,a,b)$	$\neg \text{sobre}(s1,a,b)$
$\neg \text{sobre}(s1,a,b) , \neg \text{sobre}(s2,f,e)$	$\neg \text{sobre}(X,Y,Z)$
$\neg \text{sobre}(s1,a,b) , \neg \text{esquerda}(s1,b,c)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{esquerda}(s2,d,e)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{retangulo}(a)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{retangulo}(f)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{quadrado}(b)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{quadrado}(d)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{triangulo}(c)$	undefined
$\neg \text{sobre}(s1,a,b) , \neg \text{triangulo}(e)$	undefined
$\neg \text{sobre}(s1,a,b) , \text{cena}(s2)$	undefined
—Ligações—	
$X = \text{lgg}(s1,s2), Y = \text{lgg}(a,f), Z = \text{lgg}(b,e)$	

—lgg entre—	—resultado—
$\neg \text{sobre}(s2,f,e) , \neg \text{sobre}(s1,a,b)$	$\neg \text{sobre}(X,Y,Z)$
$\neg \text{sobre}(s2,f,e) , \neg \text{sobre}(s2,f,e)$	$\neg \text{sobre}(s2,f,e)$
$\neg \text{sobre}(s2,f,e) , \neg \text{esquerda}(s1,b,c)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{esquerda}(s2,d,e)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{retangulo}(a)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{retangulo}(f)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{quadrado}(b)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{quadrado}(d)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{triangulo}(c)$	undefined
$\neg \text{sobre}(s2,f,e) , \neg \text{triangulo}(e)$	undefined
$\neg \text{sobre}(s2,f,e) , \text{cena}(s2)$	undefined
—Ligações—	
$X1 = \text{lgg}(s2,s1), Y1 = \text{lgg}(f,a), Z1 = \text{lgg}(e,b)$	

É importante notar que a operação de lgg entre dois termos não é comutativa, ou seja,

$$\text{lgg}(s1,s2) \neq \text{lgg}(s2,s1)$$

—lgg entre—	—resultado—
$\neg \text{esquerda}(s1,b,c) , \neg \text{sobre}(s1,a,b)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{sobre}(s2,f,e)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{esquerda}(s1,b,c)$	$\neg \text{esquerda}(s1,b,c)$
$\neg \text{esquerda}(s1,b,c) , \neg \text{esquerda}(s2,d,e)$	$\neg \text{esquerda}(X,M,N)$
$\neg \text{esquerda}(s1,b,c) , \neg \text{retangulo}(a)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{retangulo}(f)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{quadrado}(b)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{quadrado}(d)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{triangulo}(c)$	undefined
$\neg \text{esquerda}(s1,b,c) , \neg \text{triangulo}(e)$	undefined
$\neg \text{esquerda}(s1,b,c) , \text{cena}(s2)$	undefined
—Ligações—	
$X = \text{lgg}(s1,s2), M = \text{lgg}(b,d), N = \text{lgg}(c,e)$	

—lgg entre—	—resultado—
$\neg \text{esquerda}(s2,d,e) , \neg \text{sobre}(s1,a,b)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{sobre}(s2,f,e)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{esquerda}(s1,b,c)$	$\neg \text{esquerda}(X,M,N)$
$\neg \text{esquerda}(s2,d,e) , \neg \text{esquerda}(s2,d,e)$	$\neg \text{esquerda}(s2,d,e)$
$\neg \text{esquerda}(s2,d,e) , \neg \text{retangulo}(a)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{retangulo}(f)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{quadrado}(b)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{quadrado}(d)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{triangulo}(c)$	undefined
$\neg \text{esquerda}(s2,d,e) , \neg \text{triangulo}(e)$	undefined
$\neg \text{esquerda}(s2,d,e) , \text{cena}(s2)$	undefined
—Ligações—	
$X1 = \text{lgg}(s2,s1), M1 = \text{lgg}(d,b), N1 = \text{lgg}(e,c)$	

O cálculo das lggs entre as literais das duas cláusulas, permite reescrever

$$\text{lgg}((\text{cena}(s1) \leftarrow K), (\text{cena}(s2) \leftarrow K))$$

como

$$\begin{aligned} & \neg \text{sobre}(s1,a,b) \vee \neg \text{sobre}(X,Y,Z) \vee \neg \text{sobre}(s2,f,e) \vee \neg \text{sobre}(X1,Y1,Z1) \\ & \vee \neg \text{esquerda}(s1,b,c) \vee \neg \text{esquerda}(X,M,N) \vee \neg \text{esquerda}(X1,M1,N1) \vee \neg \text{esquerda}(s2,d,e) \\ & \vee \neg \text{retangulo}(a) \vee \neg \text{retangulo}(Y) \vee \neg \text{retangulo}(Y1) \vee \neg \text{retangulo}(f) \\ & \vee \neg \text{quadrado}(b) \vee \neg \text{quadrado}(M) \vee \neg \text{quadrado}(M1) \vee \neg \text{quadrado}(d) \\ & \vee \neg \text{triangulo}(c) \vee \neg \text{triangulo}(N) \vee \neg \text{triangulo}(N1) \vee \neg \text{triangulo}(e) \\ & \vee \text{cena}(X) \end{aligned}$$

Eliminando as literais irrelevantes que são *verdade*, uma vez que fazem parte da teoria do domínio, a cláusula é reduzida à

$$\begin{aligned} & \neg \text{sobre}(X,Y,Z) \vee \neg \text{sobre}(X1,Y1,Z1) \\ & \neg \text{esquerda}(X,M,N) \vee \neg \text{esquerda}(X1,M1,N1) \\ & \neg \text{retangulo}(Y) \vee \neg \text{retangulo}(Y1) \\ & \neg \text{quadrado}(M) \vee \neg \text{quadrado}(M1) \\ & \neg \text{triangulo}(N) \vee \neg \text{triangulo}(N1) \\ & \vee \text{cena}(X) \end{aligned}$$

Desde que as literais

$$\begin{aligned} & \text{sobre}(X1,Y1,Z1) \\ & \text{esquerda}(X1,M1,N1) \\ & \text{retangulo}(Y1) \\ & \text{quadrado}(M1) \\ & \text{triangulo}(N1) \end{aligned}$$

têm variáveis que não aparecem na cabeça da cláusula — $\text{cena}(X)$ — e não são variáveis ligadas, a cláusula é reduzida à

$\neg \text{sobre}(X,Y,Z) \vee$
 $\neg \text{esquerda}(X,M,N) \vee$
 $\neg \text{retangulo}(Y) \vee$
 $\neg \text{quadrado}(M) \vee$
 $\neg \text{triangulo}(N) \vee$
 $\text{cena}(X)$

que é, posteriormente, reduzida à $\text{cena}(X)$. Esta última redução sempre acontecerá na ausência de exemplos negativos, uma vez que *GOLEM* sempre elimina literais do corpo da cláusula, quando a eliminação não provocar a cobertura, pela cláusula, de exemplos negativos. Assim, quando nenhum exemplo negativo participa do processo de construção da expressão do conceito, *GOLEM* constrói uma cláusula extremamente geral: uma cláusula unitária, que tem por cabeça o predicado a ser aprendido.

Como comentado anteriormente, qualquer sistema indutivo é extremamente sensível à qualidade dos exemplos fornecidos. Como *GOLEM* é direcionado em seu processo de generalização — que acontece via eliminação de literais do corpo da cláusula — pelos exemplos negativos, a ausência destes provocará a mais ampla generalização possível (a obtenção da cláusula unitária). Isso sempre acontece na ausência de exemplos negativos, por mais completo e exaustivo que tenha sido o conjunto de exemplos positivos fornecido.

11.2.3 O Aprendizado do Conceito de *cena*

Este é um exemplo mais complexo que o anterior, que mostra como o *GOLEM* aprende o conceito de *cena* a partir de exemplos positivos e negativos. Uma cena consiste de três figuras geométricas diferentes — escolhidas entre retângulo, quadrado, triângulo e reta. Os cinco exemplos positivos e cinco exemplos negativos utilizados no aprendizado podem ser visualizados nas Figuras 11.2 e 11.3 respectivamente, onde uma cena pode ser descrita como

um retângulo sobre qualquer outro objeto e
um quadrado à esquerda de qualquer outro objeto

Os exemplos positivos e negativos, bem como a teoria do domínio, expressos na forma requerida por *GOLEM*, fazem parte das Tabelas 11.2 e 11.3.

Exemplos Positivos	Exemplos Negativos
$\text{cena}(s1).$	$\text{cena}(s10).$
$\text{cena}(s2).$	$\text{cena}(s11).$
$\text{cena}(s3).$	$\text{cena}(s12).$
$\text{cena}(s4).$	$\text{cena}(s13).$
$\text{cena}(s5).$	$\text{cena}(s14).$

Tabela 11.2: Exemplos Positivos e Negativos para o Aprendizado de *cena*

Com esses dados, o *GOLEM* aprende o conceito de *cena* expresso como

Figura 11.2: Exemplos Positivos para o Aprendizado de *cena*

`cena(A) :- sobre(A,B,C), esquerda(A,D,E), retangulo(B), quadrado(D).`

Segue a ativação do *GOLEM* (sem a opção de *tracing*), usando como primeiro identificador para o arquivo de saída *cena*.

```
ianomami:/work1/ianomami/IA/carmo/golem/examples>>>../golem
!- doall(cena).
[Read in 83ms]
[REMOVED: 5 FORES, 0 NEGS]
[Foreground cover removed in 0ms]
[FORES: 0, BACKS: 55, NEGS: 2, RULES: 1]
[Clauses written to cena.r]
[Total time taken for induction = 249ms]
!- read(rules,cena).
!- show(rules).
cena(A) :- sobre(A,B,C), esquerda(A,D,E), retangulo(B), quadrado(D).
!-
```

Um aspecto relevante a ser considerado em qualquer algoritmo de aprendizado está relacionado à qualidade dos exemplos fornecidos que subsidiam o aprendizado. A Tabela 11.4 contém as várias regras aprendidas por *GOLEM*, para a expressão do mesmo conceito *cena*, quando diferentes subconjuntos do conjunto de exemplos negativos {s10, s11, s12, s13, s14} são fornecidos, mostrando que apenas os exemplos negativos {s13, s14} são necessários para a indução da expressão mais específica e característica do conceito.

Teoria do Domínio		
sobre(s1,a,b).	retangulo(a).	quadrado(b).
sobre(s2,f,e).	retangulo(f).	quadrado(d).
sobre(s3,g,h).	retangulo(g).	quadrado(h).
sobre(s4,q,r).	retangulo(q).	quadrado(p).
sobre(s5,x,v).	retangulo(x).	quadrado(v).
sobre(s10,l,k).	retangulo(k).	quadrado(l).
sobre(s11,m,n).	retangulo(n).	quadrado(o).
sobre(s12,u,r).	retangulo(r).	quadrado(t).
sobre(s13,x1,x2).	retangulo(x1).	quadrado(x3).
sobre(s14,x6,x5).	retangulo(x5).	quadrado(x4).
triangulo(c).	esquerda(s1,b,c).	reta(r).
triangulo(e).	esquerda(s2,d,e).	reta(y).
triangulo(i).	esquerda(s3,h,i).	reta(u).
triangulo(j).	esquerda(s4,p,r).	reta(x2).
triangulo(m).	esquerda(s5,v,y).	reta(x6).
	esquerda(s10,j,k).	
	esquerda(s11,n,o).	
	esquerda(s12,r,t).	
	esquerda(s13,x2,x3).	
	esquerda(s14,x4,x5).	

Tabela 11.3: Teoria de Domínio para o Aprendizado de *cena*

informação que extrai dos exemplos, para guiar a busca. Sob certos aspectos, esse sistema está mais diretamente relacionado e, de certa forma, pode ser abordado como uma evolução dos tradicionais métodos proposicionais de aprendizado. Nesse sentido, contrasta com o *GOLEM*, este mais voltado à utilização das técnicas e abordagens adotadas em Programação Lógica. O aprendizado de uma relação, utilizando o *FOIL*, consiste na construção de cláusulas de Horn sem símbolos funcionais, que expressam a relação a ser aprendida, a partir de

- um conjunto de tuplas positivas e negativas³ da relação a ser aprendida
- um conjunto de tuplas que definem outras relações — teoria do domínio

Exemplos Negativos					Regras Aprendidas
s10	s11	s12	s13	s14	
					cena(A).
*					cena(A):-sobre(A,B,C),retangulo(B).
	*				cena(A):-sobre(A,B,C),retangulo(B).
		*			cena(A):-sobre(A,B,C),retangulo(B).
*	*	*			cena(A):-sobre(A,B,C),retangulo(B).
			*		cena(A):-esquerda(A,B,C),quadrado(B).
*			*		cena(A):-esquerda(A,B,C),quadrado(B).
*	*	*	*		cena(A):-esquerda(A,B,C),quadrado(B).
*			*	*	cena(A):-sobre(A,B,C),esquerda(A,D,E),retangulo(B),quadrado(D).
*	*	*	*	*	cena(A):-sobre(A,B,C),esquerda(A,D,E),retangulo(B),quadrado(D).
			*	*	cena(A):-sobre(A,B,C),retangulo(B).
			*	*	cena(A):-sobre(A,B,C),esquerda(A,D,E),retangulo(B),quadrado(D).

Tabela 11.4: Aprendizado de *cena* Usando Diferentes Exemplos Negativos

Figura 11.4: Hierarquia de Animais

Exemplos Positivos		Exemplos Negativos
ako(carnivoro,mamifero).	ako(vertebrado,animal).	ako(simio,felino).
ako(mamifero,animal).	ako(peixe,vertebrado).	
ako(tigre,mamifero).	ako(mamifero,vertebrado).	
ako(felino,mamifero).	ako(simio,primata).	
ako(carnivoro,animal).	ako(carnivoro,vertebrado).	
ako(herbivoro,vertebrado).	ako(felino,vertebrado).	

Tabela 11.5: Exemplos Positivos e Negativos para o Aprendizado de *ako* pelo *GOLEM*

Na construção da expressão do conceito, são utilizadas as relações fornecidas, bem como, eventualmente, a própria relação a ser aprendida (cláusulas recursivas). O *FOIL* pode ser visto, com certa reserva, como um sistema de aprendizado de múltiplos conceitos: dado um conjunto de relações expressas extensionalmente, ele focaliza o aprendizado em cada uma das relações por vez, e usa as restantes como teoria do domínio. Na realidade, ele transforma o aprendizado de n relações, em n aprendizados de 1 relação. Essa abordagem de aprendizado de múltiplos conceitos pode ter inconvenientes, como aqueles detectados em situações onde ocorre recursão mútua⁴.

O *FOIL* inicializa o processo de aprendizado com um conjunto de treinamento contendo todas as tuplas $\oplus$ e $\ominus$, constrói uma cláusula de Horn que cobre algumas das tuplas

³ As tuplas negativas, quando não fornecidas, são automaticamente geradas pela hipótese do mundo fechado.

⁴ e.g. [Džeroski 92]

$p(X, Y) \leftarrow r(X, Z), q(Z, Y)$

$q(X, Y) \leftarrow s(X, Z), p(Z, Y)$

Teoria do Domínio	
akol(invertebrado,animal).	ako(invertebrado,animal).
akol(vertebrado,animal).	ako(vertebrado,animal).
akol(ave,vertebrado).	ako(ave,vertebrado).
akol(peixe,vertebrado).	ako(peixe,vertebrado).
akol(mamifero,vertebrado).	ako(mamifero,vertebrado).
akol(carnivoro,mamifero).	ako(carnivoro,mamifero).
akol(herbivoro,mamifero).	ako(herbivoro,mamifero).
akol(primata,carnivoro).	ako(primata,carnivoro).
akol(felino,carnivoro).	ako(felino,carnivoro).
akol(homem,primata).	ako(homem,primata).
akol(simio,primata).	ako(simio,primata).
akol(leao,felino).	ako(leao,felino).
akol(tigre,felino).	ako(tigre,felino).
akol(gato,felino).	ako(gato,felino).
ako(tigre,mamifero).	

Tabela 11.6: Teoria do Domínio para o Aprendizado de *ako* pelo *GOLEM*

$\oplus$, remove do conjunto de treinamento aquelas tuplas $\oplus$ cobertas, e continua a busca pela próxima cláusula. Para a construção da cláusula são selecionadas literais que têm o maior ganho de informação. Idealmente, o processo de seleção de literais é repetido com apenas as instâncias satisfeitas, até que não existam mais instâncias negativas satisfeitas. A cabeça da cláusula sendo construída é sempre a literal que representa a relação sendo aprendida, e tem variáveis distintas para cada posição de argumento. O aprendizado consiste na construção do corpo da cláusula, processo que pode ser abordado como uma busca através do espaço dos possíveis corpos de cláusulas. O método usado é, basicamente, uma busca heurística *greedy*, que constrói a cláusula através de repetidamente adicionar a ela o que parece ser a *melhor* literal (ou conjunto de literais) naquele ponto. Toda literal que participa da cláusula deve ter, pelo menos, uma variável que tenha comparecido na cabeça ou, então, em qualquer literal que a preceda no corpo da cláusula. As possíveis literais do corpo da cláusula (incluindo a negação), são

$R(V_1, V_2, \dots, V_r)$	pelo menos uma das variáveis de tipo discreto deve estar previamente ligada. R pode ser a relação a ser aprendida, usada com restrições, de maneira a garantir o termino
$V_i = V_j$ $V_i \neq V_j$ $V_i = K$ $V_i \neq K$	uso da relação de igualdade (e sua negação) entre variáveis de tipo discreto parcialmente ligadas V_i e V_j e uma constante K especificada pelo usuário, que pode ser usada na definição
$V_i > V_j$ $V_i \leq V_j$ $V_i > T$ $V_i \leq T$	uso da relação de desigualdade onde V_i e V_j são variáveis de tipo contínuo e T um valor <i>threshold</i> extraído dos exemplos de treinamento

O uso de novas variáveis (i.e. variáveis quantificadas existencialmente) é cuidadosamente controlado. O *FOIL* trata uma literal determinada como aquela que introduz uma nova variável (com um limite de profundidade fixo), de maneira que cada tupla positiva tenha exatamente uma extensão, e nenhuma tupla negativa tenha mais do que uma extensão. Essas literais expandem o espaço das possíveis literais a serem escolhidas para compor a cláusula, no próximo estágio. A introdução de literais determinadas objetiva, precisamente, este aumento no espaço. Tal introdução — feita num único passo — é conveniente quando não for possível, num determinado ponto do processo de construção da cláusula, identificar uma literal com alto ganho, para incorporar à cláusula.

Na maioria dos casos, porém, o uso de literais determinadas não produz um aumento significativo no tempo necessário para a construção do conjunto de cláusulas. Apenas ocasionalmente tal uso irá provocar um aumento no tempo de busca da solução, uma vez que expande o espaço de busca.

A adaptação do conceito de literais determinadas, que é uma idéia emprestada do sistema *GOLEM*, contorna alguns dos problemas relacionados à busca *greedy*. Algoritmos que usam busca *greedy* tendem a sofrer o *problema de horizonte*: uma ação que pode ser conveniente, ou mesmo essencial, numa perspectiva global, pode parecer relativamente não promissora a nível local e, conseqüentemente, ser ignorada. O uso de literais determinadas tem o efeito de introduzir um *look-ahead* de baixo custo, em situações nas quais não se tem indicação de qual literal adicionar ao corpo da cláusula sendo construído.

O *FOIL* termina o processo de busca quando mais nenhuma cláusula pode ser construída ou, então, quando não existirem mais instâncias positivas de treinamento. Os dois critérios de parada implementados decidem

1. quando a expansão de uma cláusula não deve continuar, mesmo que o conjunto de treinamento sendo utilizado ainda contenha tuplas negativas (*critério-parada_2*)
2. quando o conjunto de cláusulas não deve ser expandido, mesmo que elas não cubram todas as tuplas da relação (*critério-parada_1*)

A utilização desses critérios permite que cláusulas inexatas — que cobrem exemplos negativos — possam ser construídas. A decisão pela não continuidade na expansão de uma cláusula tem por base o princípio *Restrição de Tamanho de Codificação* — *RTC*⁵ — que estabelece que

o número de bits necessários para a codificação de uma cláusula nunca deve exceder o número de bits necessários para, explicitamente, indicar o número de tuplas cobertas pela cláusula

e é detalhado a seguir. Seja

⁵Encoding Lenth Restriction — ELR. Essa restrição baseia-se, por sua vez, no *princípio do comprimento mínimo da descrição* [Rissanen 86, Quinlan 89].

- T o conjunto de treinamento
- $|T|$ o número de elementos de T
- p o número tuplas $\in$ em T

O número de bits necessários para indicar explicitamente essas p tuplas é dado por

$$\log_2(|T|) + \log_2(C_p^{|T|})$$

e o número de bits necessários para a codificação de uma literal é dado por

$$1 \quad \text{para indicar se negativa ou não}$$

$$+ \log_2(\text{número de relações}) \quad \text{para indicar qual relação}$$

$$+ \log_2(\text{número de possíveis argumentos}) \quad \text{para indicar quais variáveis}$$

Uma cláusula contendo n literais necessita da soma desses bits reduzida por $\log_2(n!)$, uma vez que todas as possíveis ordenações de variáveis são equivalentes. A adição de uma literal L_i a uma cláusula é descartada, se o número de bits necessários para a codificação da nova cláusula exceder o número de bits necessários para indicar as tuplas cobertas pela cláusula.

Se nenhuma literal puder ser adicionada à cláusula, mas esta for razoavelmente precisa (pelo menos 80% precisa), a cláusula inexata é retida como cláusula final. De maneira análoga, se nenhuma outra cláusula que satisfaz a restrição de codificação pode ser obtida, o conjunto incompleto de cláusulas é considerado como a definição final da relação fornecida. Em qualquer dos casos o sistema informa o usuário. A RTC tem duas deficiências [Lavrac 92]

- em domínios sem ruídos, algumas vezes essa restrição previne que o *FOIL* construa descrições completas
- em domínios com ruídos, a heurística do RTC permite, em algumas situações, a construção de cláusulas longas que cobrem poucos exemplos. Numa situação de aprendizado na qual o número de exemplos negativos é 1023 e de positivos é 1, de acordo com o RTC, o *FOIL* terá 20 bits para a construção da cláusula que cobre um único exemplo positivo

Na descrição de *FOIL* mostrada na próxima seção, o *critério_parada_1* corresponde a: *nenhuma literal pode ser adicionada à cláusula*. Quando esse critério de parada é satisfeito, a cláusula é testada para verificar se alcança uma precisão $\geq 80\%$. Se a precisão não ficou aquém do limite, a cláusula é aceita. É importante notar que a cláusula resultante pode ser uma cláusula inexata, uma vez que se a precisão for $< 100\%$, vai cobrir exemplos negativos. O *critério_parada_2* corresponde a: *nenhuma cláusula pode ser adicionada ao conjunto de cláusulas de Horn já obtido*. Se essa condição

de parada for satisfeita, nem todas as tuplas positivas são cobertas pela definição em cláusulas de Horn, ou seja, um conjunto incompleto de cláusulas de Horn é obtido.

Quando o processo de construção de cláusulas termina, tem início a fase de pós-processamento. Nesta fase, as cláusulas obtidas são revistas, visando eliminar redundância, através da remoção de literais. Qualquer algoritmo *greedy* tem a tendência de fazer escolhas locais que são ótimas mas que, algumas vezes, se revelam ruins a nível global. Nesse contexto, uma literal escolhida para compor a cláusula pode, subsequentemente, se tornar desnecessária ou mesmo contraprodutiva. São essas literais que o *FOIL* procura identificar e eliminar, na fase de pós-processamento. Ainda na fase de pós-processamento, as cláusulas obtidas são reordenadas, de maneira que as cláusulas recursivas são dispostas após as não recursivas.

A próxima seção mostra uma descrição geral do *FOIL*, para o aprendizado de uma relação, baseada nas descrições encontradas em [Pazzani 92, Quinlan 93].

11.3.1 O Algoritmo *FOIL*

begin

input

Pos: conjunto de tuplas $\oplus$ de *Pred*, predicado a ser aprendido

Neg: conjunto de tuplas de $\ominus$ de *Pred*, predicado a ser aprendido

Preds: conjunto de predicados definidos extensionalmente

Vars: uma tupla ordenada de nomes de variáveis para o predicado

set *Clausulas* to *Pred*

while *Pos* $\neq \emptyset$ **and** *criterio-parada_1* não satisfeito

begin

set *Nova_Clausula* to $\emptyset$

set *Velho* to *Vars*

while *Neg* $\neq \emptyset$ **and** *criterio-parada_2* não satisfeito

begin

for cada *Predicado* em *Preds*

for cada *V* em *variabilizacao*(*Pred*, *Velho*)

begin

criar uma *Literal* a partir de *Predicado* e *V*

calcular_ganho(*Literal*, *Pos*, *Neg*)

end

conjoin a literal com o ganho máximo em *Nova_Clausula*

add nova(s) variável(is) na literal a *Velho*

set *Pos* como *Pos* satisfeitas e estendidas por *Literal*

set *Neg* como *Neg* satisfeitas e estendidas por *Literal*

end

podar as literais redundantes de *Nova_Clausula*

remove de *Pos* todas tuplas que satisfazem *Nova_Clausula*

set *Neg* como o conjunto original de tuplas negativas

```

    add Nova_Clausula a Clausulas
  end
reordenar Clausulas
testar Clausulas com conjunto de exemplos teste (opcional)
return Clausulas
end

```

11.3.2 O Aprendizado do Conceito *cena*

O aprendizado do conceito *cena*, como realizado pelo *GOLEM* (ver Seção 11.2.2, pg. 148), a partir apenas de dois exemplos positivos, é inviável de ser realizado pelo *FOIL*. O fato de não terem sido fornecidos exemplos negativos faz com que o *FOIL* tente gerá-los pela hipótese do mundo fechado, o que, neste exemplo particular, é impraticável. Já no aprendizado do conceito *cena*, como descrito para o *GOLEM* na Seção 11.2.3, pg. 152, quando realizado pelo *FOIL*, a expressão do conceito aprendido é a mesma.

É importante mencionar que a sintaxe dos dados (exemplos positivos, negativos e teoria do domínio) adotada por ambos os sistemas é diferente. Os dados são passados ao *FOIL* em um arquivo, onde os predicados que fazem parte da teoria do domínio são identificados para o sistema através de um *. Os argumentos de cada predicado (exemplos e teoria do domínio) devem ter um tipo associado e, a esse tipo, ter sido previamente associados os possíveis valores que pode assumir. As instâncias positivas são separadas das negativas através de um ponto e vírgula.

O *GOLEM*, por exigir arquivos separados para instâncias positivas, negativas e teoria do domínio, torna o seu uso mais agradável e, algumas vezes, menos confuso que o do *FOIL*.

11.3.3 O Aprendizado de Recursão

A seguir é descrita a atuação do *FOIL* no aprendizado de predicados recursivos, no caso, o predicado *a_kind_of* (*ako*) (ver Seção 11.2.4, pg. 154), que é um predicado que expressa uma relação transitiva de hierarquia.

A Tabela 11.7 mostra os exemplos positivos que, para os seis experimentos conduzidos, mantiveram-se sempre os mesmos.

Exemplos Positivos	
ako(carnivoro,mamifero).	ako(vertebrado,animal).
ako(mamifero,animal).	ako(peixe,vertebrado).
ako(tigre,mamifero).	ako(mamifero,vertebrado).
ako(felino,mamifero).	ako(simio,primata).
ako(carnivoro,animal).	ako(carnivoro,vertebrado).
ako(herbivoro,vertebrado).	ako(felino,vertebrado).

Tabela 11.7: Exemplos Positivos para o Aprendizado de *ako* pelo *FOIL*

A Tabela 11.8 mostra os exemplos negativos e a Tabela 11.9, a teoria do domínio, fornecidos a cada um dos seis experimentos conduzidos, para o aprendizado do conceito *ako*. As instâncias que participam do experimento estão assinaladas com *.

Exemplos Negativos	Experimentos					
	1	2	3	4	5	6
ako(simio,felino).	*	*	*	*	*	*
ako(carivoro,invertebrado).	*	*	*			
ako(tigre,herbivoro).		*				
ako(homem,felino).		*				
ako(ave,invertebrado).		*				
ako(primata,felino).		*				

Tabela 11.8: Exemplos Negativos para o Aprendizado de *ako* pelo *FOIL*

Teoria do Domínio	Experimentos					
	1	2	3	4	5	6
akol(invertebrado,animal).	*	*	*	*	*	*
akol(vertebrado,animal).		*	*			*
akol(ave,vertebrado).		*	*			*
akol(peixe,vertebrado).		*	*			*
akol(mamifero,vertebrado).	*	*	*	*	*	*
akol(carnivoro,mamifero).	*	*	*	*	*	*
akol(herbivoro,mamifero).	*	*	*	*	*	*
akol(primata,carnivoro).	*	*	*	*	*	*
akol(felino,carnivoro).	*	*	*	*	*	*
akol(homem,primata).		*	*			*
akol(simio,primata).	*	*	*	*	*	*
akol(leao,felino).	*	*	*	*	*	*
akol(tigre,felino).	*	*	*	*	*	*
akol(gato,felino).		*	*			*

Tabela 11.9: Teoria do Domínio para o Aprendizado de *ako* pelo *FOIL*

A Tabela 11.10 mostra as várias expressões do conceito *ako* aprendido pelo *FOIL*. A diversidade de expressão é consequência dos diferentes conjuntos de exemplos negativos e de teoria do domínio disponíveis ao aprendizado.

O conceito de *ako* é aprendido corretamente apenas nos experimentos 2 e 3. Na expressão do conceito, obtida pelos experimentos 1 e 4, a variável *A* que comparece na primeira e terceira cláusula, não está ligada. Muito embora os mesmos dados tenham sido utilizados nos experimentos 4 e 5, a expressão do conceito obtida por ambos diverge devido ao uso de diferentes valores para a restrição RTC. Enquanto o experimento 5 usa a medida default para o RTC, o experimento 4 praticamente elimina essa restrição imposta na expansão de uma cláusula, ao estabelecer como a medida RTC um número incrivelmente alto. Os conceitos construídos pelos experimentos 5 e 6 não cobrem dois exemplos do conjunto de treinamento e não são representativos.

Experimento	Conceito Aprendido
1	ako(A,B):- not ako1(B,C) ako(A,B):- ako1(A,B) ako(A,B):- ako1(B,C),ako(B,C)
2	ako(A,B):- ako1(A,B) ako(A,B):- ako1(A,C),ako(C,B)
3	ako(A,B):- ako1(A,B) ako(A,B):- ako1(A,C),ako(C,B)
4	ako(A,B):- not ako1(B,C) ako(A,B):- ako1(A,B) ako(A,B):- ako1(B,C),ako(B,C)
5	ako(A,B):- ako1(C,A) ako(A,B):- not ako1(B,C)
6	ako(A,B):- ako1(A,C),ako(C,D)

Tabela 11.10: Conceitos *ako* Aprendidos nos Experimentos com *FOIL*

11.4 Considerações sobre *GOLEM* e *FOIL*

Embora ambos os sistemas não tenham sido comparativamente avaliados, de maneira sistemática, em situações empíricas, o estudo e a experimentação de cada um deles evidenciaram situações que merecem algumas considerações.

Durante o aprendizado de um conceito, ambos sistemas são capazes de processar um grande número de exemplos em um intervalo de tempo pequeno, muito embora o *FOIL* o faça mais rapidamente.

Apesar da eficiência, deve ser observado que ambos são altamente dependentes do vocabulário e da expressão dos exemplos, não sendo capazes de estender seu vocabulário, através da criação de novos descritores (no caso, novos predicados). Nenhum dos dois sistemas aprende conceitos que dependam da expansão da linguagem de descrição de exemplos e/ou conceitos, para serem aprendidos.

Como em qualquer outro sistema indutivo de aprendizado, ambos dependem completamente da adequação e qualidade dos exemplos fornecidos (tanto positivos quanto negativos), bem como da teoria do domínio disponível, a fim de construir uma descrição adequada do conceito a ser aprendido. Devido à heurística que conduz a busca no *FOIL*, pode-se dizer que esse sistema está mais adaptado e tem um desempenho melhor em situações que envolvem um volume exaustivo de exemplos. O *GOLEM*, por sua vez, exige uma escolha mais cuidadosa do conjunto de exemplos de treinamento [Nicoletti 93d].

É importante notar que o *FOIL* aumenta a expressividade mas sacrifica completeza; é um algoritmo heurístico que falha em aprender certas relações quando sua medida de avaliação que controla a busca *hill-climbing* não for apropriada. O uso de variáveis contínuas em *FOIL* é muito restrito, uma vez que apenas são permitidas literais que envolvem desigualdades entre variáveis contínuas previamente ligadas e valores *thresholds*.

Tanto o tratamento de ruído realizado pelo *GOLEM* — que permite que uma cláusula gerada cubra um número pré-determinado de exemplos negativos — quanto o realizado pelo *FOIL* — que pára a expansão de uma cláusula quando o critério RTC for satisfeito — são formas muito simplificadas e até mesmo rudimentares, de tratamento de ruídos. O *GOLEM* não leva em consideração a relação entre o número de exemplos positivos e negativos cobertos pela mesma cláusula *C*. Uma opção mais conveniente é a de deixar que o próprio usuário forneça o percentual de precisão para a construção das regras. A RTC, empregada pelo *FOIL*, também sofre de alguns inconvenientes, conforme já comentado na seção anterior.

Ambos sistemas exigem que a teoria do domínio $\mathcal{K}$ seja fornecida como fatos ground. No Capítulo 9, pg 110, propomos três algoritmos dos quais, dois deles, geram a expressão ground de uma teoria do domínio fornecida intencionalmente. Com a utilização das implementações apresentadas, o uso tanto do *GOLEM* quanto do *FOIL* se torna viável em situações nas quais a teoria do domínio disponível esteja expressa intencionalmente. Ambos aprendem um conceito. O *FOIL*, entretanto, pode simular o aprendizado de vários. A próxima tabela nomeia as principais características de sistemas PLI, e as do *GOLEM* e *FOIL* em particular.

Característica	<i>GOLEM</i>	<i>FOIL</i>
incremental	n	n
aprende múltiplos predicados	n	s
usa ganho de informação	n	s
permite literais negadas	n	s
busca top-down (especialização)	n	s
busca bottom-up (generalização)	s	n
múltiplas representações	n	n
usa oráculo ⁶	n	n
rlgg	s	n
símbolos funcionais	s	n
invenção de novos predicados	n	n
implementação eficiente	s	s
teoria do domínio	<i>ground</i>	<i>ground</i>
ij-determinação	termos e literais	literais

Tabela 11.11: Características dos Sistemas PLI *GOLEM* e *FOIL*

11.5 Considerações Finais

Neste Capítulo foram apresentados os sistemas *GOLEM* e *FOIL*, dois dos mais conhecidos e utilizados sistemas PLI. A escolha de ambos os sistemas, para pesquisas e experimentos com PLI, foi motivada por duas razões: a primeira, por apresentarem versões razoavelmente eficientes e disponíveis via ftp e a segunda, por adotarem es-

⁶Instrutor ou software que responde a perguntas específicas, geralmente relacionadas à pertinência.

estratégias inversas para a busca no espaço de hipóteses. Enquanto o *GOLEM* navega no espaço de hipóteses através de generalizações, o FOIL o faz através de especializações.

Ambos os sistemas não estão em suas versões finais e suas várias versões mostram que estão em constante processo de melhoramento. Tanto o *GOLEM* quanto o *FOIL* devem ser vistos como sistemas experimentais de aprendizado, onde idéias estão continuamente sendo testadas e validadas.

Capítulo 12

Conclusões

Este trabalho de pesquisa investigou aprendizado indutivo simbólico de máquina, caracterizando-o e abordando-o como função das linguagens utilizadas para a expressão de instâncias ($\mathcal{L}_\mathcal{E}$), conceitos ($\mathcal{L}_\mathcal{H}$) e teoria do domínio ($\mathcal{L}_\mathcal{K}$). Em função do potencial de representatividade das linguagens utilizadas e das consequentes estratégias necessárias à realização da busca pela expressão do conceito, no espaço de hipóteses, os métodos de aprendizado simbólico indutivo foram abordados como proposicionais e relacionais. A Figura 12.1 exhibe pictoricamente os dois ambientes de aprendizado, dos quais fazem parte as implementações desenvolvidas durante este trabalho. Do ambiente proposicional, fazem parte o aprendizado de árvores de decisão e o aprendizado construtivo, este último buscando uma ampliação do poder representacional das linguagens de descrição de instâncias e de hipóteses. O ambiente relacional, representado por PLI, usa linguagens representacionais mais poderosas. Entretanto, sistemas de Programação Lógica Indutiva se tornam factíveis apenas quando restrições dessas linguagens são consideradas.

Figura 12.1: Aprendizado e Linguagens

Presentemente, as aplicações de aprendizado de máquina que têm tido sucesso são as que utilizam sistemas de aprendizado proposicional. O sucesso da abordagem proposi-

cional se reflete no número razoável de sistemas eficientes que estão comercialmente disponíveis. Sistemas proposicionais, apesar de limitados, podem ser altamente eficientes em determinadas aplicações, com relação a tempo de execução, uma vez que os problemas são decidíveis e permitem o desenvolvimento de algoritmos eficientes [Monard 93]. Acreditamos, entretanto, que aprendizado proposicional possa ser explorado a nível de pesquisa, apenas através do investimento em aprendizado construtivo. Como aprendizado construtivo é altamente dependente do domínio do conhecimento, progressos em aprendizado construtivo estarão sempre confinados a domínios específicos de conhecimento o que, de certa forma, torna inviável a “portabilidade” de algumas soluções construtivas. De qualquer forma, uma área de pesquisa relacionada a aprendizado construtivo, que pretendemos investir futuramente, é a relacionada à realização de análises teóricas para buscar o “melhor” algoritmo para a construção de novos atributos, em domínios restritos. Pretendemos, ainda, investigar o aprendizado construtivo através do uso de oráculos, para direcionar o processo (que passará a ser semi-automático) de construção de novos descritores. A abordagem automática pode ainda ser experimentada tendo em conta a definição do *tipo* do valor do atributo. Embora parcial, acreditamos que a solução terá aplicabilidade a valores discretos.

O número considerável de sistemas/técnicas de aprendizado de máquina existentes evidencia o fato que não existe uma metodologia e/ou estratégia única para a solução do problema de automação do aprendizado. O que pode se dizer é que, dadas determinadas condições, existem sistemas que são mais apropriados de serem utilizados do que outros. Numa situação em que não existe um conjunto representativo de dados que reflita um determinado conceito, é inviável a consideração de aprendizado de máquina por exemplos.

Apesar da rápida evolução de PLI como uma área de pesquisa, existe ainda um longo caminho a ser percorrido antes que esse paradigma possa se estabilizar e contribuir com resultados práticos, os quais efetivamente viabilizem aprendizado de máquina de maneira eficiente. Os sistemas que possuem implementações mais eficientes (*GOLEM* e *FOIL*) estão ainda em fase experimental de desenvolvimento, têm aplicações ainda restritas e sua utilização deve ser monitorada e controlada através de conjuntos estudados de exemplos, de uma teoria do domínio conveniente e de parâmetros apropriados. O desenvolvimento de sistemas eficientes tem sido uma preocupação de pesquisadores da área; muitas aplicações do mundo real, tal como o problema da predição da estrutura de uma proteína, envolvem milhares ou mesmo milhões de exemplos. Dimensionar sistemas PLI para trabalharem com bases de dados dessas dimensões é uma tarefa altamente complexa e que precisa ser cuidadosamente planejada e pesquisada. Muggleton [Muggleton 93] sugere que métodos tradicionais, tal como o mecanismo da janela, discutido neste trabalho no contexto de aprendizado proposicional, sejam adaptados e possam ser utilizados por sistemas PLI.

Muito embora *GOLEM* e *FOIL* manipulem dados numéricos, às vezes tais manipulações não são realizadas de maneira elegante. Em estatística, o problema de predição numérica é conhecido como regressão. Existem muitos algoritmos estatísticos eficientes para a manipulação de dados numéricos. Uma área de investigação ainda em aberto é a da integração de tais abordagens a sistemas PLI.

Tanto no sistema *GOLEM* quanto no *FOIL*, a pesada restrição da exigência de uma teoria do domínio *ground* é aliviada através dos algoritmos (*generativo* e *heavy*) propostos neste trabalho. As implementações desses algoritmos viabilizam a utilização de uma teoria do domínio intencional, uma vez que eles providenciam a “tradução” da expressão intencional da teoria do domínio para a sua forma extensional. Entretanto, a utilização dessas implementações aumenta apenas o escopo de situações onde o aprendizado pode ocorrer. A exigência da expressão *ground* da teoria do domínio, por parte daqueles sistemas, ainda continua.

Com relação ao conceito de *ij-determinação*, implementado no *GOLEM* e parcialmente no *FOIL*, muito embora seja largamente utilizado, achamos que é uma restrição considerável e limitante ao aprendizado, uma vez que permite o aprendizado apenas de conceitos que sejam passíveis se serem expressos através de cláusulas *ij-determinadas*. É fundamental que pesquisas teóricas sejam conduzidas de maneira a buscar condições que permitam um relaxamento desta restrição; isso, obviamente, reverterá numa ampliação do espaço dos conceitos passíveis de serem aprendidos pelos sistemas que utilizam *ij-determinação* como restrição de linguagem de descrição de hipóteses.

Uma outra área de pesquisa em PLI que achamos importante nos aprofundarmos no futuro é a conhecida como *invenção de predicados* — que consiste em estender o vocabulário de maneira a automaticamente encontrar a definição de predicados previamente desconhecidos — está apenas se iniciando, e conta com poucos trabalhos relevantes [Stal 93].

Uma tendência que está se evidenciando na área de AM, e que se abre como uma linha de pesquisa em potencial, é a do investimento em *aprendizado multiestratégico*. Por *multiestratégico* entende-se sistemas

- *multi-inferenciais* — que combinam diferentes estratégias inferenciais, por exemplo, indução e aprendizado baseado em explicação; indução, analogia e dedução
- *multi-paradigmáticos* — que combinam diferentes estratégias computacionais, por exemplo, métodos simbólicos e redes neurais, métodos simbólicos e algoritmos genéticos, redes neurais e algoritmos genéticos

Para o desenvolvimento dos fundamentos para a construção de tais sistemas, é necessário o entendimento do papel e das condições de aplicabilidade das diferentes estratégias e diferentes paradigmas. Considerando que métodos integrados, muitas vezes, são complementares, os sistemas de aprendizado multiestratégicos se aplicam, algumas vezes, a uma variedade maior de problemas práticos do que os tradicionais sistemas monoestratégicos. Pretendemos também, futuramente, investir nesta linha de pesquisa.

Apêndice A

Busca como Método de Resolução de Problemas

Busca é um mecanismo largamente utilizado para a resolução de problemas em IA. Muitos dos problemas categorizados como de IA são extremamente complexos para serem solucionados através de técnicas diretas (e.g. como as implementadas em programação convencional). Via de regra, em um problema típico de IA, a sequência de ações necessárias para a sua solução não é conhecida a priori. Ela deve ser determinada explorando-se, sistematicamente, as possíveis alternativas, através de uma busca baseada em tentativas e erros, eventualmente direcionada por heurísticas¹.

Todo programa que resolve algum problema de IA implementa praticamente um algoritmo de busca. Para a formulação de um problema, cuja solução deve ser encontrada através de busca, é necessário a [Rich 91]

- definição do espaço de estados que contém todas as possíveis configurações (estados) relevantes; é possível definir este espaço sem enumerar explicitamente todos os estados que ele contém
- especificação de um ou mais estados pertinentes ao espaço, que descreve as possíveis configurações a partir das quais o processo de resolução do problema pode começar; são os chamados de *estados iniciais*
- especificação de um ou mais estados aceitáveis como solução do problema; são os chamados de *estados finais*
- especificação de um conjunto de regras — *operadores* — que descrevem ações disponíveis e promovem a mudança de um estado para outro

O problema pode, então, ser resolvido através da busca de um caminho entre o estado inicial e estado final, usando operadores combinados com uma estratégia de controle

¹Informalmente heurísticas são definidas como regras práticas, para as quais não há garantia de êxito e que visam melhorar a eficiência de um processo.

apropriada, de maneira a possibilitar a navegação no espaço de estados. Busca pode resumidamente ser definida como o estudo de estados e suas transições. Geralmente, o que se busca é uma solução ótima, isto é, uma solução para a qual não exista nenhum caminho menor.

O fato do processo de busca ser fundamental para a resolução do problema não significa, entretanto, que abordagens diretas não possam ser exploradas. Sempre que possível, elas podem ser inseridas como passos na busca. Operações aritméticas, por exemplo, são usadas como passos simples em uma busca. Não é apropriado, por exemplo, o uso de busca para encontrar um número com a propriedade de ser igual a $Y + (Z - X)$.

Busca é, pois, um mecanismo geral que pode ser usado quando não existem métodos diretos disponíveis ou, então, quando os que existem são muito onerosos para serem aplicados. Métodos de busca, igualmente, têm a conveniência de viabilizar a possibilidade de inserção de métodos diretos na solução de subpartes do problema.

O processo de busca pode, ainda, ser visto como o processo de percorrer um reticulado, onde cada nó representa um estado do problema e cada arco representa uma relação entre os estados representados pelos nós que ele conecta.

A maioria das técnicas de aprendizado descritas na literatura pode ser vista como técnicas de busca, sejam elas heurísticas ou exaustivas [Mitchell 82]. Os estados no espaço de busca são descrições de conceitos e o objetivo é encontrar um ou mais estados que satisfaçam os critérios de qualidade. Um sistema de aprendizado pode ser descrito em termos da estrutura de seu espaço de busca, de sua heurística de busca e de sua estratégia de busca.

Existem muitas estratégias de controle para busca. Se não existir a disponibilidade de uma heurística, uma maneira de realizar a busca é através de um método de *busca-cega*. Cada vez que é utilizado, um método de busca-cega é caracterizado pela ordem constante em que os nós são gerados, através da aplicação dos operadores. Os dois métodos de busca-cega mais comuns mostrados nas Figuras A.1 e A.2 são

- *busca em largura (breadth-first search)*
- *busca em profundidade (depth-first search)*

Os métodos de busca-cega, muitas vezes, não são viáveis porque a árvore de estados se torna muito grande antes do estado final ser alcançado. Como já mencionado, uma maneira de melhorar o desempenho de métodos de busca, é através do uso de heurísticas. Entretanto, uma grande dificuldade no uso de heurísticas reside nas eventuais discordâncias entre elas, situação que se evidencia, por exemplo, quando duas heurísticas fazem recomendações contraditórias. Se entre duas heurísticas uma for melhor que a outra, a sugestão da melhor deve ser acatada. Pode acontecer, entretanto, das duas serem igualmente boas. Quando isso acontece, o procedimento mais indicado é fazer uma avaliação numérica dos possíveis próximos estados e, então, escolher aquele com melhor valor. O método de cálculo de tal número é chamado de *função de avaliação*. Por convenção, os valores das funções de avaliação são valores

Figura A.1: Busca em Largura

Figura A.2: Busca em Profundidade

numéricos não negativos: quanto menor o valor, melhor o estado a ele associado. Estados finais geralmente têm um valor de função de avaliação 0. O uso de funções de avaliação torna possível a aplicação de duas novas estratégias de busca

- *busca best-first (variante da busca em largura obtida com a utilização de função de avaliação)*
- *busca hill-climbing (variante da busca em profundidade obtida com a utilização de função de avaliação)*

Os valores de funções de avaliação não são os únicos que ajudam em uma busca. Em alguns problemas, o interesse não é determinar *qualquer* caminho entre os estados inicial e final mas, sim, determinar um *bom* ou o *melhor* caminho. Isto pode ser operacionalizado através da atribuição de custos à aplicação de cada operador, da contabilização acumulativa de tais custos ao longo do caminho e da avaliação final do caminho. Isto exige a definição de uma função chamada *função de custo*, que mede quanto custou o movimento de um estado para outro. É importante lembrar que funções de avaliação referem-se aos próximos passos da busca, enquanto que funções de custo referem-se aos passos já realizados. O uso de função de custo dá origem a dois outros métodos de busca

- *busca branch-and-bound*
- *busca A^**

Se uma busca necessita encontrar o caminho com o custo mais baixo (ótimo) que a leve ao estado final, ela irá necessitar uma estratégia diferente. Se existe uma função de custo, mas tanto a heurística quanto a função de avaliação não são muito boas, a busca *branch-and-bound* pode ser utilizada. Ela é semelhante à busca best-first, mas é direcionada pelo custo: sempre encontra sucessores do estado cujo caminho, a partir do estado inicial, tem custo total mais baixo.

No caso de estarem disponíveis tanto a função de custo quanto a de avaliação, a busca A^* pode ser usada. A idéia é a de somar os valores das funções custo e de avaliação relativas a um estado, de maneira a obter, com isso, uma medida mais significativa de avaliação do estado, e então usar esse número para selecionar estados utilizando o algoritmo de busca best-first. A busca realizada pelo A^* , mostrada na Figura A.3, é uma espécie de produto híbrido da busca best-first (usando uma função de avaliação) com a busca branch-and-bound (usando uma função de custo), que incorpora informação e apresenta um desempenho frequentemente melhor que ambas. Nomeando

$$\begin{aligned} g(n) & \text{ função de custo do nó } n \\ h(n) & \text{ função de avaliação do nó } n \end{aligned}$$

a estimativa do custo de se ir da raiz (r) até a meta (m) passando pelo nó n é calculada pela fórmula

Figura A.3: Busca A^*

$$f(n) = g(n) + h(n)$$

A estratégia de busca A^* , semelhante à *branch-and-bound*, possui uma propriedade muito importante: se a função de avaliação para qualquer estado e_i é sempre menor ou igual ao custo real de e_i para a meta, então o primeiro caminho encontrado pela estratégia de busca A^* é o caminho de custo mínimo (ótimo). A Tabela A.1 resume as principais estratégias de busca.

estratégia de busca	função de avaliação	função de custo
depth-first	não	não
breadth-first	não	não
hill-climbing	sim	não
best-first	sim	não
branch-and-bound	não	sim
A^*	sim	sim

Tabela A.1: Estratégias Mais Comuns de Busca

Apêndice B

Entropia

Se um objeto pode ser classificado em

- N classes diferentes $C_1, C_2, \dots, C_N$ e
- a probabilidade de ocorrência do objeto na classe C_i é $p(C_i)$, $\forall i, 1 \leq i \leq N$

então a entropia de classificação — $H(C)$ — é dada por¹

$$H(C) = - \sum_{i=1}^N p(C_i) \log_2 p(C_i)$$

Entropia é uma medida da incerteza na classificação do objeto. $H(C)$ é usada em teoria da informação e interpretada como a quantidade de informação contida em uma mensagem. Quanto maior for o valor de $H(C)$, maior a incerteza com relação ao conteúdo da mensagem.

Nesse contexto, uma árvore de decisão para a classificação de exemplos pode ser vista como uma fonte de informação que, dado um exemplo, gera uma mensagem indicando a classificação do exemplo. Quando o nó da árvore contém apenas exemplos da mesma classe, a entropia é igual a 0 — significando que a decisão de classificação é definitiva para exemplos pertencentes àquele nó [Shaw 90].

Considere o problema de se estabelecer regras que possam fazer previsão, para uma dada companhia, do aumento ou diminuição de seus lucros, com base em algumas características e a partir de um conjunto de treinamento [Thompson 86]. O conjunto de atributos e o conjunto de treinamento utilizados são mostrados nas Tabelas B.1 e B.2 respectivamente.

Um algoritmo de indução que usa entropia como função de avaliação, particiona o conjunto de treinamento em subconjuntos de maneira a reduzir o máximo possível essa entropia e continua neste processo, recursivamente, até a entropia ser reduzida a 0.

¹Deve ser lembrado que se $\log_2(x) = y$, y é o número de bits necessários para representar um número x de objetos diferentes.

atributo	significado	valores
<i>tipo</i>	tipo da companhia	{hardware,software}
<i>tempo</i>	tempo do produto no mercado	{muito,normal,pouco}
<i>concorrente</i>	produto tem concorrente	{s,n}
<i>lucro (classe)</i>	lucro da companhia	{aumenta,diminui}

Tabela B.1: Atributos para a Previsão de Lucro

<i>tipo</i>	<i>tempo</i>	<i>concorrente</i>	<i>lucro (classe)</i>
software	muito	n	diminui
software	normal	s	diminui
hardware	normal	n	aumenta
hardware	muito	n	diminui
hardware	pouco	n	aumenta
software	pouco	n	aumenta
software	normal	n	aumenta
software	pouco	s	aumenta
hardware	normal	s	diminui
software	muito	s	diminui

Tabela B.2: Conjunto de Treinamento

Para a construção da árvore de decisão, usando entropia como função de avaliação, o que se busca evidenciar, dentre os atributos existentes, é aquele que possui o menor valor de entropia de classificação. O primeiro passo no cálculo da entropia de classificação relativa a um determinado atributo A , notada por $H(C|A)$, é dividir o conjunto de treinamento — ver Tabela B.2 — em subconjuntos, agrupando os exemplos que têm em comum o mesmo valor de A . A Tabela B.3 mostra a partição do conjunto de treinamento em duas subtabelas, considerando o atributo *concorrente* e seus possíveis valores.

<i>concorrente</i>	<i>lucro (classe)</i>
n	diminui
n	aumenta
n	diminui
n	aumenta
n	aumenta
n	aumenta
s	diminui
s	aumenta
s	diminui
s	diminui

Tabela B.3: Conjunto de Treinamento Agrupado por Valores do Atributo *concorrente*

A entropia de cada subtabela $H(C|A = a_j)$ é calculada para cada valor de atributo a_j e é dada pela expressão²

²Todos os logaritmos mencionados neste trabalho são considerados em relação à base 2, motivo pelo qual a especificação da base será omitida.

$$H(C|A = a_j) = - \sum_{i=1}^N p(C_i|a_j) \log p(C_i|a_j)$$

A função $p(C_i|a_j)$ é a probabilidade de que o valor da classe seja C_i quando o valor do atributo A for a_j . Calculando o valor da entropia para cada subtabela, tem-se

$$\begin{aligned} H(C|\text{concorrente} = n) &= -p(\text{aumenta}|\text{concorrente} = n) \times \\ &\quad \log p(\text{aumenta}|\text{concorrente} = n) - \\ &\quad p(\text{diminui}|\text{concorrente} = n) \times \\ &\quad \log p(\text{diminui}|\text{concorrente} = n) \\ &= -4/6 \log(4/6) - 2/6 \log(2/6) \\ &= 0.918 \end{aligned}$$

$$\begin{aligned} H(C|\text{concorrente} = s) &= -p(\text{aumenta}|\text{concorrente} = s) \times \\ &\quad \log p(\text{aumenta}|\text{concorrente} = s) - \\ &\quad p(\text{diminui}|\text{concorrente} = s) \times \\ &\quad \log p(\text{diminui}|\text{concorrente} = s) \\ &= -1/4 \log(1/4) - 3/4 \log(3/4) \\ &= 0.811 \end{aligned}$$

A expressão $p(\text{aumenta}|\text{concorrente} = n)$ é a probabilidade de que o valor da classe seja *aumenta*, quando o valor do atributo *concorrente* for n . Este valor é simplesmente o número de vezes que *lucro* = *aumenta* quando *concorrente* = n , dividido pelo número total de exemplos nos quais *concorrente* = n .

Para encontrar a entropia da tabela inteira — $H(C|\text{concorrente})$ — deve-se fazer a soma ponderada das entropias associadas a cada um dos possíveis valores do atributo. Se um atributo A tem por possíveis valores a_i , $i = 1, \dots, M$, a entropia de classificação de A é a média ponderada da entropia de cada um de seus possíveis valores a_i , $i = 1, \dots, M$, matematicamente expressa por

$$H(C|A) = \sum_{i=1}^M p(a_i) H(A|a_i)$$

No caso do atributo *concorrente* tem-se

$$\begin{aligned} H(C|\text{concorrente}) &= 6/10 \times 0.918 + 4/10 \times 0.811 \\ &= 0.8752 \end{aligned}$$

Para o exemplo em questão, $H(C|\text{tempo}) = 0.4$ e $H(C|\text{tipo}) = 1.0$. Desde que $H(C|\text{tempo})$ tem o menor valor de entropia, *tempo* é eleito como o melhor atributo para iniciar a construção da árvore.

Apêndice C

ID3 e o Assistant Professional

O objetivo deste apêndice é exemplificar a atuação de dois sistemas de aprendizado: o ambiente experimental da família TDIDT, utilizando entropia como função de avaliação, e o Assistant Professional¹, na geração de árvores de decisão. Tais árvores de decisão representam critérios gerais para avaliação de desempenho de programadores, com vistas a uma atualização salarial².

O conjunto de treinamento é formado por 38 exemplos, cada um deles expresso utilizando 8 atributos e seus respectivos valores, e de uma classe associada (atribuída por um avaliador). O conjunto de atributos utilizados na descrição dos exemplos, os possíveis valores de atributos e as possíveis classes estão mostrados nas Tabelas C.1, C.2, C.3 respectivamente.

atributo	significado
<i>gdificul</i>	grau de dificuldade do problema
<i>inic</i>	iniciativa do programador
<i>temp</i>	tempo gasto de programação
<i>creat</i>	criatividade do programador
<i>mod</i>	uso de programação modular
<i>doc</i>	qualidade de documentação
<i>atit</i>	atitudes do programador no trabalho
<i>inter</i>	qualidade de interface produzida

Tabela C.1: Atributos Relacionados ao Desempenho do Programador

O conjunto de treinamento é mostrado na Figura C.4, pg. 179. Nela, os exemplos assinalados com * estão incompletos: o primeiro deles não tem classe associada e o segundo não tem valor associado ao atributo *temp*.

Tanto o Assistant Professional, quanto a implementação do nosso ambiente TDIDT disponível para experimentos, foram utilizados para a construção da árvore de decisão, usando o conjunto de treinamento da Tabela C.4.

¹Versão comercial do Assistant 86.

²Exemplo apresentado no *Seminar on Inductive Learning and Knowledge Acquisition* por Dr. N. Lavrač, em Novembro 1992, junto ao ICMSC/USP, para treinamento no Assistant Professional.

atributo	valor(abreviatura)	atributo	valor(abreviatura)
<i>gdifícil</i>	fácil(fa) médio(me) difícil(dif) muito difícil(mdif)	<i>inic</i>	insuficiente(insuf) normal(nor) alta(alta)

atributo	valor(abreviatura)	atributo	valor(abreviatura)
<i>temp</i>	excedeu(ex) em tempo(et) antes(at)	<i>creat</i>	nenhuma(nao) normal(nor) novos algoritmos(na) novas técnicas(nt)

atributo	valor(abreviatura)	atributo	valor(abreviatura)
<i>mod</i>	ruim(ruim) padrão(pad) bom(bom)	<i>doc</i>	ruim(ruim) padrão(pad) bom(bom) ótima(ot)

atributo	valor(abreviatura)	atributo	valor(abreviatura)
<i>atit</i>	ruim(ruim) média(me) boa(boa) muito boa(mboa)	<i>inter</i>	ruim(ruim) normal(nor) bem feita(bf)

Tabela C.2: Atributos e seus Possíveis Valores

Quando do uso do Assistant, todos os 38 exemplos do conjunto participaram da construção da árvore, uma vez que este sistema trata dados incompletos. Quando atributos têm valor ausente, a instância de treinamento é replicada tantas vezes quantos forem os possíveis valores do atributo. Quando a classe está ausente, a instância é rotulada com a classe mais provável. O Assistant Professional usa também tanto a pré-poda quanto a pós-poda (ver Seção 5.7). A árvore obtida pelo Assistant está mostrada na Figura C, pg. 183.

Para a utilização do ambiente experimental TDIDT, os dois exemplos assinalados com ★ foram eliminados. O conjunto de treinamento é apresentado ao sistema como uma lista com 36 elementos, onde cada elemento por sua vez é também uma lista que representa um exemplo. A estrutura Prolog que representa uma árvore de decisão é aquela

classe		abreviada como
muito negativa	(-20% de salário)	mne
negativa	(-10% de salário)	ne
normal		nor
positiva	(+10% de salário)	po
muito positiva	(+20% de salário)	mpo

Tabela C.3: Classes nas quais programadores são classificados

gdificult	inic	temp	creat	mod	doc	atit	inter	classe
fa	nor	et	nao	bom	bom	me	bf	mpo
fa	insuf	ex	nao	ruim	ruim	ruim	ruim	mne
fa	nor	et	nao	ruim	pad	me	nor	nor
mdif	alta	ex	na	bom	ot	mboa	bf	mpo
mdif	insuf	ex	nao	ruim	ruim	ruim	ruim	mne
mdif	insuf	ex	nao	ruim	ruim	me	nor	nor
me	alta	et	nt	bom	ot	boa	bf	mpo
me	nor	et	nor	pad	pad	me	nor	nor
me	insuf	ex	nao	ruim	ruim	ruim	ruim	mne
dif	alta	ex	nt	pad	pad	boa	bf	po
fa	insuf	et	nao	ruim	ruim	me	nor	ne
dif	alta	ex	nt	bom	bom	mboa	bf	po
dif	nor	ex	nor	pad	pad	me	nor	nor
me	nor	et	nao	pad	pad	ruim	nor	nor
dif	alta	et	na	bom	bom	mboa	bf	po
fa	nor	et	nao	pad	pad	boa	bf	nor
fa	insuf	et	nao	ruim	pad	me	nor	ne
fa	nor	et	nor	bom	bom	boa	bf	po
me	nor	et	nor	pad	bom	boa	bf	po
fa	nor	et	nao	bom	bom	me	bf	mpo
dif	nor	et	nor	bom	ruim	me	nor	nor
* mdif	alta	ex	na	pad	ruim	me	nor	
me	nor	ex	nao	pad	ruim	boa	bf	ne
fa	insuf	ex	nao	ruim	ruim	ruim	ruim	mne
* mdif	alta		nt	bom	bom	boa	bf	mpo
mdif	alta	at	nor	pad	ruim	mboa	bf	po
mdif	nor	et	nt	bom	ot	boa	bf	mpo
me	nor	et	nao	pad	pad	ruim	nor	nor
mdif	nor	ex	nt	bom	pad	me	nor	mpo
mdif	alta	ex	nor	ruim	ruim	mboa	nor	nor
dif	nor	ex	nor	bom	pad	boa	bf	po
fa	nor	et	nao	bom	bom	me	bf	mpo
fa	alta	et	nor	pad	pad	boa	ruim	nor
me	nor	et	na	pad	bom	ruim	nor	nor
mdif	nor	et	nor	bom	ot	me	bf	po
me	insuf	ex	nor	pad	pad	boa	bf	mne
me	alta	ex	nao	pad	pad	ruim	nor	po
mdif	alta	et	na	bom	ot	mboa	bf	mpo

Tabela C.4: Conjunto de Treinamento

mostrada na Seção 5.10, pg. 51.

Na interrogação Prolog, apresentada a seguir, a variável X é instanciada com a lista de listas que representa o conjunto de treinamento e a variável Y é instanciada com a lista dos atributos usados na descrição do conjunto de treinamento. Como já mencionado, o predicado `forme_arvore/4` é o responsável pela construção da árvore, que será instanciada com a variável Arv e a variável Flag indica se a árvore é exata ou não.

```
?- exemplos(X),atributos(Y),forme_arvore(X,Y,Arv,Flag)
X =
[[gdific(fa),inic(nor),temp(et),creat(nao),mod(bom),doc(bom),atit(me),inter(bf),classe(mpo)),
 [gdific(fa),inic(insuf),temp(ex),creat(nao),mod(ruim),doc(ruim),atit(ruim),inter(ruim),classe(mne))],
 .....

 [gdific(me),inic(alta),temp(ex),creat(nao),mod(pad),doc(pad),atit(ruim),inter(nor),classe(po)),
 [gdific(mdif),inic(alta),temp(et),creat(na),mod(bom),doc(ot),atit(mboa),inter(bf),classe(mpo)]]

Y = [gdific,inic,temp,creat,mod,doc,atit,inter]

Arv =
```

```

mod - [bom : creat -
      [na : gdific -
        [mdif : mpo,
          dif : po
        ],
        nao : mpo,
        nt : gdific -
          [me : mpo,
            mdif : mpo,
            dif : po
          ],
        nor : doc -
          [bom : po,
            pad : po,
            ruim : nor,
            ot : po
          ]
      ],
      ruim : atit -
        [ruim : mne,
          mboa : nor,
          me : gdific -
            [fa : inic -
              [nor : nor,
                insuf : ne
              ],
              mdif : nor
            ]
          ],
        pad : inic -
          [nor : doc -
            [pad : nor,
              ruim : ne,
              bom : creat -
                [na : nor,
                  nor : po
                ]
            ],
            alta : gdific -
              [fa : nor,
                mdif : po,
                me : po,
                dif : po
              ],
            insuf : mne
          ]
      ]
]

```

Flag = exata

Como pode ser visto, a árvore de decisão gerada pela implementação TDIDT, utilizando os 36 exemplos e sem o uso do mecanismo de poda, é razoavelmente mais complexa do que aquela gerada pelo Assistant. Enquanto o Assistant gerou uma árvore com 14 nós, sendo 8 folhas, a implementação TDIDT gerou uma árvore com 36 nós, sendo 24 folhas.

A experiência seguinte, por nós realizada, foi utilizar o mecanismo de poda da Redução do Erro, articulado à implementação TDIDT. Para o uso do método de poda, o conjunto de treinamento de 36 exemplos foi dividido em dois conjuntos, com 24 e 12 exemplos respectivamente (que correspondem a 70% e 30%, respectivamente, do número de exemplos no conjunto original, que é a proporção considerada adequada para esse tipo de

experimento). O conjunto com 24 exemplos é, então, considerado o novo conjunto de treinamento e o conjunto com 12 exemplos, o conjunto de teste, necessário para a realização da poda. A árvore foi construída pelo predicado `forme_arvore/4`, usando 24 exemplos, e instanciada com a variável `Arv`. O conjunto de teste, instanciado através do predicado `exemplosp/1` é então utilizado para criar a *árvore estatística*, `AE`, usando o predicado `faz_arvore_estatistica/3`. É importante notar que dos 12 exemplos do conjunto de teste, um deles não é coberto pela árvore `Arv` — esse exemplo é eliminado do conjunto de teste. Como nem todas as subárvores da árvore estatística são cobertas por exemplos do conjunto de teste (lista de estatística é a lista vazia), antes do processo de poda se inicializar, essas subárvores são sumariamente podadas pelo predicado `limpa_no/2`. A poda utilizando o método da Redução do Erro é efetivamente realizada pelo predicado `poda_reduced_geral/2`, que obtém a árvore podada e a instancia com a variável `AP`. Como a árvore podada ainda carrega em cada nó as frequências de exemplos/classe, esses valores são retirados através do predicado `muda_notacao/2`, que efetivamente instancia a árvore podada pelo método da redução do Erro com a variável `APN`. A interrogação Prolog correspondente, bem como as respostas obtidas, são mostradas a seguir.

```
?- exemplos24(X),atributos(Y),forme_arvore(X,Y,Arv,F),
   exemplosp(E),faz_arvore_estatistica(Arv,E,AE),limpa_no(AE,AEL),
   poda_reduced_geral(AEL,AP), muda_notacao(AP,APN).

X =
[[gdific(dif),inic(nor),temp(ex),creat(nor),mod(pad),doc(pad),atit(me),inter(nor),classe(nor)),
 .....

[gdific(me),inic(insuf),temp(ex),creat(nor),mod(pad),doc(pad),atit(boa),inter(bf),classe(mne)]]

Y = [gdific,inic,temp,creat,mod,doc,atit,inter]

Arv =
mod -
  [pad : inic -
    [nor : doc -
      [pad : nor,
        ruim : ne,
        bom : creat -
          [na : nor,
            nor : po
          ]
      ],
      alta : gdific -
        [fa : nor,
          mdif : po,
          me : po
        ],
      insuf : mne
    ],
    ruim : atit -
      [me : ne,
        ruim : mne,
        mboa : nor
      ],
    bom : creat -
      [nor : doc -
        [bom : po,
          pad : po,
          ruim : nor,
```



```

        ot : po
    ],
    nt : mpo,
    na : gdific -
        [dif : po,
        mdif : mpo
        ],
    nao : mpo
]
]
F = exata

E =
[gdific(fa),inic(insuf),temp(ex),creat(nao),mod(ruim),doc(ruim),atit(ruim),inter(ruim),classe(mne)],
.....
[gdific(mdif),inic(alta),temp(ex),creat(na),mod(bom),doc(ot),atit(mboa),inter(bf),classe(mpo)]]

AE =
mod / [mpo : 3,mne : 3,nor : 3,ne : 1,po : 1] -
    [pad : inic / [mne : 1] -
        [nor : doc / [mne : 1] -
            [pad : nor / [mne : 1],
            ruim : ne / [],
            bom : creat / [] -
                [na : nor / [],
                nor : po / []
                ],
            ],
        ],
        alta : gdific / [] -
            [fa : nor / [],
            mdif : po / [],
            me : po / []
            ],
        ],
        insuf : mne / []
    ],
    ruim : atit / [mne : 2,nor : 3,ne : 1] -
        [me : ne / [nor : 2,ne : 1],
        ruim : mne / [mne : 2,nor : 1],
        mboa : nor / []
        ],
    bom : creat / [mpo : 3,po : 1] -
        [nor : doc / [] -
            [bom : po / [],
            pad : po / [],
            ruim : nor / [],
            ot : po / []
            ],
        ],
        nt : mpo / [mpo : 1,po : 1],
        na : gdific / [mpo : 1] -
            [dif : po / [],
            mdif : mpo / [mpo : 1]
            ],
        ],
        nao : mpo / [mpo : 1]
    ]
]

AEL =
mod / [mpo : 3,mne : 3,nor : 3,ne : 1,po : 1] -
    [pad : inic / [mne : 1] -
        [nor : doc / [mne : 1] -
            [pad : nor / [mne : 1]
            ],
        ],
    ],
    ruim : atit / [mne : 2,nor : 3,ne : 1] -
        [me : ne / [nor : 2,ne : 1],

```

```

        ruim : mne / [mne : 2,nor : 1]
    ],
    bom : creat / [mpo : 3,po : 1] -
        [nt : mpo / [mpo : 1,po : 1],
        na : gdific / [mpo : 1] -
            [mdif : mpo / [mpo : 1]
            ],
        nao : mpo / [mpo : 1]
    ]
]

AP =
mod / [mpo : 3,mne : 3,nor : 3,ne : 1,po : 1] -
    [pad : mne / [mne : 1],
    ruim : nor / [mne : 2,nor : 3,ne : 1],
    bom : mpo / [mpo : 3,po : 1]
]

APB =
mod - [pad : mne,ruim : nor,bom : mpo]

```

Pode ser observado que a atuação da poda foi bem drástica, no sentido que a árvore resultante, mostrada na Figura C.2, tem apenas 4 nós, sendo 3 deles folhas. É importante observar que, mesmo implementando basicamente o mesmo algoritmo de construção de árvores, o Assistant Professional e implementação TDIDT produzem resultados completamente diversos, devido, principalmente, à maneira como tratam os dados.

Figura C.1: Árvore Gerada pelo Assistant Professional

Figura C.2: Árvore Podada Gerada pelo TDIDT

Bibliografia

- [Aha 92] Aha, D.W. *Relating Relational Learning Algorithms*. S.H. Muggleton (ed.), Inductive Logic Programming, Academic Press, pp 233-259.
- [Arity 91] Arity Prolog, V. 6.00/91, Arity Corporation, Mass., 1991.
- [Blumer 87] Blumer, A.; Ehrenfeucht, A.; Haussler, D.; Warmuth, M.K. *Occam's Razor*. Information Processing Letters 24, 1987, pp 377-380.
- [Boose 89] Boose, J.H. *A Knowledge Acquisition Techniques and Tools*. Knowledge Acquisition 1, 1989, pp 3-37.
- [Boswell 90] Boswell, T. *Manual for NewId version 4.1* The Turing Institute, TI/P2154/RAB/4/2.3, January 1990.
- [Boswell 90a] Boswell, T. *Manual for CN2 version 4.1* The Turing Institute, TI/P2154/RAB/4/1.3, January 1990.
- [Bratko 89] Bratko, I. *Machine Learning*. Human and Machine Problem Solving, K.Gilhooly (ed.), Plenum Press, New York, 1989.
- [Bundy 85] Bundy, A.; Silver, B.; Plummer, D. *An Analytical Comparison of Some Rule-learning Programs*. Artificial Intelligence, 27, pp 137-181.
- [Buntine 88] Buntine, W. *Generalized Subsumption and its Applications to Induction and Redundancy*. Artificial Intelligence 36, 2, pp 149-176.
- [Breiman 84] Breiman, L.; Freidman, J.; Olshen, R.; Stone, C. *Classification and Regression Trees*. Wodsworth International, CA, 1984.
- [Cameron-Jones 93] Cameron-Jones, R.M.; Quinlan, J.R. *Efficient Top-down Induction of Logic Programs*. Technical Report (Draft version), Basser Department of Computer Science, The University of Sydney, Australia, November 1993, 30 pg.
- [Carbonell 83] Carbonell, J.G.; Michalski, R.S.; Mitchell, T.M. *An Overview of Machine Learning*. R.S. Michalski; J.G. Carbonell & T.M. Mitchell (eds.), Machine Learning: An Artificial Intelligence Approach, Vol. I, Tioga Publishing Co., Palo Alto, CA, 1983, pp 3-24.

- [Carbonell 89] Carbonell, J.G. *Introduction: Paradigms for Machine Learning*. Artificial Intelligence 40, 1989, pp 1-9.
- [Clark 87] Clark, P.; Niblett, T. *Induction in Noisy Domains*. I. Bratko & N. Lavrač (eds.), Progress in Machine Learning, Sigma Press, 1987, pp 11-30.
- [Clark 89] Clark, P.; Niblett, T. *The CN2 Induction Algorithm*. Machine Learning 3, 1989, pp 261-283.
- [Clark 91] Clark, P.; Boswell, R. *Rule Induction with CN2: Some Recent Improvements*. In: Proceedings of the Fifth European Working Session on Learning, Springer Verlag, Berlin, pp 151-163.
- [Castiñeira 90] Castiñeira, M.I. *Aprendizado de Máquina por Exemplos Usando Árvores de Decisão*. Dissertação de Mestrado, ICMSC - USP, São Carlos, SP, 1990.
- [Castiñeira 90a] Castiñeira, M.I.; Monard, M.C. *Análise de um Método de Poda para Árvores Indutivas de Decisão*. Anais do VII Simpósio Brasileiro de Inteligência Artificial, Paraíba, Novembro de 1990, pp 17-27.
- [Castiñeira 91] Castiñeira, M.I.; Monard, M.C.; Nicoletti, M.C. *Aprendizado de Máquina: Descrição e Implementação de um Algoritmo Geral para a Construção de Árvores de Decisão*. Notas Nº 98, ICMSC-USP, São Carlos, SP, 1991.
- [Cestnik 87] Cestnik, B.; Kononenko, I.; Bratko, I. *Assistant 86: A Knowledge-Elicitation Tool for Sophisticated Users*. Progress in Machine Learning, I. Bratko & N. Lavrač (eds.), Sigma Press, 1987, pp 31-45.
- [Cestnik 90] Cestnik, B. *Estimating Probabilities: A Crucial Task in Machine Learning*. In: Proceedings of the Ninth European Conference on Artificial Intelligence, Pitman, London, pp 147-149, 1990.
- [Cestnik 91] Cestnik, B. *Estimating Probabilities in Machine Learning*. Ph.D. thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia, 1991.
- [Cohen 82] Cohen, P.R.; Feigenbaum, E.A. (eds.) *The Handbook of Artificial Intelligence*. Vol. III, Addison-Wesley Publishing Company, CA, 1982.
- [Cohen 93] Cohen, W.W. *Learnability of Restricted Logic Programs*. In: Proceedings of the Third International Workshop on Inductive Logic Programming — ILP'93, Bled, Slovenia, April 1993, pp 1-3.
- [DeJong 86] DeJong, G.; Mooney, R.J. *Explanation-Based Learning: An Alternative View*. Machine Learning 2, Vol. 1, 1986, pp 145-176.

- [Dolšak 92] Dolšak, B.; Muggleton, S.H. *The Application of Inductive Logic Programming to Finite-Element Mesh Design*. S.H. Muggleton (ed.) *Inductive Logic Programming*, Academic Press, 1992, pp 453-472.
- [Dolšak 92a] Dolšak, B.; Jezernik, A.; Bratko, I. *A Knowledge Base for Finite Element Mesh Design*. In: *Proceedings of the International School for the Synthesis of Expert Knowledge — ISSEK Workshop'92*, Bled, Slovenia, September 3-5, 1992.
- [Drastal 89] Drastal, G.; Raatz, S.; Czako, G. *Induction in an Abstract Space*. In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, August, Morgan Kaufmann, 1989,
- [Džeroski 91] Džeroski, S. *Handling Noise in Inductive Logic Programming*. Master's thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Slovenia, 1991.
- [Džeroski 92] Džeroski, S.; Muggleton, S.H.; Russell, S. *PAC - Learnability of Determinate Logic Programs*. In: *Proceedings of the ISSEK Workshop'92*, Slovenia, September 1992.
- [Edinburgh 87] *Edinburgh Prolog.*, AI Applications Institute, University of Edinburgh, 1987.
- [Eshelman 87] Eshelman, L.; Ehret, D.; McDermott, J.; Tan, M. *MOLE: A Tenacious Knowledge Acquisition Tool*. *International Journal of Man-Machine Studies*, Vol. 26, Number 1, 1987, pp 41-54.
- [Flann 86] Flann, N.; Dietterich, T. *Selecting Appropriate Representation for Learning from Examples*. In: *Proceedings of the AAAI - 86*, Morgan Kaufmann, 1986, pp 460-466.
- [Feigenbaum 80] Feigenbaum, E.A. *Knowledge Processing: From File Servers to Knowledge Servers*. *Applications of Expert Systems - Vol. 1* - J.R. Quinlan (ed.) Turing Institute Press in association with Addison-Wesley Publishing Company, 1980.
- [Feng 90] Feng, C.; Muggleton, S.H. *GOLEM 1.0 User's Commands.*, 1990.
- [Grobelnik 92] Grobelnik, M. *MARKUS: An Optimized Model Inference System*. *Second Generation Knowledge Acquisition Systems*. In: *Proceedings of the Workshop on Logical Approaches to Machine Learning, Tenth European Conference on Artificial Intelligence*, Vienna, Austria, 1992.
- [Hayes-Roth 87] Hayes-Roth, F. *Expert Systems*. *Encyclopedia of Artificial Intelligence* S. C. Shapiro; D. Eckroth; G. A. Vallasi (eds.), John Wiley & Sons, New York, 1987, pp 287-298.
- [Hogger 90] Hogger, C.J. *Essentials of Logic Programming*. Clarendon Press, Oxford, 1990.

- [Hooke 89] Hooke, A.T. *Multiple Splitting of Numerical Variables in Inductive Learning*. TIRM-89-037, Turing Institute Press, Glasgow, 1989.
- [Hunt 66] Hunt, E.B.; Marin, J.; Stone, P.J. *Experiments in Induction*. Academic Press, New York, 1966.
- [Kietz 93] Kietz, J-U. *Some Lower Bounds for the Computational Complexity of Inductive Logic Programming*. Lecture Notes in Artificial Intelligence 667, P.B. Bradzil (ed.), Springer-Verlag, 1993, pp 115-123.
- [Kietz 92] Kietz, J-U.; Wrobel, S. *Controlling the Complexity of Learning in Logic through Syntactic and Task-Oriented Models*. Inductive Logic Programming, S.H. Muggleton (ed.), Academic Press, 1992.
- [King 92] King, R.D.; Muggleton, S.H.; Lewis, R.A.; Sternberg, M.J.E. *Drug Design by Machine Learning: The Use of Inductive Logic Programming to Model the Structure-activity Relationships of Trimethoprim Analogues Binding to Dihydrofolate Reductase*. In: Proceedings of the International Workshop on Inductive Logic Programming — ILP'92, Tokyo, Japan, June 6-7, 1992, pp 202-227.
- [Kononenko 85] Kononenko, I. *The Design of the Assistant Inductive Learning System*. E. Kardely University, M.Sc. thesis, 1985.
- [Kowalski 87] Kowalski, R.A.; Hogger, C.J. *Logic Programming*. S.C. Shapiro; D. Eckroth; G.A. Vallasi (eds.), Encyclopedia of Artificial Intelligence, John Wiley & Sons, New York, 1987, pp 544-558.
- [Lavrač 89] Lavrač, N. *Meta-level Architecture of a Second Generation Knowledge Acquisition System*. In: Proceedings of the EWSL 89, 1989, pp 99-109.
- [Lavrač 90] Lavrač, N. *Principles of Knowledge Acquisition in Expert Systems*. Ph.D. thesis, Faculty of Technical Sciences, University of Maribor, Maribor, Slovenia.
- [Lavrač 92] Lavrač, N.; Džeroski, S. *Inductive Learning of Relations from Noisy Examples*. S.H. Muggleton (ed.) Inductive Logic Programming, Academic Press, 1992, pp 495-516.
- [Lavrač 93] Lavrač, N. *Personal Communication*. December 1993.
- [Lavrač 94] Lavrač, N.; Džeroski, S. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, London, 1994.
- [Matheus 90] Matheus, C.J. *Feature Construction: An Analytic Framework and an Application to Decision Trees*. Ph.D. thesis, University of Illinois at Urbana Champaign, Urbana, Illinois, USA, 1990.
- [Martins 93] Martins, C.A. *Uso de Árvores de Decisão na Geração de Hipóteses Indutivas em Aprendizado de Máquina*. Monografia de Exame de Qualificação, ICMSC-USP, setembro 1993, 56 pg.

- [Martins 94] Martins, C.A.; Monard, M.C. *Árvores de Decisão Binárias: Descrição e Implementação de Algoritmo para a Construção e Poda no Aprendizado Indutivo de Máquina*. Relatórios Técnicos do ICMSC, Nº 17, fevereiro 1994, 35 pg.
- [McCluskey 56] McCluskey, E.J. *Minimization of Boolean Functions*. Bell System Tech. J., 35, Number 5, November 1956, pp 1417-1444.
- [Michalski 83] Michalski, R.S. *A Theory and Methodology of Inductive Learning*. R.S. Michalski; J.G. Carbonell & T.M. Mitchell (eds.), Machine Learning: An Artificial Intelligence Approach, Vol. I, Tioga Publishing Co., Palo Alto, CA, 1983, pp 83-134.
- [Michalski 93] Michalski, R.S.; Tecuci, G. *Multistrategy Learning*. Tutorial T15, IJ-CAI 1993, 1993.
- [Mingers 87] Mingers, J. *Expert Systems — Rule Induction with Statistical Data*. Journal of the Operational Research Society, 38, 1987, pp 39-47.
- [Mingers 89] Mingers, J. *An Empirical Comparison of Selection Measures for Decision-Tree Induction*. Machine Learning 3, 1989, pp 319-342.
- [Mingers 89a] Mingers, J. *An Empirical Comparison of Pruning Methods for Decision Tree Induction*. Machine Learning 4, 1989, pp 227-243.
- [Mitchell 82] Mitchell, T.M. *Generalization as Search*. Artificial Intelligence 18, 1982, pp 203-226.
- [Monard 91] Monard, M.C.; Rodrigues, S.R. *An Environment for Developing Expert Systems's Shells*. In: Proceedings of Expert Systems World Congress, J. Liebowitz (ed.), Pergamon Press, Vol. 3, 1991, pp 1757-1765.
- [Monard 93] Monard, M.C.; Nicoletti, M.C. *An Improved Implementation of Wang's Theorem Prover Algorithm*. Advances in Modelling & Analysis, Vol. 25, Nº 1, 1993, pp 55-64.
- [Monard 93a] Monard, M.C.; Nicoletti, M.C. *Programas Prolog para Processamento de Listas e Aplicações*. Notas Didáticas do ICMSC-USP, Nº 7, janeiro 1993, 71 pg.
- [Monard 93b] Monard, M.C.; Nicoletti, M.C. *Técnicas Avançadas de Programação Prolog para Tratamento de Árvores*. Notas Didáticas do ICMSC-USP, Nº 8, fevereiro 1993, 51 pg.
- [Monard 94] Monard, M.C.; Martins, C.A.; Nicoletti, M.C. *Projeto e Implementação de Estratégias de Construção de Features Utilizando Árvores de Decisão Binárias*. (em preparação).
- [Morales 92] Morales, E. *Testing the Applicability of ILP System*. In: Proceedings of the International Workshop on Inductive Logic Programming — ILP'92, Tokyo, Japan, June 6-7, 1992, pp 260-270.

- [Morik 93] Morik, K.; Wrobel, S.; Kietz, J.-U.; Emde, W. *Knowledge Acquisition and Machine Learning*. Academic Press, 1993.
- [Mozetic 85] Mozetič, I. *NEWGEM: Program for Learning from Examples, Technical Documentation and User's Guide*. Reports of Intelligent Systems Group UIUCDCS-F-85-949, Department of Computer Science, University of Illinois, Urbana Champaign, 1985.
- [Muggleton 87] Muggleton, S.H. *Duce, an Oracle Based Approach to Constructive Induction*. In: Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, Morgan Kaufmann, CA, 1987, pp 287-292.
- [Muggleton 88] Muggleton, S.H.; Buntine, W. *Machine Invention of First-order Predicates by Inverting Resolution*. In: Proceedings of the Fifth International Conference on Machine Learning, Morgan Kaufmann, CA, pp 339-352.
- [Muggleton 90a] Muggleton, S.H. *Inductive Acquisition of Expert Knowledge*. Turing Institute Press, Glasgow, 1990.
- [Muggleton 90] Muggleton, S.H.; Feng, C. *Efficient Induction of Logic Programs*. TIRM-90-044, Turing Institute Press, Glasgow, October 1990.
- [Muggleton 91] Muggleton, S.H. *Inductive Logic Programming*. New Generation Computing, Vol. 8, 1991, pp 295-318.
- [Muggleton 92] Muggleton, S.H.; King, R.D.; Sternberg, M.J.E. *Protein Secondary Structure Prediction Using Logic*. In: Proceedings of the International Workshop on Inductive Logic Programming — ILP'92, Tokyo, Japan, June 6-7, 1992, pp 228-259.
- [Muggleton 92a] Muggleton, S.H.; Buntine W. *Machine Invention of First-Order Predicates by Inverting Resolution*. In: S.H. Muggleton (ed.) *Inductive Logic Programming*, Academic Press, 1992. pp 262-280.
- [Muggleton 93] Muggleton, S.H. *Inductive Logic Programming: Derivations, Successes and Shortcomings*. P.B. Bradzil (ed.), Lecture Notes in Artificial Intelligence 667, 1993.
- [Murthy 93] Murthy, S.; Kasif, S.; Salzberg, S.; Beigel, R. *OC1: Randomized Induction of Oblique Decision Trees*. AAAI 93, pp 122-128, 1993.
- [Niblett 88] Niblett, T. *Constructing Decision Trees in Noisy Domains*. I. Bratko, N. Lavrač (eds.), *Progress in Machine Learning*, Sigma Press, England, 1988.
- [Nicoletti 92] Nicoletti, M.C.; Castiñeira, M.I.; Monard, M.C. *Descrição e Implementação dos Mecanismos de Janela e Poda em Árvores de Decisão no Aprendizado Indutivo de Máquina*. Notas do ICMSC-USP, Nº 110, maio 1992, 37 pg.

- [Nicoletti 92a] Nicoletti, M.C.; Monard, M.C. *Construção Automática de Features no Aprendizado Indutivo de Funções Booleanas*. ILTC, Série de C & T, Vol. 1, Nº 8, junho 1992, pp 1-20.
- [Nicoletti 92b] Nicoletti, M.C.; Bezerra, P.C.; Monard, M.C. *Learning Boolean Function Using Prime-Implicants as Features*. In: Proceedings of the International Multiconference on Neural Networks, Modeling & Simulation, Knowledge Processing. Chicago, USA, September 1992, pp 177-183.
- [Nicoletti 92c] Nicoletti, M.C.; Bezerra, P.C. *Aprendizado de Máquina: como Acelerar o Processo de Aquisição de Conhecimento no Desenvolvimento de Sistemas Especialistas*. Anais Eletrônicos do XXV Congresso Nacional de Informática, 19-23 Outubro 1992, Recife-PE.
- [Nicoletti 93] Nicoletti, M.C.; Monard, M.C. *Inverting Resolution: Plotkin's Least General Generalization*. Notas do ICMSC-USP, Nº 130, março 1993, 30 pg.
- [Nicoletti 93a] Nicoletti, M.C.; Monard, M.C. *Herbrand Interpretation, Model and Least Model within the Framework of Logic Programming*. Notas do ICMSC-USP, Série Computação, Nº 2, junho 1993, 30 pg.
- [Nicoletti 93b] Nicoletti, M.C.; Monard, M.C. *Empirical Evaluation of Two Pruning Methods Applied to Constructive Learning of Boolean Functions*. Anales del Primer Congreso Internacional de Informatica, Computacion y Teleinformatica, INFORMATICA'93, Mendoza, Argentina, Junho 1993, pp 265-277.
- [Nicoletti 93c] Nicoletti, M.C.; Monard, M.C. *The Role of Description Languages in Inductive Concept Learning*. I Simpósio Brasileiro de Automação Inteligente, Rio Claro, SP, 8-10 Setembro 1993, pp 403-412.
- [Nicoletti 93d] Nicoletti, M.C.; Monard, M.C. *Learning Horn Clauses Using the ILP System GOLEM.*, Notas do ICMSC-USP, Série Computação, Nº 4, Novembro 1993, 35 pg.
- [Nicoletti 94] Nicoletti, M.C.; Monard, M.C. *Limiting the Background Knowledge in Inductive Logic Programming*. Notas do ICMSC-USP, Série Computação, Nº 5, Janeiro 1994, 18 pg.
- [Nicoletti 94a] Nicoletti, M.C.; Monard, M.C. *Learning Restricted Horn Clauses: Some Considerations on the ij-Determination Concept*. Notas do ICMSC-USP, Série Computação, Nº 9, Abril 1994, 16 pg.
- [Nicoletti 94b] Nicoletti, M.C.; Monard, M.C. *Inducing Logic Programs with the Learning System GOLEM*. Trabalho submetido ao II International Congress and Exposition in Informatics, Mendoza, Argentina.
- [Nicoletti 94c] Nicoletti, M.C.; Monard, M.C. *Reviewing ij-Determination and Related Concepts in Inductive Logic Programming*. Trabalho submetido à XX Conferencia Latinoamericana de Informatica, Monterrey, Mexico.

- [O'Keefe 90] O'Keefe, R.A. *The Craft of Prolog*. MIT Press Series in Logic Programming. E. Shapiro (ed.), 1990.
- [Pagallo 89] Pagallo, G.; Haussler, W. *Two Algorithms that Learn DNF by Discovering Relevant Features*. In: Proceedings of the Sixth International Workshop on Machine Learning, Morgan Kaufmann Publishers, June 1989, pp 119-123.
- [Pagallo 89a] Pagallo, G.; Haussler, W. *Learning DNF by Decision Trees*. In: Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, MI, August 1989, Morgan Kaufmann Publishers, pp 639-644.
- [Pagallo 90] Pagallo, G.; Haussler, W. *Boolean Feature Discovery in Empirical Learning*. Machine Learning 5, 1990, pp 71-99.
- [Pazzani 92] Pazzani, M.J.; Clifford, A.B.; Silverstein, G. *An Information-Based Approach to Integrating Empirical and Explanation-Based Learning*. S.H. Muggleton (ed.) Inductive Logic Programming, Academic Press, 1992, pp 373-394.
- [Plotkin 71] Plotkin, G.D. *Automatic Methods of Inductive Inference*. Ph. D. thesis, Edinburgh University, August 1971.
- [Plotkin 71a] Plotkin, G.D. *A Further Note on Inductive Generalization*. B.Meltzer, D.Michie (eds.), Machine Intelligence, American Elsevier, 1971, pp 101-124.
- [Quine 52] Quine, W.V. *The Problem of Simplifying Truth Functions*. Am. Math. Monthly, 59, Number 8, October 1952, pp 521-521.
- [Quinlan 79] Quinlan, J.R. *Discovering Rules by Induction from Large Collections of Examples*. Expert Systems in the Micro Electronic Age, D.Michie (ed.) Edinburgh University Press, Edinburgh, 1979, pp 168-201.
- [Quinlan 83] Quinlan, J.R. *Learning Efficient Classification Procedures and their Application to Chess End Games*. R.S. Michalski; J.G. Carbonell & T.M. Mitchell (eds.), Machine Learning: An Artificial Intelligence Approach, Vol. I, Tioga Publishing Co., Palo Alto, CA, 1983, pp 463-482.
- [Quinlan-86] Quinlan, J.R.; Compton, P.J.; Horn, K.A.; Lazarus, L. *Inductive Knowledge Acquisition: a Case Study*. In: Proceedings of the 2nd Australian Conference on Applications of ES, Sydney, 1986, pp 183-203.
- [Quinlan 86a] Quinlan, J.R. *Induction of Decision Trees*. Machine Learning 1, 1986, pp 81-106.
- [Quinlan 87] Quinlan, J.R. *Simplifying Decision Trees*. International Journal of Man-Machine Studies 27, 1987, pp 221-234.

- [Quinlan 89] Quinlan, J.R.; Rivest, R.L. *Inferring Decision Trees Using the Minimum Description Length Principle*. Information and Computation 80, 1989, pp 227-248.
- [Quinlan 90] Quinlan, J.R. *Learning Logical Definitions from Relations*. Machine Learning, 5, 1990, pp 239-266.
- [Quinlan-91] Quinlan, J.R. *Knowledge Acquisition from Structured Data*. IEEE Expert, December 1991, pp 32-37.
- [Quinlan-91a] Quinlan, J.R. *Determinate Literals in Inductive Logic Programming*. In: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Sydney, 1991, pp 746-750.
- [Quinlan 93] Quinlan, J.R.; Cameron-Jones, R.M. *FOIL - An Overview*. FOIL Version 5.0, 1993.
- [Quinlan 93a] Quinlan, J.R.; Cameron-Jones, R.M. *FOIL: A Midterm Report*. In: Proceedings of the European Conference on Machine Learning, Springer Verlag, 1993, pp 3-20.
- [Rich 91] Rich, E. ; Knight, K. *Artificial Intelligence*. McGraw-Hill Book Company, New York, 1991.
- [Rissanen 86] Rissanen, J. *Stochastic Complexity and Sufficient Statistics*. Technical Report, IBM Research Laboratory, San Jose, CA, 1986.
- [Robinson 65] Robinson, J. *A Machine-oriented Logic Based on the Resolution Principle*. Journal of the ACM, 12(1), pp 23-41.
- [Rodrigues 93] Rodrigues, S.R.; Monard, M.C. *Sistemas Baseados em Conhecimento: Conceitos Fundamentais e Aplicações*. Anais da 1ª Jornada USP-SUCESU-SP de Informática e Telecomunicações, São Paulo, SP, Junho 1993, pp 355-364.
- [Rodrigues 93a] Rodrigues, S.R.; Rozenfeld, H.; Monard, M.C. *Feature Based Design Interface for a CAPP Environment*. Anales del Primer Congreso Internacional de Informatica, Computacion y Teleinformatica, INFORMATICA'93, Mendoza, Argentina, Junho 1993, pp 33-42.
- [Rouveirol 90] Rouveirol, C.; Puget, J.-F. *Beyond Inversion of Resolution*. In: Proceedings of Seventh Conference of Machine Learning, Morgan Kaufmann, 1990.
- [Rouveirol 91] Rouveirol, C. *ITOU: Induction de Theories en Ordre Un*. PhD Thesis, Laboratoire de Recherche en Informatique, Universite Paris XI, 1991 (em francês).
- [Rouveirol 92] Rouveirol, C. *Extensions of Inversion of Resolution Applied to Theory Completion*. In S.H.Muggleton (ed.), Inductive Logic Programming, Academic Press, 1992, pp 63-86.

- [Sammut 93] Sammut, C. *The Origins of Inductive Logic Programming: A Prehistoric Tale*. In: Proceedings of the 3rd. International Workshop on Inductive Logic Programming, ILP'93, April 1993, Slovenia, pp 127-147.
- [Schlimmer 86] Schlimmer, J.C.; Fisher, D. *A Case Study of Incremental Concept Induction*. In: Proceedings of the Fifth National Conference on Artificial Intelligence, Morgan Kaufmann, 1986, pp 496-501.
- [Shapiro 83] Shapiro, E. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
- [Shaw 88] Shaw, M.L.G. *Problems of Validation in a Knowledge Acquisition System using Multiple Experts*. J. Boose; B. Gaines & M. Linster (eds.), In: Proceedings of the 2nd European Knowledge Acquisition Workshop – EKAW-89, Bonn, Germany, June 1988, pp 5.1-15.
- [Shaw 90] Shaw, M.J.; Gentry, J.A. *Inductive Learning for Risk Classification*. IEEE Expert, February 1990, pp 47-53.
- [Sicstus 93] *Sicstus Prolog.*, Swedish Institute of Computer Science, 1988-1993.
- [Spangler 89] Spangler, S.; Fayyad, U.M.; Uthurusamy, R. *Induction of Decision Trees from Inconclusive Data*. In: Proceedings of the Sixth International Workshop on Machine Learning, Morgan Kaufmann Publishers, Inc., June 1989, pp 146-150.
- [Sthal 93] Sthal, I. *Predicate Invention in ILP - an Overview*. In: Proceedings of the 6th. European Conference on Machine Learning, P. Bradzil (ed.), Lecture Notes in Artificial Intelligence, Springer Verlag, 1993, pp 313-322.
- [Sterling 86] Sterling, L.; Shapiro, E. *The Art of Prolog*. MIT Press Series in Logic Programming, E. Shapiro (ed.), 1986.
- [Simon 83] Simon, H.A. *Why Should Machine Learn?* Machine Learning, Vol. I, Morgan Kaufmann Publishers, Inc., 1983.
- [Sokal 81] Sokal, R.; Rohlf, F. *Biometry*. San Francisco, Freeman, 1981.
- [Thompson 86] Thompson, B.; Thompson, W. *Finding Rules in Data*. BYTE, November 1986, pp 149-158.
- [Thrun 91] Thrun, S.B. et al. *The Monk's Problems: A Performance Comparison of Different Learning Algorithms*. CMU-CS-91-197, Carnegie Mellon University, December 1991.
- [Utgoff 86] Utgoff, P.E. *Shift of Bias for Inductive Concept Learning*. Machine Learning: An Artificial Intelligence Approach, Vol. II, Morgan Kaufmann Publishers Inc., Los Altos, CA, 1986, pp 107-148.

- [Utgoﬀ 88] Utgoﬀ, P.E. *ID5: An Incremental ID3*. In: Proceedings of the Fifth International Conference on Machine Learning, University of Michigan, June 1988, pp 107-120.
- [Utgoﬀ 89] Utgoﬀ, P.E. *Incremental Induction of Decision Trees*. Machine Learning 4, 1989, pp 161-186.
- [Waterman 86] Waterman, D.A. *A Guide to Expert Systems*. Addison-Wesley Publishing Company, 1986.
- [Wielinga 87] Wielinga, B.J.; Bredeweg, B.; Breuker, J.A. *Knowledge Acquisition for Experts Systems.*, Lecture Notes in Artificial Intelligence, J. Siekmann (ed.), Subseries of Lecture Notes in Computer Science, G. Goods & J.Hatmanis (eds.), Vol. 345, 1987, pp 96-121.
- [Winston 75] Winston, P. *Learning Structural Descriptions from Examples*. The Psychology of Computer Vision, P.H.Winston (ed.), McGraw-Hill, New York, 1975, pp 157-209.
- [Wirth 88] Wirth, J.; Catlet, J. *Experiments on the Costs and Benefits of Windowing in ID3*. In: Proceedings of the Fifth International Conference on Machine Learning, University of Michigan, June 88, pp 87-99.
- [Wrobel 87] Wrobel, S. *Higher-order Concepts in a Tractable Knowledge Representation*. In K. Morik, (ed.), In: Proceedings of the 11th German Workshop on Artificial Intelligence – GWAI-87, Informatik-Fachberichte Nr. 152, Springer-Verlag, October 1987, pp 129-138.