HTILDE: TORNANDO ÁRVORES DE DECISÃO RELACIONAIS ESCALÁVEIS PARA GRANDES BASES DE DADOS

Carina Isabel Medeiros Lopes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Gerson Zaverucha, Ph.D.

Prof. Valmir Carneiro Barbosa, Ph.D.

Profª. Bianca Zadrozny, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
JUNHO DE 2008

LOPES, CARINA ISABEL MEDEIROS

HTILDE: Tornando árvores de decisão relacionais escaláveis para grandes bases de dados [Rio de Janeiro] 2008

XIV, 85p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Aprendizado de máquina
2. ILP
3. Inferência estatística
4. Escalabilidade de algoritmos
5. Mineração de grandes bases de dados

I. COPPE/UFRJ II. Título(série)

Aos meu pais, Tereza e Francisco, e ao meu irmão Luis, por todo o apoio e carinho que sempre me deram, sem os quais nada disso seria possível.

Agradecimentos

Agradeço a Deus, pela ajuda e tudo o mais que me deu na vida.

Aos meus pais, Tereza e Francisco, e ao meu irmão Luis, por todo o carinho e apoio que sempre me deram. Por compreenderem a minha ansiedade e, ultimamente, ausência, por confiarem em mim e sempre estarem dispostos a me ajudar. Sem eles, eu não chegaria aonde estou.

Ao meu namorado João, por todo o carinho e apoio, por me escutar quando preciso desabafar, mesmo se for sobre um problema que tive no decorrer do trabalho e ele não entender direito o assunto.

Ao meu orientador, Professor Gerson Zaverucha, pelos conhecimentos transmitidos, por me dedicar o seu tempo, me guiar e acreditar em mim, até mesmo quando eu ficava em dúvidas que as coisas dariam certo.

Ao Professor Hendrik Blockeel, criador do TILDE, por fornecer o código do sistema e permitir que eu pudesse basear nele este trabalho. Agradeço ao Jan Struyf, que me ajudou e muito a enteder o TILDE. Ele me ajudou a usar o sistema, a enteder o código, a modificá-lo, etc. Sempre foi muito solícito e este trabalho seria impossível de ser feito sem a sua ajuda.

Ao Professor Pedro Domingos por tornar o VFDT público, o que possibilitou o esclarecimento de dúvidas a respeito do sistema.

Agradeço Professor Vitor Costa por toda a ajuda e por fornecer a base de dados Cora.

Agradeço também a vários amigos, que de diversas formas me ajudaram no decorrer deste trabalho.

Agradeço a Aline Paes, que é um poço de conhecimento e sabe tudo sobre tudo. E, até mais importante do que isso, está sempre disposta a ajudar, seja me escutando, tirando as minhas dúvidas, ou o que for preciso. Agradeço a Ana Luisa Duboc que, mais do que ninguém, compreende as dificuldades que tive. Ela foi a minha "companheira de tema"por mais da metade do tempo que levei para fazer

o trabalho, me ajudou a entender uma série de coisas, escutou minhas frustações quando alguma coisa dava errado e sempre me apoiou.

Ao Aloísio Pina, que sempre foi solícito e se dispôs a tirar as minhas dúvidas até mesmo na véspera da sua defesa de Tese de Doutorado. Ao Elias Bareinboim, pelas várias conversas e por todo o apoio. Agradeço ao Thiago Cordeiro e ao Pedro Cardoso por me ajudarem a entender o VFDT e tirarem minhas dúvidas em diversos momentos, até mesmo já não estando mais na COPPE há um bom tempo.

A todos os amigos do mestrado e fora dele pela apoio, carinho e compreensão. Aos que sabiam como estavam as coisas só pela minha expressão ou aos que me deram palavras de apoio e incentivo durante estes anos. Foram muitos, não cabe listá-los, mas agradeço muito a todos.

Agradeço a CAPES pela ajuda financeira.

Obrigada a todos que torceram por mim.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

HTILDE: TORNANDO ÁRVORES DE DECISÃO RELACIONAIS ESCALÁVEIS PARA GRANDES BASES DE DADOS

Carina Isabel Medeiros Lopes

Junho/2008

Orientador: Gerson Zaverucha
Programa: Engenharia de Sistemas e Computação

Atualmente, muitas organizações possuem bases de dados com milhões de registros. Uma questão relevante é como extrair informações a partir dessas bases uma vez que, devido a limitações de tempo e até de espaço, os algoritmos tradicionais não podem ser usados. Domingos e Hulten criaram uma metodologia baseada em amostragem para tornar algoritmos de aprendizado de máquina escaláveis para grandes bases de dados. Ela usa o limite de Hoeffding para escolher o número de exemplos que será utilizado pelo algoritmo e foi aplicada a alguns métodos proposicionais, como o VFDT, que é uma árvore de decisão. Outro interesse que surge é o de se utilizar sistemas ILP para aprender modelos a partir destas bases de dados, devido ao caráter relacional das mesmas. Entretanto, sistemas ILP são menos eficientes do que os proposicionais devido ao alto custo de se testar se uma cláusula cobre um exemplo. O TILDE é uma árvore de decisão de lógica de primeira ordem que prova exemplos de maneira eficiente por utilizar o aprendizado a partir de interpretações e os pacotes de cláusulas. O presente trabalho propõe o HTILDE, um sistema ILP escalável para grandes bases de dados baseado no TILDE e no VFDT. O sistema foi testado em duas bases de dados, uma sintética e outra real. Os resultados obtidos mostram que o HTILDE consegue gerar teorias, para bases de dados muito grandes, de forma mais eficiente e sem haver prejuízo para as medidas de qualidade das mesmas.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M. Sc.)

HTILDE: SCALING UP RELATIONAL DECISION TREES FOR LARGE DATABASES

Carina Isabel Medeiros Lopes

June/2008

Advisor: Gerson Zaverucha
Department: Systems Engineering and Computer Science

Nowadays, many organizations have databases with millions of records. An important question is how to extract information from these databases, since traditional machine learning algorithms can not be used, due to time and even space limitations. Domingos and Hulten created a methodology, based on sampling, for scaling up machine learning algorithms for large databases. It uses Hoeffding bound for choosing the number of examples that will be used by the algorithm and it was applied to some propositional methods, like VFDT, which is a decision tree. Also, it would be interesting to use ILP systems to learn models from these databases, due to the relational aspect of them. However, ILP systems are less efficient than proposicional ones due to the high cost of testing whether a clause covers an example. TILDE is a first order logical decision tree which efficiently proves examples by using learning from interpretations and query packs. This work proposes HTILDE, which is an ILP system, based on TILDE and VFDT, able to handle large databases. The system was tested in two datasets, a synthetic one and a real one. The results show that HTILDE generates theories, from very large datasets, more efficiently and without harming their quality measures.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

O aprendizado de máquina é uma parte da inteligência artificial que desenvolve técnicas e algoritmos que visam extrair informação automaticamente a partir de dados.

Uma das áreas de aprendizado de máquina é a programação em lógica indutiva (*ILP - Inductive Logic Programming*). Ela é definida como a interseção de aprendizado indutivo e programação em lógica [29] e tem como objetivo aprender teorias em lógica de primeira ordem a partir de exemplos e de um conhecimento preliminar. A lógica de primeira ordem tem como importante característica conseguir representar, de forma elegante, situações complexas envolvendo vários objetos e as relações entre eles. Devido a isso, sistemas ILP têm um poder de expressividade maior do que sistemas proposicionais, também chamados de métodos atributo-valor, e são capazes de resolver problemas que estes últimos não conseguem [5].

Uma importante questão que tem sido foco de diversos estudos em aprendizado de máquina, de uma forma geral, é o desenvolvimento de algoritmos escaláveis para um conjunto muito grande de exemplos. Esse interesse decorre do fato de, atualmente, muitas organizações possuírem bases de dados muito grandes, com milhões de registros. Grandes *sites* de comércio eletrônico ou operadoras de cartões de crédito, por exemplo, lidam com um número muito grande de transações diariamente e muita informação poderia ser extraída, mas algoritmos tradicionais

de aprendizado de máquina não podem ser utilizados, devido a limitações de tempo e até mesmo de espaço [32] [11].

Uma das principais formas de lidar com grandes bases de dados é considerar apenas uma amostra dos exemplos. O ponto chave desta abordagem é saber escolher uma parte dos exemplos que produza modelos equivalentes ou até melhores do que aqueles que seriam gerados se fosse utilizado todo o conjunto de dados [32]. Friedman, em [14], diz que métodos de amostragem têm uma longa tradição em estatística e podem ser usados na área de mineração de dados, de forma a melhorar os modelos obtidos e diminuir os requisitos computacionais.

Domingos e Hulten desenvolveram uma metodologia baseada em amostragem para tornar algoritmos de aprendizado de máquina escaláveis para grandes bases de dados [13]. Ela usa o limite de Hoeffding [16] para escolher o número de exemplos que será utilizado pelo algoritmo, de forma a aumentar o seu desempenho. Garante-se que o modelo obtido considerando-se apenas a amostra de exemplos dada por este limite não difere muito do que seria obtido se a base de dados inteira fosse analisada. A metodologia proposta por Domingos e Hulten foi aplicada a alguns métodos proposicionais de aprendizado de máquina, como redes bayesianas, clusterização e árvores de decisão, tendo sido criados, respectivamente, os algoritmos VFBN [17], VFKM [12] e VFDT [11].

Sistemas atributo-valor, entretanto, não são muito apropriados para aprender a partir de dados relacionais, uma vez que a representação utilizada por estes métodos não consegue expressar, de uma maneira geral, as relações existentes entre os valores dos atributos. Sistemas ILP, ao contrário, utilizam lógica de primeira ordem para representar os exemplos, o conhecimento preliminar e a teoria gerada. Devido a isso, eles têm um maior poder de expressividade do que sistemas proposicionais, conforme foi dito anteriormente, e conseguem aprender a partir de dados relacionais. Seria interessante, portanto, utilizar programação em lógica indutiva para aprender teorias a partir de grandes bases de dados relacionais, uma

vez que a representação é mais apropriada e, por isso, acreditamos ser possível extrair mais conhecimento do que um sistema proposicional seria capaz.

Por outro lado, sistemas ILP são menos eficientes do que os proposicionais [3]. Isto ocorre porque provar um exemplo é uma tarefa custosa na programação em lógica indutiva, ou seja, testar se uma cláusula cobre um exemplo é mais complexo do que em métodos atributo-valor. Como forma de diminuir o custo da prova dos exemplos e, com isso, tornar o algoritmo de aprendizado mais rápido, o TILDE [3] [2], uma árvore de decisão em lógica de primeira ordem proposta por Blockeel e De Raedt, utiliza o aprendizado a partir de interpretações [37] e os pacotes de cláusulas (*query packs*) [1].

O presente trabalho propõe um sistema de programação em lógica indutiva escalável para grandes bases de dados. O sistema é denominado HTILDE (*Hoeffding TILDE*) e utiliza a metodologia proposta por Domingos e Hulten como forma de tornar o TILDE escalável para grandes bases de dados. O HTILDE tem como objetivo conciliar o poder de expressividade e a capacidade de lidar com dados relacionais de sistemas ILP com a eficiência obtida ao se utilizar o limite de Hoeffding para amostrar exemplos. O TILDE foi o sistema de programação em lógica indutiva escolhido por tratar de forma eficiente a questão do alto custo da prova de exemplos em ILP. É importante destacar que, devido ao fato do TILDE ser uma árvore de decisão de lógica de primeira ordem, o HTILDE é baseado também no VFDT, que é a árvore de decisão proposicional escalável para grandes bases de dados proposta por Domingos e Hulten.

A dissertação está organizada da seguinte maneira: no capítulo 2 são apresentados alguns conceitos básicos relacionados com a pesquisa, como uma introdução à programação em lógica indutiva e às árvores de decisão. Nos capítulos 3 e 4, são detalhados, respectivamente, o TILDE [3] e o VFDT [11], que são os dois sistemas que serviram de base para o nosso trabalho. No capítulo 5, introduzimos o HTILDE, que é o algoritmo proposto pela dissertação. No capítulo 6, apresen-

tamos os resultados experimentais obtidos. Finalmente, no capítulo 7, é feita a conclusão e sugerimos algumas direções para trabalhos futuros.

Capítulo 2

Conceitos básicos

Neste capítulo, apresentaremos alguns conceitos essenciais para o entendimento do restante da dissertação. Na seção 2.1, explicaremos o aprendizado de árvores de decisão. Já na seção 2.2, será feita uma introdução à programação em lógica indutiva. Na seção 2.3, explicaremos como avaliar um algoritmo de aprendizado de máquina. Na seção 2.4, mostraremos alguns conceitos de inferência estatística e como eles podem ser aplicados a aprendizado de máquina. Finalizamos este capítulo com a seção 2.5, em que explicamos o VFILPh [7], que é um sistema ILP que utiliza proposicionalização e o limite de Hoeffding [16] para lidar com bases de dados muito grandes. Os leitores já familiarizados com estes assuntos podem se encaminhar para o próximo capítulo.

2.1 Árvore de decisão

Nesta seção, faremos um breve introdução a um método de aprendizado de máquina fundamental para o nosso trabalho, chamado árvore de decisão. Para obter maiores informações, veja [26].

O aprendizado de árvores de decisão é um método utilizado para aproximar funções objetivo de valores discretos [26]. Em outras palavras, este algoritmo gera uma estrutura chamada árvore de decisão, que é utilizada para classificar exemplos. Uma árvore deste tipo pode ser vista na figura 2.1 e indica se um dia é adequado ou não para se jogar tênis.

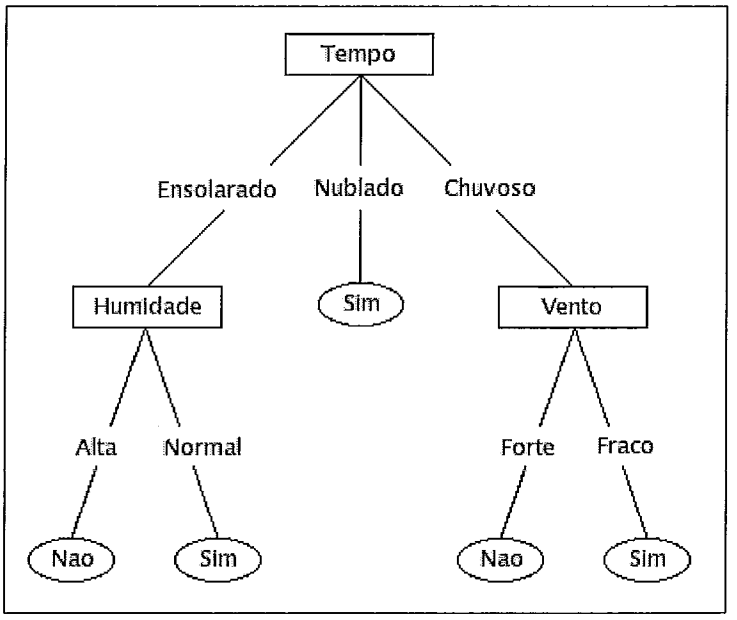


Figura 2.1: Exemplo de uma árvore de decisão para o conceito *JogarTênis*.

Nos problemas tratados por árvores de decisão, as funções objetivo têm valores de saída discretos, que são as possíveis classes dos exemplos. Problemas deste tipo, ou seja, que têm o objetivo de associar uma categoria a cada exemplo, dado um conjunto discreto de categorias, são chamados de *problemas de classificação.* Em problemas adequados a árvores de decisão, os exemplos são representados por pares atributo-valor e os valores também são discretos, mas existem extensões feitas ao algoritmo básico que possibilitam que eles possuam valores reais. Exemplos deste tipo podem ser vistos na tabela 2.1 e são usados como exemplos de treinamento para geração da árvore exibida na figura 2.1. Como esta árvore é gerada será explicado na sub-seção 2.1.1. Agora, é interessante observar que os dias são representados pelos atributos *Tempo*, *Temperatura*, *Humidade* e *Vento*, que possuem valores discretos, e as possíveis classes para o conceito *JogarTênis* são *Sim* e *Não* (ou seja, formam um conjunto discreto de categorias).

Todo nó interno de uma árvore de decisão especifica um atributo e cada ramo originado no nó corresponde a um dos possíveis valores do atributo. Toda folha de uma árvore deste tipo indica uma classe. Uma árvore de decisão classifica um

Dia	Tempo	Temperatura	Humidade	Vento	JogarTênis
D1	Ensolarado	Quente	Alta	Fraco	Não
D2	Ensolarado	Quente	Alta	Forte	Não
D3	Nublado	Quente	Alta	Fraco	Sim
D4	Chuvoso	Amena	Alta	Fraco	Sim
D5	Chuvoso	Fria	Normal	Fraco	Sim
D6	Chuvoso	Fria	Normal	Forte	Não
D7	Nublado	Fria	Normal	Forte	Sim
D8	Ensolarado	Amena	Alta	Fraco	Não
D9	Ensolarado	Fria	Normal	Fraco	Sim
D10	Chuvoso	Amena	Normal	Fraco	Sim
D11	Ensolarado	Amena	Normal	Forte	Sim
D12	Nublado	Amena	Alta	Forte	Sim
D13	Nublado	Quente	Normal	Fraco	Sim
D14	Chuvoso	Amena	Alta	Forte	Não

Tabela 2.1: Exemplos de treinamento para o conceito *JogarTênis*.

exemplo da seguinte forma: em cada nó interno, testa-se o atributo indicado no mesmo. Verifica-se qual é o valor deste atributo no exemplo e, então, ele segue para o próximo nó através do ramo correspondente a este valor. Este processo começa na raiz da árvore e termina quando o exemplo chega a uma folha, que define a classe do mesmo.

Para possibilitar o melhor entendimento de como uma árvore de decisão classifica exemplos, considere a da figura 2.1. Desejamos saber se o dia ($Tempo$ = *Ensolarado*, $Temperatura$ = *Quente*, $Humidade$ = *Alta*, $Vento$ = *Forte*) é adequado ou não para se jogar tênis. Este exemplo cai na folha mais a esquerda da árvore e, portanto, é classificado como *Não* (no caso, *JogarTênis* = *Não*). Logo, a árvore indica que este dia é inadequado para se jogar tênis.

Uma árvore de decisão pode ser convertida em um *conjunto de regras* equivalente. Cada folha corresponde a uma regra e esta é formada por uma conjunção dos testes que aparecem no caminho da raiz até a folha em questão [2]. O conjunto de regras exibidos a seguir corresponde à árvore da figura 2.1.

SE $Tempo = Ensolarado$ E $Humidade = Alta$ ENTÃO $Classe = Não$
SE $Tempo = Ensolarado$ E $Humidade = Normal$ ENTÃO $Classe = Sim$
SE $Tempo = Nublado$ ENTÃO $Classe = Sim$
SE $Tempo = Chuvoso$ E $Vento = Forte$ ENTÃO $Classe = Não$
SE $Tempo = Chuvoso$ E $Vento = Fraco$ ENTÃO $Classe = Sim$

É importante destacar que dizemos que uma regra *cobre* um exemplo específico se todas as condições de uma regra são verdadeiras para ele.

Quando um conjunto de regras é ordenado, ou seja, uma regra só é aplicável quando nenhuma das anteriores é, o chamamos de *lista de decisão* [2]. Para representar estas listas explicitando a ordem, podemos usar regras do formato SE-SENÃO-ENTÃO, em vez das do tipo SE-ENTÃO exibidas anteriormente. Note que as regras provenientes de uma árvore de decisão podem ser representadas por uma lista de decisão, uma vez que um exemplo cai em apenas uma folha da árvore, o que significa que ele é coberto por apenas uma das regras.

2.1.1 Indução de árvores de decisão

A maioria dos algoritmos que aprendem árvores de decisão realizam uma busca *top-down* e gulosa no espaço de todas as possíveis árvores de decisão [26]. Exemplos destes algoritmos são o ID3 [34] e o sucessor C4.5 [35]. Uma versão simplificada do ID3, que aprende árvores de decisão para problemas de duas classes, pode ser visto no algoritmo 2.1.

O ID3 aprende árvores de decisão tentando achar o melhor atributo para ser colocado em cada nó, começando pela raiz. Como um atributo é avaliado será explicado na sub-seção 2.1.2. Em cada nó, é criado um ramo para cada valor possível do atributo escolhido e, então, os exemplos de treinamentos deste nó são passados para os filhos apropriados, de acordo com o valor deste atributo em cada exemplo. O processo é repetido para todos os nós filhos, usando os exemplos de treinamento que caíram em cada um deles. O processo continua até que a árvore classifique perfeitamente os exemplos de treinamento ou até que todos os atributos tenham sido usados [26].

Algoritmo 2.1 Versão do algoritmo ID3 para aprender árvores de decisão para problemas de duas classes [26]

Entrada:

Exs é o conjunto de exemplos de treinamento,

AtribObjetivo é o atributo cujo valor a árvore deverá predizer.

Atribs é uma lista dos outros atributos, que poderão ser testes de nós internos da árvore.

Saída:

Uma árvore de decisão que classifica corretamente os exemplos de treinamento.

Procedimento ID3 (*Exs*, *AtribObjetivo*, *Atribs*)

1: Crie *Raiz*, que será o nó raiz da árvore.
2: Se todos os exemplos de *Exs* forem positivos, então
3: Faça *Raiz.classe* = "+".
4: Retorne *Raiz*, que é o nó raiz da árvore de decisão (formada por apenas um nó).
5: Se todos os exemplos de *Exs* forem negativos, então
6: Faça *Raiz.classe* = "-".
7: Retorne *Raiz*.
8: Se *Atribs* for uma lista vazia, então
9: Faça *Raiz.classe* = valor de *AtribObjetivo* mais freqüente entre os exemplos de *Exs* (ou seja, a classe majoritária).
10: Retorne *Raiz*.
11: Senão
12: Faça A ser o atributo de *Atribs* que melhor classifica os exemplos de *Exs* (ou seja, o atributo com o maior ganho de informação).
13: Faça *Raiz.atributo* = A.
14: Para cada possível valor v_i do atributo A
15: Adicione um novo ramo ao nó *Raiz*, correspondente ao teste $A = v_i$.
16: Faça Exs_{v_i} ser o subconjunto de *Exs* formado pelos exemplos que têm o atributo A com valor v_i.
17: Se o conjunto Exs_{v_i} for vazio, então
18: Crie uma nova folha F.
19: Faça *F.classe* = valor de *AtribObjetivo* mais freqüente entre os exemplos de *Exs* (ou seja, a classe majoritária).
20: Adicione F ao ramo.
21: Senão
22: Crie uma subárvore *SubArv*.
23: Faça *SubArv* = ID3(Exs_{v_i}, *AtribObjetivo*, *Atribs* - $\{A\}$).
24: Adicione *SubArv* ao ramo.
25: Retorne *Raiz*, que é o nó raiz da árvore de decisão.

2.1.2 Ganho de informação

Para entender como o ID3 seleciona o atributo que será colocado em um nó da árvore, é necessário definir dois conceitos: *entropia* e *ganho de informação.*

Entropia é uma medida muito usada em teoria da informação e caracteriza a impureza de uma coleção arbitrária de exemplos. Para obter maiores informações sobre o seu significado, veja [26]. Dada uma coleção S que contenha exemplos positivos e negativos de algum conceito a ser aprendido, a entropia de S relativa a esta classificação booleana (ou seja, no caso em que só existem duas classes) é dada pela equação 2.1, onde $p_{\oplus}$ é a proporção de exemplos positivos em S e $p_{\ominus}$ é a proporção de exemplos negativos em S. Nos cálculos de entropia, definimos $0 \log 0 = 0$.

$$Entropia(S) = -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus} \tag{2.1}$$

A figura 2.2 mostra o gráfico da entropia para o caso em que existem duas classes. É possível notar que se todos os membros de S forem da mesma classe, a entropia será 0. Se S contiver a mesma quantidade de exemplos positivos e negativos, a entropia será 1. Finalmente, se não ocorrer nenhum destes dois casos, a entropia será um número entre 0 e 1.

De forma a possibilitar o melhor entendimento de como a entropia é calculada, utilizaremos um exemplo. Suponha, novamente, que existam apenas duas classes. Seja S uma coleção com 14 exemplos, sendo 9 positivos e 5 negativos (usaremos a notação S = [9+, 5-]). O cálculo da entropia de S relativa a esta classificação é exibido a seguir:

$$Entropia([9+, 5-]) = - (9/14) \log_2 (9/14) - (5/14) \log_2 (5/14) = 0{,}940$$

Para finalizarmos o estudo sobre entropia que interessa ao nosso trabalho, é importante destacar o caso geral do cálculo da mesma, em que existem mais de duas classes. A entropia da coleção de exemplos S no caso em que existem c classes

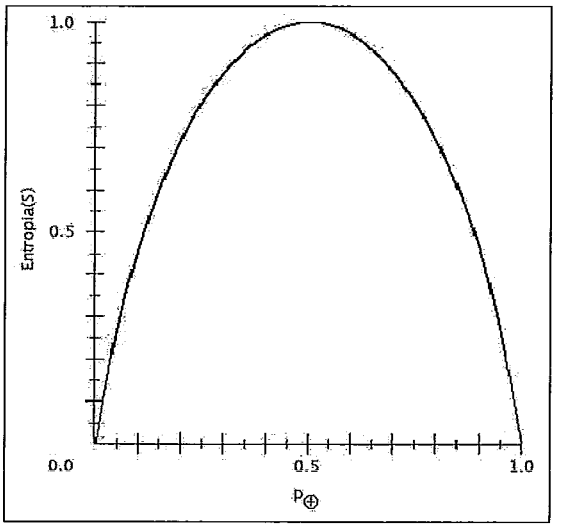


Figura 2.2: Gráfico da entropia relativa a uma classificação booleana. $p_{\oplus}$ é a proporção de exemplos positivos e varia de 0 a 1.

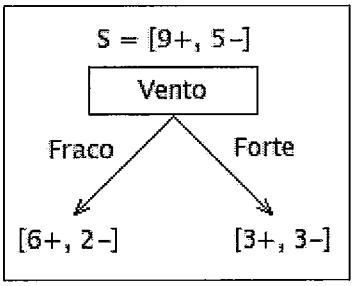
é dada pela equação 2.2, onde p_i é a proporção de exemplos em S pertencentes à classe i. Neste caso, o valor máximo da entropia não é mais 1, como no caso booleano, mas $\log_2 c$.

$$Entropia(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2.2}$$

Ganho de informação é uma medida de quão eficaz é um atributo em classificar os exemplos de treinamento. O ganho de informação de um atributo A relativo à coleção de exemplos S é dado pela equação 2.3, onde $Valores(A)$ é o conjunto de todos os possíveis valores de A e S_v é o subconjunto de S em que o atributo A tem valor v (ou seja, $S_v = \{s \in S | A(s) = v\}$). Observe que o primeiro termo da equação 2.3 é a entropia da coleção original S e o segundo termo é o valor esperado da entropia após S ser particionado de acordo com A. Este valor esperado é a soma das entropias dos subconjunto S_v, sendo o peso de cada parcela a proporção de exemplos do subconjunto. Portanto, $Ganho(S, A)$ é a redução esperada da entropia causada pela divisão dos exemplos de acordo com o atributo A [26].

$$Ganho(S, A) = Entropia(S) - \sum_{v \in Valores(A)} \frac{|S_v|}{|S|} Entropia(S_v) \qquad (2.3)$$

Utilizaremos, novamente, um exemplo para facilitar o entendimento. Suponha que S seja uma coleção de exemplos de treinamento e um dos atributos seja $Vento$, que pode ter os valores $Fraco$ ou $Forte$. Como anteriormente, S contém 9 exemplos positivos e 5 negativos. Destes 14 exemplos, 6 positivos e 2 negativos têm $Vento = Fraco$; os restantes têm $Vento = Forte$. A figura 2.3 mostra o cálculo do ganho de informação do atributo $Vento$ para este caso.



$Valores(Vento) = \{Fraco, Forte\}$
$S = [9+, 5-]$
$S_{Fraco} \leftarrow [6+, 2-]$
$S_{Forte} \leftarrow [3+, 3-]$

$Ganho(S, Vento) = Entropia(S) - \sum_{v \in \{Fraco, Forte\}} \frac{|S_v|}{|S|} Entropia(S_v)$
$= Entropia(S) - (8/14) Entropia(S_{Fraco}) - (6/14) Entropia(S_{Forte})$

$Entropia(S) = 0,940$ (calculada anteriormente)
$Entropia(S_{Fraco}) = -(6/8)\log_2(6/8) - (2/8)\log_2(2/8) = 0,811$
$Entropia(S_{Forte}) = -(3/6)\log_2(3/6) - (3/6)\log_2(3/6) = 1,00$

Logo,
$Ganho(S, Vento) = 0,940 - (8/14)0,811 - (6/14)1,00 = 0,048$

Figura 2.3: Exemplo de cálculo de ganho de informação.

O ganho de informação é a medida que o ID3 usa para selecionar o melhor atributo para ser colocado em cada nó. O atributo escolhido é o que tem o maior valor de ganho de informação.

Para finalizar, vale ressaltar que existem outros critérios para selecionar atributos durante a geração de árvores de decisão. Uma outra medida muito conhecida é a *taxa de ganho de informação*, que é exibida pela equação 2.5, onde $InfoDivisao$ é a *informação de divisão*, definida na equação 2.4.

$$InfoDivisao(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2 \frac{|S_i|}{|S|} \tag{2.4}$$

$$TaxaGanho(S, A) = \frac{Ganho(S, A)}{InfoDivisao(S, A)} \tag{2.5}$$

2.2 Programação em lógica indutiva

Nesta seção, faremos uma introdução à programação em lógica indutiva (*ILP - Inductive Logic Programming*). Para maiores informações sobre o assunto, veja [28], [22], [29]. Já para obter maiores informações sobre a terminologia de lógica de primeira ordem utilizada, veja [8].

A programação em lógica indutiva foi definida como a área de pesquisa formada pela interseção de aprendizado indutivo e programação em lógica [29]. Ela tem como objetivo aprender programas lógicos a partir de exemplos e de um conhecimento preliminar. A tarefa mais básica em *ILP* é aprender uma relação (predicado alvo) a partir de exemplos, em termos das relações definidas no conhecimento preliminar e possivelmente do próprio predicado alvo, no caso de relações recursivas. Um esquema do aprendizado em *ILP* pode ser visto na figura 2.4. No esquema, $E = E^+ \cup E^-$ é o conjunto de exemplos, onde E^+ e E^- são, respectivamente, os exemplos positivos e negativos (ou seja, tuplas que pertencem e não pertencem à relação alvo p). Além disso, B é o conhecimento preliminar, formado por relações (ou predicados) preliminares e H é a teoria em lógica de primeira ordem aprendida pelo sistema, que desejamos que explique todos os exemplos positivos e nenhum negativo.

A definição 2.1, extraída de [3], apresenta a tarefa tratada pela programação em lógica indutiva de uma maneira mais formal. Existem outras formulações para

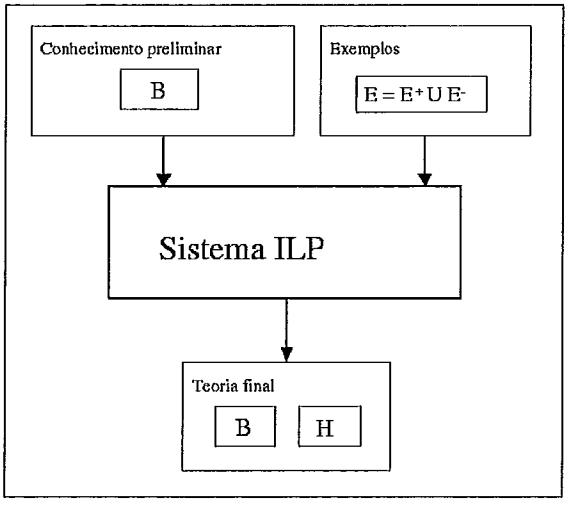


Figura 2.4: Esquema do aprendizado em programação em lógica indutiva (*ILP*), onde B é o conhecimento preliminar, E é o conjunto de exemplos, formado pelos exemplos positivos (E^+) e negativos (E^-), e H é a teoria aprendida pelo sistema.

o problema, que podem ser vistas em [37], mas o *aprendizado a partir de implicação lógica* [28] é a forma tradicional de aprendizado em *ILP*.

Definição 2.1 (Aprendizado a partir de implicação lógica). *Sejam dados:*

- *Um conjunto E^+ de exemplos positivos e um conjunto E^- de exemplos negativos.*
- *Uma teoria preliminar B.*

Deseja-se achar:

- *Uma hipótese H, tal que*
  - $\forall e \in E^+ : H \wedge B \models e, \qquad e$
  - $\forall e \in E^- : H \wedge B \not\models e$

Para ilustrar melhor o tipo de problema tratado em *ILP*, é interessante usar um exemplo. Um problema clássico de programação em lógica indutiva é o dos trens, que tem como objetivo separá-los em duas classes: os que vão para o leste e os que se dirigem ao oeste. A figura 2.5 exibe os 10 exemplos considerados no

problema, sendo 5 de cada classe (ou seja, 5 trens que vão para cada uma das direções). Cada trem possui pelo menos um vagão e cada vagão pode ter várias propriedades: pode ser longo ou curto, pode ser aberto ou fechado, pode carregar objetos no seu interior, etc. De acordo com a descrição do trem e dos seus vagões, desejamos encontrar uma teoria em lógica de primeira ordem que defina quando um trem vai para o leste e quando ele vai para o oeste.

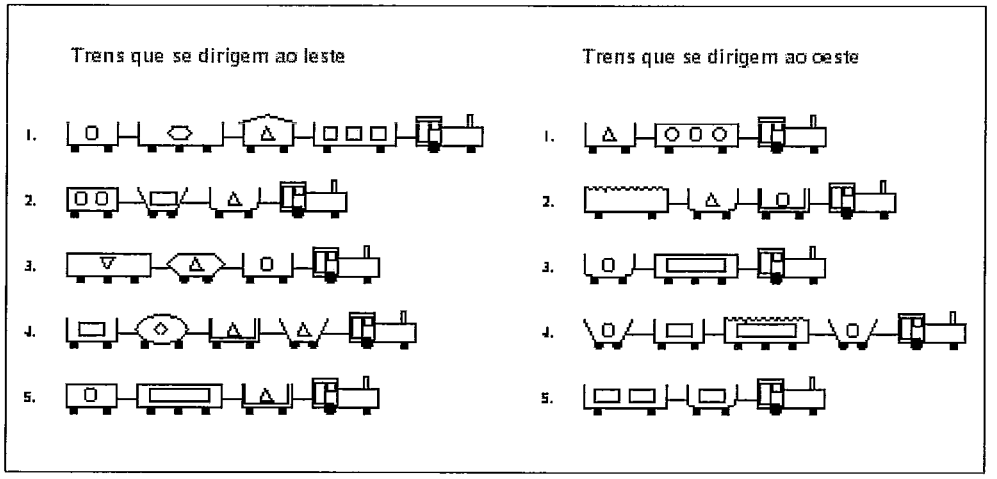


Figura 2.5: Exemplos considerados no problema dos trens.

Este problema pode ser definido em *ILP* da seguinte maneira: desejamos aprender o predicado *direcao_leste*(T), que deve ser ser $Verdadeiro$ se o trem T for para o leste e $Falso$ se T for para o oeste. Mostramos, a seguir, os exemplos usados para o aprendizado para este predicado. Os trens com nome **leste***i*, $1 \leq i \leq 5$, são os exemplos positivos, ou seja, os trens do conjunto de exemplos que vão para o leste. Já os trens com nome **oeste***i*, $1 \leq i \leq 5$, são os exemplos negativos, ou seja, os trens que não se dirigem ao para o leste.

Exemplos positivos:
direcao_leste(leste1).
direcao_leste(leste2).
direcao_leste(leste3).
direcao_leste(leste4).
direcao_leste(leste5).

Exemplos negativos:
direcao_leste(oeste1).
direcao_leste(oeste2).
direcao_leste(oeste3).
direcao_leste(oeste4).
direcao_leste(oeste5).

A seguir, apresentamos parcialmente o conhecimento preliminar, através do qual informamos quais vagões cada trem possui e quais as características destes vagões. No trecho exibido, informamos que primeiro trem do conjunto de exemplos que vai para o leste tem 4 vagões. O primeiro vagão é longo, aberto e carrega 3 objetos retangulares; já o segundo vagão é curto, fechado e carrega um objeto triangular.

```
tem_vagao(leste1,vagao_11).          tem_vagao(leste1,vagao_12).
tem_vagao(leste1,vagao_13).          tem_vagao(leste1,vagao_14).
longo(vagao_11).                     curto(vagao_12).
aberto(vagao_11).                    fechado(vagao_12).
carrega(vagao_11,retangulo,3).       carrega(vagao_12,triangulo,1).
...
```

A partir dos exemplos e do conhecimento preliminar, esperamos que o sistema *ILP* seja capaz de gerar a seguinte teoria lógica:

```
direcao_leste(A) :- tem_vagao(A,B), curto(B), fechado(B).
```

Esta é uma teoria simples, formada por uma única cláusula, que diz que um trem se dirige ao leste se ele tiver um vagão curto e fechado. Observe a figura 2.5 e note que, de fato, todos os trens que vão para o leste obedecem a esta regra e nenhum dos que vão para o oeste possuem esta característica.

Existem alguns sistemas de programação em lógica indutiva, como o PROGOL [27], o Aleph [38], o FOIL [33] e o TILDE [3] [2], que é um dos sistemas em que o nosso trabalho é baseado e será explicado no capítulo 3. É importante destacar que o problema dos trens foi modelado anteriormente de acordo com a representação utilizada pelo Aleph. A representação do TILDE seria diferente, conforme explicaremos no capítulo 3.

Para concluir a seção, apresentamos o pseudo-código do FOIL [33], que é um dos primeiros sistemas *ILP*. O algoritmo 2.2, que foi extraído de [36], exibe o pseudo-código do FOIL, que contém dois *loops*, um externo e um interno. No

externo, a cada iteração, é gerada uma cláusula que cobre parte dos exemplos positivos. Este *loop* termina quando todos os exemplos positivos são cobertos. Já o *loop* interno constrói uma única cláusula, onde a cada iteração é adicionado um literal para que exemplos negativos deixem de ser cobertos. Este *loop* termina quando a cláusula em questão não cobre mais nenhum exemplo negativo e, então, ela é adicionada à teoria.

Não entraremos em detalhes a respeito do espaço de busca dos literais, ou seja, quais literais são considerados pelo FOIL para serem adicionados a uma cláusula. Achamos interessante destacar, apenas, que a heurística usada para se escolher um literal a cada iteração do *loop* interno é o *ganho de informação*: o literal que apresentar o maior ganho é adicionado à cláusula que está sendo construída.

Algoritmo 2.2 Algoritmo do FOIL, proposto em [36]

1: Seja *teoria* um programa Prolog vazio (ou seja, sem nenhuma cláusula).
2: Seja *posRestantes* o conjunto de todas os exemplos positivos da relação a ser aprendida R.
3: Enquanto *posRestantes* não for vazio
4: Faça *clausula* = **R(A, B, ...) :-** (em outras palavras, comece uma nova cláusula).
5: Enquanto *clausula* cobrir exemplos negativos de R
6: Encontre o literal L apropriado (ou seja, que exclua alguns exemplos negativos).
7: Adicione L ao lado direito da *clausula*.
8: Remova os exemplos positivos cobertos pela *clausula* de *posRestantes*.
9: Adicione *clausula* à *teoria*.
10: Retorne *teoria*.

2.3 Avaliação do aprendizado

Nesta seção, apresentaremos formas de se avaliar um algoritmo de aprendizado de máquina. De maneira simplificada, queremos saber se a estrutura gerada pelo algoritmo (por exemplo, uma árvore de decisão ou uma teoria em lógica de primeira ordem) conseguiu classificar bem os exemplos.

Começaremos explicando algumas medidas usadas para este fim. Como nos problemas utilizados nesta dissertação os exemplos são classificados apenas em *positivos* e *negativos*, apresentaremos as fórmulas destas medidas considerando somente estas duas classes. Um *verdadeiro positivo* é um exemplo que foi classificado pelo algoritmo como positivo e, de fato, possui esta classe. Um *falso positivo* é um exemplo que foi classificado como positivo pelo algoritmo, mas de forma errada porque, na realidade, ele é negativo. Analogamente, um *verdadeiro negativo* é um exemplo que foi classificado corretamente como negativo pelo algoritmo e, finalmente, um *falso negativo* é um exemplo que foi classificado erroneamente como negativo pelo algoritmo porque, na realidade, ele é positivo. Sejam V_p, F_p, V_n e F_n, respectivamente, o número de verdadeiros positivos, de falsos positivos, de verdadeiros negativos e de falsos negativos obtidos ao se classificar um conjunto de exemplos utilizando-se um modelo gerado por um algoritmo de aprendizado de máquina. Tendo sido estabelecida esta notação, podemos definir as medidas que serão usadas neste trabalho.

A *acurácia* é a proporção dos exemplos que foram classificados corretamente pelo algoritmo. A equação 2.6 apresenta a fórmula desta medida.

$$Acurácia = \frac{V_p + V_n}{V_p + F_p + V_n + F_n} \tag{2.6}$$

A acurácia é a medida mais comumente usada para se avaliar o desempenho de um algoritmo. Entretanto, ela não é a mais adequada quando a base de dados é desbalanceada, ou seja, uma das classes possui muito mais exemplos do que a outra. Vamos utilizar um exemplo para facilitar o entendimento do problema em questão: suponha que a base de dados considerada tenha 90% dos exemplos de uma classe e apenas 10% da outra. Se o algoritmo classificasse todos os exemplos como sendo da majoritária, a acurácia seria de 90%. Apesar da acurácia ser alta, todos os exemplos da outra classe seriam classificados de forma errada, o que é um sério problema.

Existem duas outras medidas mais apropriadas para se avaliar um algoritmo quando a base de dados é desbalanceada: a *precisão* (*precision*) e a *revocação* (*recall*), cujas fórmulas são exibidas, respectivamente, nas equações 2.7 e 2.8. A precisão é a proporção de exemplos corretamente classificados como positivos entre todos os que foram classificados como positivos. Já a revocação é a proporção de exemplos corretamente classificados como positivos entre todos os que realmente são positivos.

$$Precisão = \frac{V_p}{V_p + F_p} \tag{2.7}$$

$$Revocação = \frac{V_p}{V_p + F_n} \tag{2.8}$$

A *medida F* é uma medida que combina a precisão e a revocação. A fórmula mais simples, do caso em que se dá igual importância à precisão e à revocação, é mostrada na equação 2.9. Neste caso, a medida F corresponde à média harmônica destas duas outras medidas. Ela tem como valor de máximo 1, no caso em que $precisão = revocação = 1$, e tem valor de mínimo 0, quando $precisão = 0$ ou $revocação = 0$. Para maiores informações a respeito da medida F, veja [39].

$$Medida\ F = \frac{2 \times precisão \times revocação}{precisão + revocação} \tag{2.9}$$

Para se avaliar um algoritmo, geralmente, são utilizados exemplos que não foram usados para treiná-lo. Por isso, normalmente, dividimos o conjunto total de exemplos em dois subconjuntos disjuntos: o *conjunto de treinamento* e o *conjunto de teste*. O primeiro é usado pelo algoritmo para se aprender o modelo; já o segundo é usado para ele ser avaliado, usando-se uma das medidas explicadas anteriormente. Utilizamos estes dois conjuntos porque o modelo está mais adaptado para avaliar corretamente os exemplos que o treinaram, mas desejamos saber se o modelo aprendido consegue classificar corretamente exemplos que não foram previamente vistos. Entretanto, em alguns casos pode ser interessante avaliar também o modelo usando o conjunto de treinamento. Um exemplo ocorre quando a medida

usada para avaliá-lo é muito baixa, o que pode significar que são necessários mais exemplos para se treinar o algoritmo.

Finalizamos esta seção explicando a *validação cruzada* [19], que é um método normalmente usado para se avaliar algoritmos de aprendizado de máquina. A partir da base de dados original, são gerados novos conjuntos de exemplos, chamados *folds*. Cada *fold* é formado por duas partições, que são os conjuntos de treinamento e de teste do mesmo. Cada algoritmo é executado várias vezes, uma por *fold*, e a medida de avaliação do algoritmo (por exemplo, a acurácia do mesmo) é a média dos valores obtidos em cada *fold*.

Existem algumas formas de validação cruzada, como a *k-fold* [19] e a 5x2 [10]. Na *validação cruzada k-fold*, que é uma das formas mais tradicionais, o conjunto original de exemplos é dividido em k partições. São criados k *folds* e, em cada um deles, uma dessas partições forma o conjunto de teste e as outras $k-1$ partições constituem o conjunto de treinamento. Já na *validação cruzada 5x2*, os exemplos da base de dados são agrupados em cinco conjuntos, que chamaremos de C_i ($1 \leq i \leq 5$). Cada um deles é dividido em dois subgrupos, que chamaremos de S_{i1} e S_{i2}. Cada conjunto C_i cria dois *folds*: no primeiro, S_{i1} é usado para treinar o algoritmo e S_{i2} é utilizado para teste; no segundo, ao contrário, S_{i1} é o conjunto de teste e S_{i2} é o de treinamento.

2.4 Inferência estatística

A inferência estatística é um ramo da matemática que investiga como tirar conclusões a respeito de uma população utilizando-se apenas informações de uma amostra. Este assunto é de particular interesse para o nosso trabalho porque uma das principais formas de lidar com bases de dados muito grandes é considerar apenas um subconjunto dos exemplos [32]. Existem duas questões muito importantes que devem ser consideradas: como a amostra será gerada e qual será o seu tamanho.

As duas principais formas de se gerar um subgrupo de exemplos são a amostragem casual simples e a amostragem estratificada. Na primeira, também chamada de aleatória simples, todos os elementos da população têm igual probabilidade de serem escolhidos. Já a amostragem estratificada pode ser usada quando a população é heterogênea. Nesse caso, a população é dividida em subgrupos homogêneos e, então, seleciona-se uma amostra aleatória de cada população. A amostragem estratificada é dita proporcional quando as subpopulações são representadas na amostra na mesma proporção do que na população original [9].

No caso de problemas de classificação em que a base de dados é desbalanceada, ou seja, uma classe tem um número muito maior de exemplos do que as outras, a amostragem estratificada deve ser usada, como forma de garantir que as classes minoritárias sejam representadas na amostra. Em algumas abordagens, é usada a amostragem proporcional; em outras, elementos da classe minoritária são selecionados com uma freqüência maior da que os da classe majoritária, visando-se balancear a amostra [32].

Existem várias técnicas que podem ser usadas para determinar o tamanho da amostra dos dados. Estratégias mais simples consideram apenas um número fixo ou uma fração dos exemplos para treinar o algoritmo de aprendizado de máquina [24]. Outras técnicas determinam o número de exemplos de treinamento utilizando limites estatísticos para o erro que pode ser gerado pela amostra. O sistema VFDT [11] adota este tipo de abordagem e utiliza o limite de Hoeffding [16], que será explicado no capítulo 4. Outras estratégias criam uma série de modelos, a partir de uma seqüência de amostras, até chegarem a um ponto em que a acurácia não consegue mais ser melhorada. Um exemplo desta abordagem é a amostragem progressiva [31] e será explicada a seguir.

2.4.1 Amostragem progressiva

Antes de apresentarmos a amostragem progressiva, é importante explicarmos o conceito de *curva de aprendizado*. Uma curva de aprendizado é um gráfico que

exibe o desempenho do modelo aprendido de acordo com o tamanho da amostra de exemplos utilizado. O eixo horizontal representa a quantidade de exemplos usados para treinar o algoritmo. Já o eixo vertical mostra alguma medida usada para avaliá-lo, normalmente a acurácia.

Geralmente, curvas de aprendizado possuem três regiões características. No início, ela possui uma subida íngreme; no meio, a medida de desempenho (por exemplo, a acurácia) cresce de maneira mais moderada; já no final, ela estabiliza e é praticamente constante. Quando o algoritmo chega à região final da curva, conhecida como *plateau*, dizemos que ele convergiu. O tamanho de cada uma dessas regiões depende do problema. A figura 2.6 mostra dois exemplos de curvas de aprendizado, uma com convergência lenta e outra com convergência rápida.

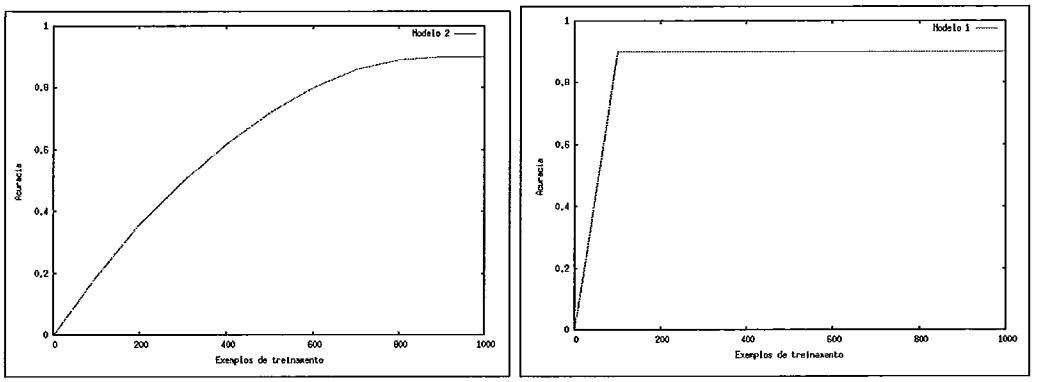


Figura 2.6: Exemplos de curvas de aprendizado, uma com convergência lenta e a outra com convergência rápida.

A amostragem progressiva tem como objetivo encontrar n_{min}, que é o tamanho da menor conjunto de treinamento suficiente para o aprendizado do modelo. Em outras palavras, modelos construídos com menos exemplos têm desempenho pior (por exemplo, acurácia menor) e modelos construídos com uma amostra maior não são melhores do que o construído com n_{min} exemplos [31]. Observando a curva de aprendizado, n_{min} é a quantidade necessária de exemplos para que se chegue ao *plateau*.

O algoritmo 2.3 exibe o pseudo-código da amostragem progressiva. Basicamente, são gerados modelos a partir de amostras de diferentes tamanhos, até que

se chegue à convergência, ou seja, o modelo não consiga obter melhor desempenho ao se aumentar o tamanho da amostra. Existem duas questões relevantes para este tipo de amostragem: como determinar a seqüência S de tamanhos de amostras e como detectar que se chegou à convergência.

Algoritmo 2.3 Algoritmo da amostragem progressiva [31]

Entrada:

A é o algoritmo de aprendizado de máquina,
T é o conjunto de exemplos de treinamento.

Saída:

Modelo M produzido pelo algoritmo A usando n_{min} exemplos de T.

Procedimento AmostragemProgressiva (A, T)

1: Crie uma seqüência de tamanhos de amostras $S = \{n_0, n_1, ..., n_k\}$.
2: Faça $n = n_0$.
3: Faça M ser o modelo produzido por A usando n exemplos de T.
4: Enquanto não convergir
5: Crie uma nova seqüência S, se for necessário.
6: Faça n ser o próximo elemento de S, maior do que o valor atual de n.
7: Faça M ser o modelo produzido por A usando n exemplos de T.
8: Retorne M.

Uma abordagem usada para se gerar a seqüência de amostras é utilizar uma progressão geométrica. A seqüência S adquire a forma $S_g = a^i n_0 = \{n_0, an_0, a^2 n_0, ..., N\}$, para as constantes n_0 e a. Um exemplo de seqüência deste tipo é $S = \{100, 200, 400, 800, ..., N\}$. Em [31], é provado que a amostragem geométrica é assintoticamente ótima.

Não exite uma abordagem definitiva para a detecção de convergência. Em [31], é utilizada a regressão linear com amostragem local (*LRLS - linear regression with local sampling*). Neste método, para cada amostra de tamanho n_i, outras amostras com tamanho próximo são usadas para se realizar uma regressão linear. Quando a inclinação for suficientemente próxima de zero, diz-se que se chegou à convergência.

2.5 VFILPh

O sistema VFILPh [7] [6] é um sistema ILP capaz de aprender modelos a partir de bases de dados muito grandes. VFILP significa *Very Fast Inductive Logic Programming*; já o *h* ressalta o fato do sistema utiliza o limite de Hoeffding [16] como forma de amostrar exemplos.

O VFILPh utiliza o VFDT [11], que é um dos sistemas proposicionais que usam a metodologia proposta por Domingos e Hulten em [13] como forma de lidar com grandes bases de dados. O VFDT será explicado em detalhes no capítulo 4, por ser também um dos algoritmos que serviram de base para o sistema proposto nesta dissertação. Nesta seção, é interessante ressaltar que o VFDT é uma árvore de decisão proposicional que utiliza o limite de Hoeffding como forma de amostrar exemplos e aprender modelos a partir de bases de dados muito grandes.

Como o VFILPh é um sistema ILP e o VFDT é proposicional, é feita uma transformação do problema relacional para uma representação atributo-valor. Este processo é chamado de proposicionalização [21] [7].

O VFILPh possui quatro módulos, sendo os dois primeiros responsáveis por realizar o processo de proposicionalização. O primeiro módulo cria os atributos e é baseado no sistema RSD *Relational Sub-group Discovery* [23]. Cada atributo é um conjunto de literais de lógica de primeira ordem que compartilham variáveis. Um exemplo de atributo que poderia ser gerado para o problema dos trens, que foi descrito na seção 2.2, seria $atributo01(T)$:- $tem\_vagao(T,C)$, $longo(C)$. No segundo módulo, a partir dos atributos e do conhecimento preliminar, é gerada uma tabela proposicional. Esta tabela indica, para cada exemplo, o valor de cada atributo ($Verdadeiro$ ou $Falso$) [7].

O terceiro módulo do VFILPh realiza o aprendizado. A tabela proposicional criada previamente é dada como entrada para o sistema VFDT, que então gera uma árvore de decisão. Um exemplo de árvore que poderia ser gerada por este módulo pode ser visto na figura 2.7. Finalmente, o quarto módulo constrói as regras de primeira ordem a partir da árvore de decisão e dos atributos [7].

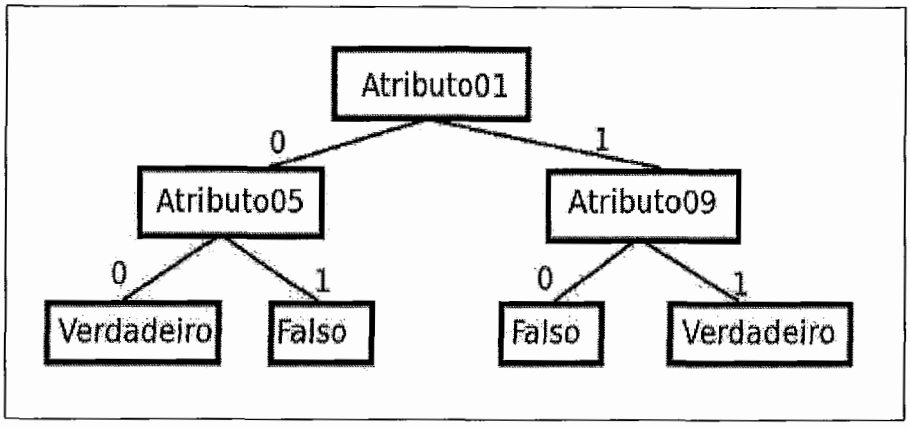


Figura 2.7: Exemplo de uma árvore de decisão gerada pelo terceiro módulo do sistema VFILPh (figura extraída de [7]).

Capítulo 3

TILDE - Top-down Induction of Logical Decision Trees

Neste capítulo, será apresentado o TILDE (*Top-down Induction of Logical Decision Trees*) [3] [2], que é um dos dois sistemas que serviram de base para o nosso trabalho. Na seção 3.1, será feita uma introdução ao método. Na seção 3.2, explicaremos como um problema é representado no sistema e apresentaremos a definição de *aprendizado por interpretações*, que é um conceito fundamental para o TILDE. Na seção 3.3, mostraremos o algoritmo de indução de árvores de decisão de lógica de primeira ordem do sistema, além do algoritmo que classifica um exemplo utilizando este tipo de árvore. Já na seção 3.4, entraremos em maiores detalhes a respeito de como é definido o operador de refinamento do sistema. Finalizamos o capítulo com a seção 3.5, em que explicamos o conceito de *pacotes de cláusulas*, que é uma ferramenta importante do TILDE para acelerar a prova de exemplos em ILP.

3.1 Introdução

O TILDE é um sistema que classifica exemplos através da geração de uma árvore de decisão de lógica de primeira ordem. Isto significa que os testes dos nós internos não são atributos, como em árvores de decisão tradicionais, mas literais ou conjunções de literais. Devido a isso, os ramos da árvore podem ter apenas os valores $Verdadeiro$ ou $Falso$, o que a faz ser binária. O ramo esquerdo do

nó corresponde ao valor *Verdadeiro* e o direito, ao *Falso*. Um exemplo de uma árvore deste tipo pode ser visto na figura 3.1. Entraremos em maiores detalhes a respeito de como estas árvores são construídas na seção 3.3.

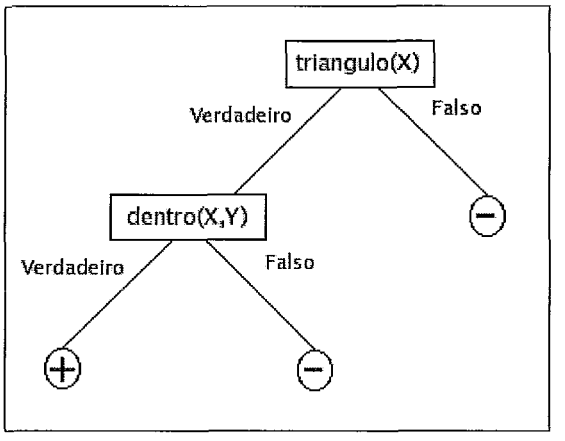


Figura 3.1: Exemplo de uma árvore gerada pelo TILDE.

A figura 3.1 exibe a árvore gerada pelo TILDE para o problema Bongard [4]. Neste problema, os exemplos são figuras e deseja-se classificá-las como *positivas* ($\oplus$) ou *negativas* ($\ominus$) de acordo com os objetos que as constituem. Cada figura é formada por um número variável de objetos e existem relações entre eles (por exemplo, um objeto pode estar dentro de outro). Alguns exemplos e as suas classes são exibidas na figura 3.2. Entraremos em mais detalhes a respeito da representação do problema na seção 3.2, mas por enquanto é importante que a árvore representada na figura 3.1 seja compreendida. O literal $triangulo(X)$ testa se há um triângulo na figura e o literal $dentro(X, Y)$ testa se o objeto X está dentro do objeto Y. Portanto, descendo sempre pelos ramos esquerdos da árvore, que são os de valor *Verdadeiro*, testa-se se há um triângulo na figura e se ele está dentro de algum outro objeto. Como por este caminho chegamos a uma folha $\oplus$, podemos concluir que os exemplos que obedecem a esse teste são positivos.

De forma a possibilitar o entendimento de algumas características do TILDE que serão apresentadas neste capítulo, é necessário salientar algumas diferenças importantes entre sistemas ILP e proposicionais. Sistemas ILP são capazes de re-

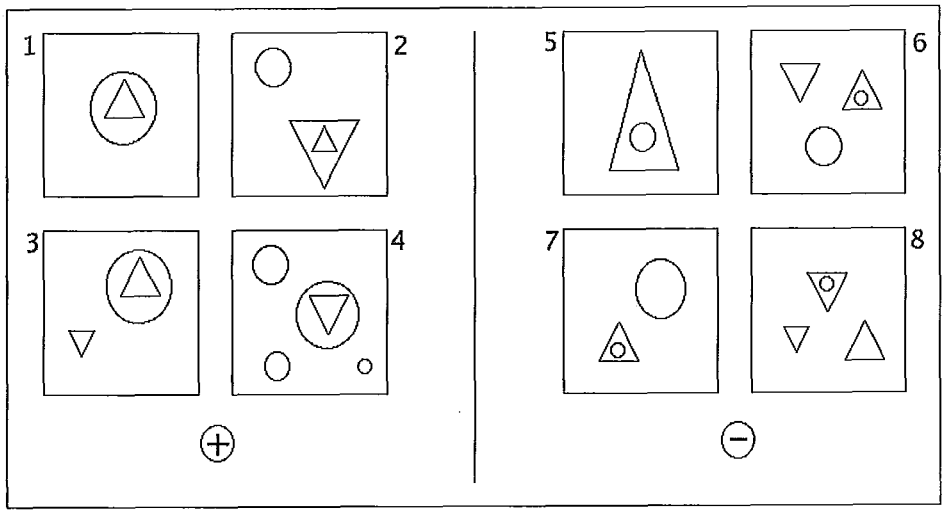


Figura 3.2: Exemplos do problema Bongard.

solver problemas que métodos atributo-valor não conseguem, por terem um poder de expressividade maior [5]. Entretanto, sistemas ILP são menos eficientes e uma das principais razões disto é que testar se uma cláusula cobre um exemplo é mais complexo em ILP do que em métodos atributo-valor. Isto ocorre porque, em sistemas proposicionais, é usado o *teste de cobertura local*, já que o fato de uma cláusula cobrir ou não um exemplo não depende de outros exemplos. Entretanto, na grande maioria dos sistemas ILP, é usado o *teste de cobertura global*, já que para se testar se uma cláusula cobre um exemplo pode ser necessário considerar todos os outros exemplos e o conhecimento preliminar.

Para facilitar a compreensão do conceito de teste de cobertura global, utilizaremos como exemplo o predicado *pertence*(E, L), que indica se um elemento E pertence à lista L. Suponha que sejam fornecidos os exemplos e as suas respectivas classes ($\oplus$ e $\ominus$), conforme ilustrado a seguir, onde [] é uma lista vazia.

$\oplus$: pertence(a, [a,b,c]).
$\oplus$: pertence(d, [e,d,c,b]).
$\oplus$: pertence(d, [d,c,b]).
$\ominus$: pertence(b, [a,c,d]).
$\ominus$: pertence(a, []).
$\ominus$: pertence(d, [c,b]).

As classes estão relacionadas com o valor do literal que o exemplo representa: os exemplos de classe $\oplus$ são os literais que têm valor verdade $Verdadeiro$, enquanto os de classe $\ominus$ são os literais que têm valor verdade $Falso$. Esperamos que um sistema ILP obtenha a definição do predicado *pertence* exibida a seguir, onde se $L = [X|Z]$ uma lista, X é o primeiro elemento e Z é a sublista resultante da remoção do primeiro elemento de L.

pertence(X, $[X|Z]$).
pertence(X, $[Y|Z]$) :- pertence(X,Z).

Esta definição diz que um elemento pertence a uma lista L se ele for o primeiro elemento da mesma ou se ele pertencer à sublista resultante da retirada do primeiro elemento de L.

Nota-se que a classe de um exemplo depende da classe de outros exemplos. Isto pode ser constatado observando que a classe de $pertence(d, [e, d, c, b])$ depende da classe de $pertence(d, [d, c, b])$, que é um outro exemplo. Em outras palavras, para se testar se a cláusula *pertence* cobre o exemplo $(d, [e, d, c, b])$, precisamos saber se ela cobre o exemplo $(d, [d, c, b])$.

Testes de cobertura globais são mais custosos do que os locais. Na seção 3.2, mostraremos como o TILDE utiliza o teste de cobertura local como forma de melhorar a eficiência do sistema.

Uma outra ferramenta usada pelo TILDE para diminuir o custo da prova dos exemplos e, com isso, tornar o algoritmo mais eficiente, é a utilização dos pacotes de cláusulas (*query packs*) [1]. Explicaremos esta técnica na seção 3.5.

3.2 Especificação de um problema

O TILDE assume que cada exemplo é um banco de dados pequeno ou uma parte do banco de dados completo. Todo exemplo é um conjunto de *fatos*, que codificam propriedades dos exemplos. Dado um conjunto finito de classes, cada exemplo é associado a uma classe. Além disso, também pode ser fornecido ao sistema um conhecimento preliminar em forma de um programa Prolog.

Para o melhor entendimento, voltemos ao problema Bongard, cujos exemplos estão exibidos na figura 3.2. Supondo que os exemplos são representados de acordo com a sua forma, configuração (se ele aponta para cima ou para baixo, o que vale apenas para triângulos) e posição relativa (objetos podem estar dentro de outros objetos), as figuras podem ser representadas da seguinte maneira:

Exemplo 1: { circulo(o1), triangulo(o2), aponta(o2, cima), dentro(o2, o1)}
Exemplo 2: { circulo(o3), triangulo(o4), aponta(o4, cima), triangulo(o5),
aponta(o5, baixo), dentro(o4, o5)}
etc.

Vale ressaltar que o_i são constantes usadas para auxiliar na representação dos objetos geométricos e das relações entre eles, mas elas não aparecerão nos testes dos nós internos da árvore que será gerada pelo sistema [3].

Também pode ser fornecido ao TILDE um conhecimento preliminar na forma de um programa Prolog. Um exemplo disso pode ser visto a seguir, em que são definidos os predicados *poligono*(O) e *dois_triangulos*($O1, O2$).

poligono(O) :- triangulo(O).
poligono(O) :- quadrado(O).
dois_triangulos(O1,O2) :- triangulo(O1), triangulo(O2), O1≠O2.

Ao se considerar um exemplo com o conhecimento preliminar é possível deduzir outros fatos para o mesmo. Por exemplo, ao se analizar os fatos do exemplo 2 com as possíveis regras do conhecimento preliminar listadas anteriormente, podemos considerar como verdadeiros para este exemplo os fatos *dois_triangulos*($o4, o5$) e *poligono*($o4$) [3].

Um problema é especificado no TILDE através de 3 arquivos: o arquivo com extensão *.kb*, que informa os exemplos; o arquivo *.bg*, que contém a teoria preliminar e é opcional; e o arquivo *.s*, que define algumas configurações, opções do usuário, o operador de refinamento (que será explicado na seção 3.4), além de informar, por exemplo, o conjunto de classes do problema. Para maiores informações sobre esses arquivos, veja [15]. No caso do exemplo do Bongard, poderíamos

colocar o programa Prolog mostrado anteriormente no arquivo *bongard.bg*. Os exemplos seriam informados pelo arquivo *bongard.kb*, que é ilustrado a seguir:

```
begin(model(1)).          begin(model(2)).
pos.                      pos.
circulo(o1).              circulo(o3).
triangulo(o2).            triangulo(o4).
aponta(o2, cima).         aponta(o4, cima).
dentro(o2, o1).           triangulo(o5).
end(model(1)).            aponta(o5, baixo).
                          dentro(o4, o5).
etc.                      end(model(2)).
```

Neste arquivo, cada exemplo começa com a linha *begin(model(Id))*, onde *Id* é um identificador para o exemplo, e termina com uma linha *end(model(Id))*. Entre as linhas *begin* e *end*, são informados a classe do exemplo e os fatos do mesmo.

É interessante destacar que o problema dos trens, que foi abordado no capítulo 2, seria representado no TILDE de forma diferente da que foi exibida naquele capítulo, em que foi usada a representação do Aleph [38]. No TILDE, os trens com nomes **leste***i* corresponderiam aos exemplos da classe *pos* e os com nomes **oeste***i*, aos exemplos da classe *neg*. Não seria fornecido nenhum programa Prolog como conhecimento preliminar (lembre-se que ele é opcional) e o que foi indicado como conhecimento preliminar na representação do Aleph no capítulo 2 corresponderia aos fatos dos exemplos na representação usada pelo TILDE. Por exemplo, todos os fatos que dizem respeito ao trem **leste1** (ou seja, os predicados **tem_vagao(leste1,vagao_1***j*) e os que envolvem os vagões **vagao_1***j*) seriam colocados entre linhas *begin(model(Id))* e *end(model(Id))*. Após a linha *begin(model(Id))*, haveria uma linha *pos*, referente à classe do exemplo.

A especificação de um problema no TILDE é dada de maneira mais formal na definição 3.1 e é conhecida em ILP como *aprendizado a partir de interpretações* [37].

Definição 3.1 (Aprendizado a partir de interpretações). *Sejam dados:*

- *Um conjunto de classes C (cada classe c é um predicado sem termos).*
- *Um conjunto de exemplos E classificados (cada elemento de E é da forma (e,c), onde e é um conjunto de fatos e c é uma classe).*
- *Uma teoria preliminar B.*

  Deseja-se achar:

- *Uma hipótese H (um programa Prolog), tal que para todo (e,c) $\in E$,*
  - $H \wedge e \wedge B \models c$, e
  - $\forall c' \in C$ - $\{c\}$: $H \wedge e \wedge B \not\models c'$

Uma interpretação é um conjunto de fatos. É importante ressaltar que uma hipótese implícita desta especificação é que a classe de um exemplo depende apenas dele e não de qualquer outro exemplo, o que possibilita que seja utilizado o teste de cobertura local. Essa é uma hipótese razoável para muitos problemas de classificação, mas impossibilita, por exemplo, o aprendizado de predicados recursivos como $pertence(E, L)$, explicado anteriomente na seção 3.1. Este tipo de problema não ocorre em sistemas que utilizam o *aprendizado a partir de implicação lógica*, exibido na definição 2.1, que é a forma tradicional de aprendizado em ILP.

Percebe-se, portanto, que o aprendizado a partir de interpretações é menos poderoso do que o aprendizado a partir de implicação lógica, mas é mais eficiente. Além disso, em [3] afirma-se que, apesar das restrições, o aprendizado a partir de interpretações é suficiente para maioria das aplicações práticas. O leitor interessado em maiores detalhes é convidado a consultar [37], em que é feito um estudo sobre os diferentes tipos de aprendizado e as relações entre eles.

3.3 Indução de árvores de decisão de lógica de primeira ordem

Nesta seção, alguns conceitos necessários para o melhor entendimento do TILDE serão apresentados e definidos. Após isto, finalmente, o algoritmo usado para a geração da árvore será mostrado.

Um conceito importante que tem sido usado no decorrer deste capítulo é o de árvore de decisão de lógica de primeira ordem, também chamada de *FOLDT* (*First Order Logical Decision Tree*). A definição 3.2 apresenta este conceito formalmente.

Definição 3.2 (Árvore de decisão de lógica de primeira ordem (*FOLDT*) [3] [2]). *Uma árvore de decisão de lógica de primeira ordem é uma árvore binária em que:*

- *os nós contêm uma conjunção de literais, e*
- *nós diferentes podem compartilhar variáveis, desde que as introduzidas em um determinado nó N (ou seja, variáveis que não existiam nos nós ancestrais de N) não ocorram no ramo direito do mesmo.*

Um exemplo de *FOLDT* pode ser visto na figura 3.1. A restrição imposta no segundo item da definição ocorre devido à semântica da árvore. Quando uma variável X é introduzida em um nó N, ela é quantificada existencialmente na conjunção do nó. Na subárvore direita de N, esta conjunção tem valor $Falso$, o que significa "não existe tal X". Neste ramo, portanto, X não tem significado [3] [2] e não faz sentido fazer referência a ela. Esta questão deve ficar mais clara com um exemplo e, por isso, voltaremos a ela em breve.

Uma árvore de decisão de lógica de primeira ordem pode ser convertida em um programa Prolog. Isto é feito de forma análoga à conversão de uma árvore de decisão tradicional para regras: cada regra é obtida ao se percorrer o caminho da raiz até uma folha e fazendo a conjunção dos testes dos nós internos. As regras obtidas desta forma para a árvore da figura 3.1 são exibidas a seguir:

SE triangulo(X) = *Verdadeiro* E dentro(X,Y) = *Verdadeiro* ENTÃO
  classe = pos
SENÃO
  SE triangulo(X) = *Verdadeiro* E dentro(X,Y) = *Falso* ENTÃO
    classe = neg
  SENÃO
    SE triangulo(X) = *Falso* ENTÃO
      classe = neg

O programa Prolog equivalente à árvore da figura 3.1 pode ser visto a seguir e é interessante notar que todas as cláusulas, exceto a última, terminam com um *cut* (representado por um "*!*"). Isto significa que, ao se avaliar um exemplo, se ele for coberto por uma cláusula do programa Prolog (ou seja, se o exemplo tornar verdadeiro o corpo dela), as seguintes não serão avaliadas. O programa corresponde, portanto, a uma lista de decisão de primeira ordem, o que captura a semântica das regras da forma SE-ENTÃO-SENÃO exibidas anteriormente.

```
classe(pos) :- triangulo(X), dentro(X,Y), !.
classe(neg) :- triangulo(X), !.
classe(neg).
```

Observando a árvore da figura 3.1 e o programa Prolog equivalente, fica mais fácil entender a restrição imposta às variáveis compartilhadas, que foi enunciada na definição 3.2. Considere a cláusula "*classe(neg) :- triangulo(X), !.*"e o caminho na árvore a que ela corresponde, formado pelo ramo esquerdo de "*triangulo(X)*"e pelo direito de "*dentro(X, Y)*". Note que, na segunda cláusula do programa Prolog, não aparece a variável Y, o que era esperado: a restrição diz que a variável Y não poderia aparecer no ramo direito do nó cujo teste é "*dentro(X, Y)*", porque neste ramo o teste tem valor *Falso*, o que faz esta variável não ser quantificada existencialmente. Por isso, não faria sentido fazer referência à variável Y neste ramo nem ela aparecer na cláusula correspondente do programa Prolog, o que de fato não ocorre.

O algoritmo 3.1 mostra como uma árvore de decisão de lógica de primeira ordem é usada para se classificar um exemplo. É importante fazer uma breve

observação sobre a notação utilizada. Um nó N da árvore ou é uma folha cuja classe é c ou é um nó interno cujo teste é a conjunção de literais $conj$. No caso em que o nó é uma folha, nos referimos a ele como $N = \textbf{folha}(c)$; no caso em que é um nó interno, indicamos $N = \textbf{noInterno}(conj, esq, dir)$, onde esq e dir são os nós filhos de N (respectivamente, os filhos esquerdo e direito).

Algoritmo 3.1 Algoritmo de classificação de um exemplo usando um *FOLDT* (com um conhecimento preliminar B), proposto em [3] [2]

Entrada:
e é um exemplo.

Saída:
c é uma classe.

Procedimento Classifica (e)

1: Seja Q a consulta Prolog $Verdadeiro$.
2: Seja N o nó raiz da árvore.
3: Enquanto $N \neq \textbf{folha}(c)$
4: Seja $N = \textbf{noInterno}(conj, esq, dir)$.
5: Se $Q \wedge conj$ for verdadeira em $e \wedge B$, então
6: Faça $Q = Q \wedge conj$.
7: Faça $N = esq$.
8: senão
9: Faça $N = dir$.
10: Retorne c (a classe da folha).

Como um exemplo e é um programa Prolog, realizar um teste em um nó N corresponde a verificar se uma consulta Prolog $\leftarrow C$ é verdadeira em $e \wedge B$, sendo B o conhecimento preliminar e $\leftarrow$ apenas uma notação utilizada para explicitar que C é uma consulta. É importante ressaltar que não é suficiente utilizar apenas a conjunção $conj$ do nó, uma vez que $conj$ pode compartilhar variáveis com conjunções de nós ancestrais de N. Por isso, C é formada por várias conjunções que aparecem no caminho da raiz até o nó em questão. Mais especificamente, C é da forma $Q \wedge conj$, onde $conj$ é a conjunção de N e Q é a conjunção de todos os testes dos nós que estão no caminho da raiz até o nó atual cujos ramos esquerdos foram escolhidos. Chamamos $\leftarrow Q$ de *consulta associada* do nó. Por isso, conforme visto no algoritmo 3.1, quando um exemplo vai para o ramo esquerdo de um nó, Q é

atualizada com a adição da conjunção *conj*. Quando um exemplo vai para o ramo direito, como o teste do nó é falso no exemplo, ele não introduz novas variáveis e Q não precisa ser atualizada [3] [2].

Árvores de decisão de lógica de primeira ordem podem ser induzidas da mesma forma que árvores de decisão proposicionais. O algoritmo do TILDE é baseado em um algoritmo tradicional de árvores de decisão, mais precisamente no C4.5 [35]. A principal diferença entre os dois algoritmos é o conjunto de testes que são considerados em cada nó. Enquanto no C4.5 os candidatos a teste do nó são comparações entre um atributo e um valor, no TILDE eles são definidos pelo usuário através de um operador de refinamento ρ. Este operador especifica que literais ou conjunções de literais podem ser adicionados à consulta associada do nó [3]. Como o usuário define este operador será explicado na seção 3.4.

Na tabela 3.1, ressaltamos as diferenças de um algoritmo de árvore de decisão tradicional para o do TILDE. Exibimos os pseudo-códigos em alto nível, demonstrando como ocorre a escolha de um atributo para ser colocado como teste de um nó interno nos dois algoritmos. Vale ressaltar que a classe de uma folha é dada pela classe majoritária dos exemplos de treinamento que caíram nela. Os passos que não estão exibidos do TILDE (1 e 5) são iguais aos do algoritmo tradicional. Finalmente, exibimos o pseudo-código do TILDE em maiores detalhes no algoritmo 3.2.

3.4 Operador de refinamento

Para refinar um nó com a consulta associada $\leftarrow Q$, o TILDE calcula $\rho(\leftarrow Q)$ e escolhe a consulta $\leftarrow Q_b \in \rho(\leftarrow Q)$ que gere a melhor divisão para o nó. A melhor divisão é a que maximiza algum critério de qualidade, que pode ser o ganho de informação ou a taxa de ganho de informação, que foram explicadas no capítulo 2. A conjunção colocada no nó é Q_b - Q, ou seja, os literais que foram adicionados à cláusula associada Q para produzior Q_b [3].

Algoritmo 3.2 Algoritmo do TILDE, para indução de árvores de decisão de lógica de primeira ordem, proposto em [3] [2]

Entrada:

E é um conjunto de exemplos,
Q é uma consulta Prolog.

Saída:

T é uma árvore de decisão de lógica de primeira ordem.

Procedimento ConstroiArvore (E, Q)

1: Seja $\leftarrow Q_b$ o elemento de $\rho(\leftarrow Q)$ com o maior valor de ganho.
2: Se $\leftarrow Q_b$ não for bom (por exemplo, tiver ganho nulo), então
3: Faça T = folha(classe_majoritaria(E)).
4: senão
5: Faça $conj = Q_b$ - Q.
6: Faça $E1 = \{e \in E | \leftarrow Q_b$ é verdadeira em $e \wedge B\}$.
7: Faça $E2 = \{e \in E | \leftarrow Q_b$ é falsa em $e \wedge B\}$.
8: Faça esq = ConstroiArvore($E1$, Q_b).
9: Faça dir = ConstroiArvore($E2$, Q).
10: Faça T = noInterno($conj$, esq, dir).
11: Retorne T.

Entrada:

E é um conjunto de exemplos.

Saída:

T é uma árvore.

Procedimento Tilde (E)

1: Seja T = ConstroiArvore(E, $Verdadeiro$).
2: Retorne T.

Árvore de decisão tradicional	TILDE
1. Raiz: *Exs* = todos os exemplos de treinamento.	
2. Baseado no conjunto de exemplos *Exs*, escolha o melhor atributo A para ser colocado como teste do nó.	⇒ 2. Baseado no conjunto de exemplos *Exs*, escolha o melhor refinamento R para ser colocado como teste do nó.
3. Crie um ramo para cada valor vi do atributo A, correspondente ao teste $A = vi$.	⇒ 3. Crie dois ramos: um correspondente ao teste $R = Verdadeiro$ e o outro a $R = Falso$.
4. Divida os exemplos entre os nós filhos, de acordo com o valor do atributo A, gerando os conjuntos de exemplos *Exs* de cada filho.	⇒ 4. Divida os exemplos entre os nós filhos, de acordo com o valor do refinamento R, gerando os conjuntos de exemplos *Exs* de cada filho.
5. Repita 2 para cada filho.	

Tabela 3.1: Diferenças entre os algoritmos em alto nível de uma árvore de decisão tradicional e do TILDE.

Para se entender melhor o operador de refinamento, é interessante usar um exemplo. Considere a árvore da figura 3.1 e que o nó raiz já tenha o teste $triangulo(X)$. Suponha que o filho esquerdo da raiz não tenha ainda nenhum teste e que desejamos calcular os possíveis refinamentos para ele. A consulta associada deste nó é $\leftarrow triangulo(X)$ e, portanto, o TILDE gera $\rho(\leftarrow triangulo(X))$. Dependendo da especificação do operador de refinamento dada pelo usuário, o resultado disto pode ser visto na figura 3.3, em que ";"foi utilizado para separar elementos de ρ e que ","foi usado para denotar uma conjunção em Prolog. Assumindo que o melhor refinamento é $Q_b = triangulo(X), dentro(X, Y)$, a conjunção que é colocada no nó é $Q_b - Q = dentro(X, Y)$ [3].

O algoritmo 3.3 mostra como o TILDE escolhe o melhor refinamento para um nó. Note que isto é feito de forma análoga à maneira que as árvores de decisão proposicionais escolhem um atributo para um nó. A diferença é que, como são considerados refinamentos, os únicos valores possíveis são *Verdadeiro* e *Falso*. Os contadores $cont[Verdadeiro]$ e $cont[Falso]$ são distribuições de classes

$$\rho(triangulo(X)) = \{\leftarrow triangulo(X), dentro(X,Y);$$
$$\leftarrow triangulo(X), dentro(Y,X);$$
$$\leftarrow triangulo(X), quadrado(Y);$$
$$\leftarrow triangulo(X), circulo(Y)\}$$

Figura 3.3: Exemplo de possíveis refinamentos para o filho esquerdo da raiz da árvore da figura 3.1.

(ou seja, mapeamentos que indicam quantos exemplos pertencem a cada classe), assim como em árvores de decisão proposicionais. Um exemplo de distribuição de classes é S_{Fraco} = [6+, 2-], que aparece no exemplo da figura 2.3. Vale ressaltar que *entropia_media_ponderada*, que aparece na linha 10 do algoritmo 3.3, nada mais é do que o segundo termo da equação 2.3.

Algoritmo 3.3 Algoritmo que escolhe o melhor refinamento para um nó, proposto em [3]

Entrada:

Q é uma consulta Prolog.

Saída:

Q_b é o melhor refinamento de Q.

Procedimento MelhorRefinamento (Q)

1: Para cada refinamento $\leftarrow Q_i \in \rho(\leftarrow Q)$
2: Para cada classe c
3: Faça $cont[Verdadeiro][c] = 0$.
4: Faça $cont[Falso][c] = 0$.
5: Para cada exemplo e
6: Se $\leftarrow Q_i$ for verdadeira em $e \wedge B$, então
7: Incremente $cont[Verdadeiro][classe(e)]$ em 1.
8: Senão
9: Incremente $cont[Falso][classe(e)]$ em 1.
10: Faça s_i = entropia_media_ponderada($cont[Verdadeiro]$, $cont[Falso]$).
11: Faça $Q_b = Q_i$ para o qual s_i é mínimo (ou seja, Q_i cujo ganho é máximo).
12: Retorne Q_b.

A forma como o operador de refinamento do TILDE é definido pelo usuário é baseada no PROGOL [Muggleton, 1995]. Para finalizar o capítulo, apresentaremos brevemente como ele pode ser definido. Para maiores informação, veja [3], [2], [15].

Através de fatos da forma *rmode(n: conj)*, onde *conj* é uma conjunção, o usuário indica que conjunções podem ser adicionadas à consulta associada do nó. O parâmetro n indica o número máximo de vezes que a conjunção *conj* pode ocorrer em uma consulta. Este parâmetro é opcional, ou seja, se a declaração for da forma *rmode(conj)*, não existe limite máximo para este número [3]. Um exemplo de declaração de *rmodes* para o operador de refinamento do problema Bongard pode ser visto a seguir:

```
rmode(5: triangulo(+-V)).
rmode(5: quadrado(+-V)).
rmode(5: circulo(+-V)).
rmode(5: dentro(+V, +-W)).
rmode(5: dentro(-V, +W)).
rmode(5: aponta(+V, cima)).
rmode(5: aponta(+V, baixo)).
```

Através dos *rmodes*, também são definidos os *modos* das variáveis. O modo de uma variável é indicado por um "-", "+"ou um "+-"antes da mesma. O símbolo "-"antes da variável V indica que ela é uma variável de saída, ou seja, ela deve ser nova, não podendo existir na consulta associada do nó em que o novo teste será colocado. A notação $+V$ define que V é uma variável de entrada, o que significa que ela já deve aparecer na consulta associada do nó. Finalmente, $+\text{-}V$ indica que ela pode ser de saída ou de entrada, ou seja, ela pode ser uma nova variável ou já existir na consulta [3].

Para entender melhor como os *rmodes* são utilizados, vamos considerar novamente a árvore da figura 3.1 e a declaração de *rmodes* descrita anteriormente. Durante o processo de escolha de que teste será colocado no nó raiz, o TILDE considera apenas os refinamentos *triangulo(X)*, *quadrado(X)* e *circulo(X)*, porque todos os outros testes exigem que alguma variável exista na consulta associada do nó, o que não ocorre na consulta associada da raiz, que é $Verdadeiro$. O

literal *triangulo(X)* foi considerado o melhor refinamento e, sendo assim, foi colocado como o teste do nó raiz. Já o filho esquerdo da raiz tem consulta associada $\leftarrow triangulo(X)$, que contém uma variável X. Por isso, neste nó, mais refinamentos podem ser considerados. Os possíveis testes deste nó são exibidos a seguir:

triangulo(X)	quadrado(X)	circulo(X)
triangulo(Y)	quadrado(Y)	circulo(Y)
dentro(X,Y)	dentro(Y,X)	
aponta(X,cima)	aponta(X,baixo)	

O refinamento *dentro(X, Y)* foi o que proporcionava a melhor divisão para o nó e, por isso, foi escolhido como o teste do mesmo [3].

Outro conceito importante utilizado no TILDE é o de *tipos* das variáveis. Quando um novo literal é adicionado, as variáveis dele só podem ser unificadas com variáveis que sejam do mesmo tipo [2]. Um exemplo de declaração de tipos para o problema Bongard é o seguinte:

```
type(triangulo(objeto)).
type(quadrado(objeto)).
type(circulo(objeto)).
type(dentro(objeto,objeto)).
type(aponta(objeto,direcao)).
```

Neste exemplo, indica-se que a variável que aparecer no literal *triangulo* deve ser do tipo *objeto*, assim como as duas variáveis do literal *dentro*. Note, porém, que a segunda variável do literal *aponta* é de um outro tipo (*direcao*). Isto impede que, por exemplo, dada a consulta associada $triangulo(X)$, $circulo(Y)$, o literal $aponta(X, Y)$ seja considerado como um possível refinamento. Como a variável Y é do tipo *objeto*, por ter aparecido como argumento do literal *circulo*, ela não poderia aparecer como o segundo argumento de *aponta*, que deve ser do tipo *direcao*.

Além de modos e tipos de variáveis, o usuário pode especificar *lookaheads*. Desta forma, o TILDE pode considerar não só os refinamentos da consulta associada do nó em questão, mas os refinamentos destes refinamentos. Isto pode

ser útil porque às vezes um refinamento pode não proporcionar nenhum ganho de informação, mas pode introduzir variáveis novas e importantes. Por exemplo, a declaração *lookahead*(*triangulo*(T), *aponta*(T, *cima*)) define que, sempre que o literal *triangulo*(T) for um teste candidato para ser adicionado a uma consulta associada, a conjunção *triangulo*(T), *aponta*(T, *cima*) também deve ser considerada, no mesmo passo do algoritmo. A importância do *lookahead* pode ser verificada na seguinte situação: o TILDE normalmente construiria o teste *triangulo*(T), *aponta*(T, *cima*) colocando no nó N atual o literal *triangulo*(T) e, depois, no seu filho esquerdo, o teste *aponta*(T, *cima*). Entretanto, se *triangulo*(X) já ocorresse na consulta associada de N, o refinamento *triangulo*(T) não proporcionaria nenhum ganho. Isto ocorreria porque, como já saberíamos que existia um triângulo (no caso, X), a nova pergunta "existe um triângulo?" não nos daria nenhuma informação nova. Como *triangulo*(T) teria ganho nulo, nem ele seria adicionado ao nó N nem o teste *aponta*(T, *cima*) seria colocado no seu filho esquerdo [3]. Isso poderia evitar, por exemplo, que construíssemos a cláusula *triangulo*(X), *aponta*(X, *baixo*), *triangulo*(T), *aponta*(T, *cima*), que descreveria o caso em que existissem dois triângulos na figura: um apontando para cima e o outro, para baixo.

Para finalizar, é importante ressaltar que os conceitos de modos, tipos e *lookahead* que foram apresentados nesta seção não foram criados pelo TILDE. Eles são muito utilizados em vários sistemas ILP, como o Aleph [38]. No TILDE, o usuário coloca estas declarações no arquivo com extensão *.s*, já citado anteriormente.

3.5 Pacotes de cláusulas

Nesta seção, explicaremos brevemente o conceito de pacotes de cláusulas (*query packs*), que é uma das ferramentas usadas pelo TILDE para diminuir o custo da prova de exemplos em ILP. Para maiores informações, veja [1].

O TILDE utiliza os pacotes de cláusulas como forma de acelerar a prova de um grupo de cláusulas que tenham um prefixo em comum. Para facilitar a compreensão, é interessante utilizar um exemplo. Suponha que tenhamos o conjunto de cláusulas exibido na figura 3.4 e, dado um exemplo e, desejamos saber quais são verdadeiras e quais são falsas para este exemplo.

```
1: p(X).
2: p(X), q(X,a).
3: p(X), q(X,b).
4: p(X), q(X,c).
5: p(X), q(X,Y), t(X).
6: p(X), q(X,Y), t(X), r(Y,1).
7: p(X), q(X,Y), t(X), r(Y,2).
```

Figura 3.4: Cláusulas que desejamos saber se cobrem ou não o exemplo e.

A forma tradicional de se testar se estas cláusulas cobrem ou não o exemplo seria avaliá-las individualmente, ou seja, passaríamos cada uma delas para o Prolog e verificaríamos se ela obteve sucesso ou falha. Note que todas elas têm o mesmo prefixo $p(X)$, que deverá ser provado uma vez para cada cláusula (sete vezes para este exemplo). Além disso, o prefixo $p(X), q(X,Y), t(X)$ será avaliado três vezes: uma para a cláusula 5, outra para a 6 e uma terceira vez para a 7.

Como forma de reaproveitar a prova destes prefixos, o TILDE utiliza um pacote de cláusulas, que pode ser visto como uma árvore. A árvore exibida na figura 3.5 corresponde ao pacote de cláusulas formado pelas cláusulas da figura 3.4. Cada caminho da raiz até uma das folhas corresponde a uma cláusula. Para facilitar a visualização, embaixo de cada uma das folhas está o número da cláusula da figura 3.4 correspondente.

Enquanto na forma tradicional o prefixo $p(X)$ seria avaliado várias vezes (uma por cláusula), com o pacote de cláusulas, ele seria avaliado uma única vez, assim como $p(X), q(X,Y), t(X)$. Além disso, os autores dos pacotes de cláusulas fizeram modificações no próprio Prolog utilizado pelo TILDE, de forma a realizar otimizações para tornar este método ainda mais eficiente.

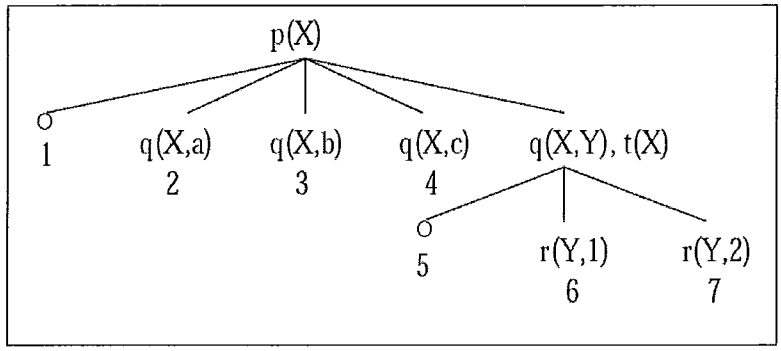


Figura 3.5: Exemplo do pacote de cláusulas correspondente às cláusulas da figura 3.4.

Os pacotes de cláusulas são muito adequados ao TILDE porque, durante a indução da árvore, ocorrem muitos casos em que é necessário avaliar um exemplo em cláusulas que têm prefixos em comum. Particularmente, durante o processo de escolha do melhor refinamento a ser colocado como teste de um nó, para cada exemplo que caiu na folha, é necessário avaliar todos os possíveis refinamentos para o nó. Todos estes possíveis refinamentos têm um prefixo em comum: a consulta associada do nó. Um exemplo disto pode ser visto na figura 3.3, em que todos os refinamentos têm o prefixo $triangulo(X)$, que é a consulta associada do nó em questão.

Além disso, dependendo dos *lookaheads* utilizados, alguns refinamentos podem ter outros prefixos em comum. Por exemplo, quando explicamos este conceito na seção 3.4, utilizamos como exemplo o comando $lookahead(triangulo(T), aponta(T, cima))$, que define que quando o literal $triangulo(T)$ for um teste candidato para ser adicionado a uma consulta associada, a conjunção $triangulo(T)$, $aponta(T, cima)$ também deve ser testada. Desta forma, se o nó em questão tivesse como consulta associada $\leftarrow circulo(X)$, os possíveis refinamentos poderiam ser os indicados na figura 3.6. Note que todos eles têm o prefixo $circulo(X)$ em comum e que os dois últimos têm $circulo(X)$, $triangulo(Y)$ em comum, devido ao *lookahead* citado anteriormente. Neste caso, o pacote de cláusulas melhoraria o desempenho da por provar uma única vez não só a consulta associada, mas o prefixo $circulo(X)$, $triangulo(Y)$ das duas últimas cláusulas.

$$\begin{aligned}\rho(circulo(X)) = \{&\leftarrow circulo(X), dentro(X,Y);\\ &\leftarrow circulo(X), dentro(Y,X);\\ &\leftarrow circulo(X), quadrado(Y);\\ &\leftarrow circulo(X), triangulo(Y);\\ &\leftarrow circulo(X), triangulo(Y), aponta(Y, cima)\}\end{aligned}$$

Figura 3.6: Exemplo de possíveis refinamentos para um nó com consulta associada $\leftarrow circulo(X)$.

Capítulo 4

VFDT - Very Fast Decision Tree

Neste capítulo, será apresentado o VFDT (*Very Fast Decision Tree*) [11] [18], que é um dos dois sistemas que serviram de base para o nosso trabalho. Na seção 4.1, faremos uma introdução ao método. Já na seção 4.2, apresentaremos o algoritmo de indução de árvores de Hoeffding, que serve de base para o VFDT. Na seção 4.3, exibiremos uma importante propriedade destas árvores e a sua prova. Concluímos o capítulo com a seção 4.4, que mostra algumas diferenças de implementação do VFDT em relação ao algoritmo de árvores de Hoeffding.

4.1 Introdução

O VFDT é um dos sistemas proposicionais de aprendizado de máquina que utilizam a metodologia proposta por Domingos e Hulten em [13] para lidar com grandes bases de dados.

Ele foi proposto em [11] e classifica exemplos através da geração de uma árvore de decisão proposicional, ou seja, que utiliza atributos como testes dos nós, assim como as árvores de decisão tradicionais, que foram explicadas na seção 2.1. Entretanto, ao contrário delas, o algoritmo do VFDT é escalável para grandes bases de dados, por utilizar o *limite de Hoeffding*, que é um resultado estatístico que será explicado na seção 4.2.

Além disso, ele difere das árvores de decisão mostradas no capítulo 2 por ser um algoritmo incremental e *anytime*. Isto significa que novos exemplos podem ser

incorporados à árvore, modificando-a, e um modelo pode ser criado utilizando-se poucos exemplos e, progressivamente, ser refinado [11]. Note que isto é diferente das árvores de decisão tradicionais, em que um modelo só pode ser construído considerando-se todos os exemplos e, se novos forem adicionados, ele deve ser descartado e uma nova árvore ser construída.

4.2 Árvore de Hoeffding

O algoritmo de indução de árvores de Hoeffding difere dos que geram árvores de decisão tradicionais por ser incremental e utilizar um subconjunto dos exemplos de treinamento que caem em um nó para escolher o melhor atributo para ser colocado como teste do mesmo.

O ID3, apresentado no capítulo 2, é um algoritmo *batch*. Na raiz, são utilizados todos os exemplos de treinamento para se calcular o valor da heurística de cada atributo (por exemplo, o ganho de informação do mesmo) e, então, se escolher o melhor, que será colocado como teste do nó. Os exemplos são divididos entre os filhos e o processo se repete em cada um deles. Já no algoritmo de indução de árvores de Hoeffding, cada exemplo é usado uma única vez. Os primeiros exemplos são usados para se selecionar o teste da raiz. Uma vez escolhido este atributo, os exemplos seguintes serão passados para as folhas apropriadas e serão usados para escolher o melhor atributo delas, e assim por diante [11].

Para decidir exatamente quantos exemplos são necessários em cada folha para ela sofrer um *split* (ou seja, o melhor atributo dela ser colocado como teste e ela ser transformada em um nó interno), é usado um resultado estatístico chamado *limite de Hoeffding*, exibido na definição 4.1. O limite de Hoeffding também é chamado de *limite de Chernoff aditivo*.

Definição 4.1 (Limite de Hoeffding). *Seja r uma variável aleatória com imagem de tamanho R. Por exemplo, se r for uma variável binária com imagem 0 ou 1, R=1; se r for o ganho de informação,* $R = \log(c)$*, onde c é o número de classes.*

Suponha que façamos n observações independentes de r e calculemos a média $\overline{r}$. O limite de Hoeffding afirma que $P(\mu_r \geq \overline{r} - \epsilon) = 1 - \delta$, onde

- *μ_r é a verdadeira média da variável r,*
- *ϵ é dado pela equação 4.1 e*
- *δ é um número muito baixo.*

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}} \tag{4.1}$$

O limite de Hoeffding tem a característica importante de ser independente da distribuição de probabilidade de r. Esta generalidade faz com que ele necessite de mais observações, para chegar aos mesmos δ e ϵ, do que outros limites específicos para uma distribuição de probabilidade. Porém, ele tem a vantagem de poder ser aplicado em várias situações, sem que precisemos nos preocupar com a distribuição da variável aleatória em questão.

Em árvores de Hoeffding, a equação 4.1 é utilizada para se escolher o melhor teste para um nó. Suponha que G seja a heurística usada para selecionar o melhor atributo para o nó em questão, cujo valor deve ser maximizado (por exemplo, o ganho de informação). Seja $G(A)$ o valor desta heurística para o atributo A.

Sejam X_a e X_b, respectivamente, os atributos com o maior e o segundo maior valores observados $\overline{G}$ da heurística, após serem vistos n exemplos no nó considerado. Seja $\Delta\overline{G} = \overline{G}(X_a) - \overline{G}(X_b)$ a diferença entre os valores observados da heurística dos atributos X_a e X_b, respectivamente. Seja $\Delta G_r = G(X_a) - G(X_b)$ a diferença entre os valores verdadeiros da heurística dos mesmos atributos.

Pelo limite de Hoeffding, sabemos com probabilidade $1-\delta$ que $\Delta G_r \geq \Delta\overline{G} - \epsilon$, onde ϵ é dado pela equação 4.1. Portanto, se obtivermos $\Delta\overline{G} > \epsilon$, podemos dizer, com a mesma probabilidade, que $\Delta G_r > 0$. Isto significa que $G(X_a) > G(X_b)$, ou seja, o valor verdadeiro da heurítica de X_a é maior do que o valor verdadeiro de G

para o atributo X_b. Em outras palavras, com probabilidade $1 - \delta$, pode-se afirmar que X_a é de fato o melhor atributo.

Resumindo, se obtivermos $\Delta\overline{G} > \epsilon$, garantimos que X_a é o melhor atributo para o nó, com probabilidade $1 - \delta$. Sendo assim, uma folha de uma árvore de Hoeffding só precisa acumular exemplos (ou melhor, estatísticas destes exemplos, como será explicado em breve) até que ϵ, dado pela equação 4.1, se torne menor do que $\Delta\overline{G}$. Neste ponto, a folha pode ser transformada em um nó interno, sendo o melhor atributo X_a escolhido como o teste do nó.

Vale ressaltar que, para chegarmos a esta conclusão, assumimos que o terceiro melhor atributo e os demais têm valor de heurística menor o suficiente para que a probabilidade de um deles ser o melhor atributo de verdade é desprezível. Além disso, assumimos que o valor observados da heurística $\overline{G}$ de um nó possam ser vistos como uma média dos valores de G para os exemplos do nó em questão, o que é o caso para as heurísticas usadas tipicamente, como o ganho de informação [11], [18].

Na tabela 4.1, exibimos as diferenças de um algoritmo de árvore de decisão tradicional para o de árvores de Hoeffding. De forma análoga ao que fizemos no capítulo 3, exibimos aqui os pseudo-códigos em alto nível, demonstrando como ocorre a escolha de um atributo para ser colocado como teste de um nó interno nos dois algoritmos e, novamente, ressaltamos que a classe de uma folha é dada pela classe majoritária dos exemplos de treinamento que caíram nela. Os passos que não estão exibidos do algoritmos de árvores de Hoeffding (2, 3 e 5) são iguais aos do tradicional.

O algoritmo de indução de uma árvore de Hoeffding é exibido em maiores deralhes no algoritmo 4.1. Os contadores n_{ijk} são as estatísticas dos exemplos da folha necessárias para se calcular a maioria das heurísticas usadas para escolher o melhor atributo de um nó, como o ganho de informação. $X_\emptyset$ é um "atributo nulo", que representa escolher não transformar a folha em nó interno.

Árvore de decisão tradicional	Árvore de Hoeffding
1. Raiz: *Exs* = todos os exemplos de treinamento.	⇒ 1. Raiz: *Exs* = os N primeiros exemplos de treinamento, N dado pelo limite de Hoeffding ($\Delta\overline{G} > \epsilon$).
2. Baseado no conjunto de exemplos *Exs*, escolha o melhor atributo A para ser colocado como teste do nó.	
3. Crie um ramo para cada valor *vi* do atributo A, correspondente ao teste $A = vi$.	
4. Divida os exemplos entre os nós filhos, de acordo com o valor do atributo A, gerando os conjuntos de exemplos *Exs* de cada filho.	⇒ 4. Descarte os exemplos já considerados para o *split*. Divida os próximos exemplos entre os nós filhos, de acordo com o valor do atributo A, gerando os conjuntos de exemplos *Exs* de cada filho. O número de exemplos de cada filho também será determinado pelo limite de Hoeffding.
5. Repita 2 para cada filho.	

Tabela 4.1: Diferenças entre os algoritmos em alto nível de uma árvore de decisão tradicional e de árvores de Hoeffding.

Achamos que as linhas 3 e 21 do algortimo podem não estar muito claras. A linha 3 pode levar a entender que o algoritmo prediz a classe mais freqüente dos exemplos em S, o que parece estranho, uma vez que ele se propõe a utilizar cada exemplo uma só vez. Já a linha 21 pode nos fazer pensar que ele deveria descobrir a classe majoritária dos exemplos que caíram na folha l_m, que acabou de ser criada. Ao analisarmos o código do VFDT, percebemos que, o que ocorre, é que cada vez que se chega a uma folha e se deseja saber a sua classe, ela é calculada como a majoritária dos seus exemplos. Já no caso da linha 3, uma classe padrão é retornada, uma vez que ainda não chegaram exemplos na raiz, mas este caso não faz muito sentido, uma vez que significaria pedir pra árvore predizer uma classe sem ter dado a ela nenhum exemplo de treinamento. Também é interessante dizer que $\overline{G_l}(X_\emptyset)$ é a entropia da folha l.

Algoritmo 4.1 Algoritmo de indução de uma árvore de Hoeffding, proposto em [11]

Entradas:

S é uma seqüência de exemplos,

$\mathbf{X}$ é um conjunto de atributos discretos,

$G(.)$ é uma função de avaliação de *split*,

δ é um menos a probabilidade desejada de se escolher o atributo correto em um dado nó.

Saída:

HT é uma árvore de decisão.

Procedimento HoeffdingTree (S, $\mathbf{X}$, G, δ)

Seja HT uma árvore com uma única folha l_1 (a raiz).
Seja $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$.
Seja $\overline{G_1}(X_\emptyset)$ o valor de $\overline{G}$ obtido ao se predizer a classe mais freqüente em S.
Para cada classe y_k
    Para cada valor x_{ij} de cada atributo $X_i \in \mathbf{X}$
        Faça $n_{ijk}(l_1) = 0$.
Para cada exemplo ($\mathbf{x}$, y_k) em S
    Desça ($\mathbf{x}$, y) até uma folha l usando HT.
    Para cada x_{ij} em $\mathbf{x}$ tal que $X_i \in \mathbf{X}_l$
        Incremente $n_{ijk}(l)$.
    Rotule a folha l com a classe majoritária dos exemplos que chegaram nela até agora.
    Se os exemplos que chegaram até agora em l não forem todos da mesma classe, então
        Calcule $\overline{G_l}(X_i)$ para cada atributo $X_i \in \mathbf{X}_l$ - $\{X_\emptyset\}$ usando os contadores $n_{ijk}(l)$.
        Faça X_a ser o atributo com o maior valor de $\overline{G_l}$.
        Faça X_b ser o atributo com o segundo maior valor de $\overline{G_l}$.
        Calcule ϵ usando a equação 4.1.
        Se $\overline{G_l}(X_a)$ - $\overline{G_l}(X_b) > \epsilon$ e $X_a \neq X_\emptyset$, então
            Substitua l por um nó interno cujo teste é o atributo X_a (ou seja, faça o *split* com o atributo X_a).
            Para cada ramo do *split*
                Adicione uma nova folha l_m, e faça $\mathbf{X}_m = \mathbf{X}$ - $\{X_a\}$.
                Faça $\overline{G_m}(X_\emptyset)$ ser o valor de $\overline{G}$ obtido ao se predizer a classe mais freqüente em l_m.
                Para cada classe y_k e cada valor x_{ij} de cada atributo $X_i \in \mathbf{X}_m$ - $\{X_\emptyset\}$
                    Faça $n_{ijk}(l_m) = 0$.
Retorne HT.

4.3 Propriedade das árvores de Hoeffding

Uma propriedade importante das árvores de Hoeffding é que se pode garantir que o algoritmo produz árvores muito próximas às que seriam geradas por um algoritmo *batch*, que utiliza todos os exemplos para escolher o teste a ser colocado em cada nó. A prova desta propriedade, extraída de [11], é mostrada a seguir.

Prova 4.1. *O algoritmo de árvores de Hoeffding produz árvores assintoticamente próximas às produzidas por um algoritmo batch.*

Para se provar esta propriedade, precisamos definir alguns conceitos. Primeiramente, vamos definir quando dois nós de árvores de decisão são diferentes e quando dois caminhos são diferentes.

Definição 4.2 (Desigualdade entre nós). *Dois nós de árvores de decisão são diferentes nos seguintes casos:*

- *Dois nós internos são diferentes quando os seus testes são diferentes.*
- *Duas folhas são diferentes quando predizem classes diferentes.*
- *Um nó interno é diferente de uma folha.*

Definição 4.3 (Desigualdade entre caminhos). *Dois caminhos de árvores de decisão são diferentes se eles diferem em tamanho ou em pelo menos um nó.*

É necessário também definir o conceito de *discordância* entre duas árvores de decisão. Sejam:

- $P(\mathbf{x})$ a probabilidade do vetor de atributos $\mathbf{x}$ ser observado.
- $I(.)$ uma função que retorna 1 se o seu argumento for verdadeiro e 0 em caso contrário.

Definição 4.4 (Discordância Extensional). *Seja $DT_i(\mathbf{x})$ a classificação dada pela árvore DT_i para o exemplo $\mathbf{x}$. A discordância extensional Δ_e entre duas árvores*

de decisão DT_1 e DT_2 é a probabilidade de elas predizerem classes diferentes para um mesmo exemplo:

$$\Delta_e(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[DT_1(\mathbf{x}) \neq DT_2(\mathbf{x})] \tag{4.2}$$

Definição 4.5 (Discordância Intensional). *Seja $Caminho_i(\mathbf{x})$ o caminho do exemplo $\mathbf{x}$ pela árvore DT_i. A discordância intensional Δ_i entre duas árvores de decisão DT_1 e DT_2 é a probabilidade do caminho de um exemplo pela árvore DT_1 ser diferente do caminho do mesmo exemplo pela árvore DT_2.*

$$\Delta_i(DT_1, DT_2) = \sum_{\mathbf{x}} P(\mathbf{x}) I[Caminho_1(\mathbf{x}) \neq Caminho_2(\mathbf{x})] \tag{4.3}$$

Duas árvores concordam de formal intensional em relação a um exemplo se este desce por cada uma das árvores pela mesma seqüência de nós e ambas predizem a mesma classe para ele. Vale notar que discordância intensional é um conceito mais forte do que discordância extensional.

Seja p_l a probabilidade de que um exemplo que desça até o nível l em uma árvore de decisão alcance uma folha neste nível. Para simplificar, assumiremos que esta probabilidade é constante para todas as folhas ($\forall l\ p_l = p$), o que é uma hipótese realista para árvores de decisão típicas. Chamaremos p de *probabilidade de folha*. Seja HT_δ a árvore de Hoeffding gerada com a probabilidade desejada δ, dada uma seqüência infinita de exemplos S. Seja DT_* a árvore de decisão assintótica produzida por um algoritmo *batch*, em que se escolhe em cada nó o atributo com maior valor de G, usando-se para isso infinitos exemplos em cada nó. Seja $E[\Delta_i(HT_\delta, DT_*)]$ o valor esperado de $\Delta_i(HT_\delta, DT_*)$, calculado considerando-se todas as infinitas seqüências de treinamento possíveis. Podemos afirmar o seguinte resultado:

Teorema 4.1. *Se HT_δ é a árvore de decisão de Hoeffding produzida com a probabilidade desejada δ dados infinitos exemplos, DT_* é a árvore de decisão assintótica*

produzida por um algoritmo batch e p é a probabilidade de folha, então $E[\Delta_i(HT_\delta, DT_*)] \leq \delta/p$.

Prova: Considere um exemplo $\mathbf{x}$ que, na árvore HT_δ, cai em uma folha no nível l_h e, na árvore DT_*, cai em uma folha no nível l_d. Sejam:

- $l = \min\{l_h, l_d\}$.
- $Caminho_H(\mathbf{x}) = (N_1^H(\mathbf{x}), N_2^H(\mathbf{x}), ..., N_l^H(\mathbf{x}))$ o caminho de $\mathbf{x}$ pela árvore HT_δ até o nível l, onde $N_i^H(\mathbf{x})$ é o nó em que o exemplo $\mathbf{x}$ cai no nível i em HT_δ. Analogamente, seja $Caminho_D(\mathbf{x})$ o caminho de $\mathbf{x}$ pela árvore DT_*. Se $l = l_h$, então $N_l^H(\mathbf{x})$ é uma folha e, portanto, prediz uma classe; de forma semelhante, se $l = l_d$, então $N_l^D(\mathbf{x})$ é uma folha.
- I_i a proposição "$Caminho_H(\mathbf{x}) = Caminho_D(\mathbf{x})$ até o nível i, inclusive", e $I_0 = true$. Note que $P(N_l^H(\mathbf{x}) \neq N_l^D(\mathbf{x})|I_{l-1})$ inclui $P(l_h \neq l_d)$, porque se os dois caminhos têm tamanhos diferentes, então uma árvore deve ter uma folha onde a outra tem um nó interno.

Então,

$$
\begin{aligned}
& P(Caminho_H(\mathbf{x}) \neq Caminho_D(\mathbf{x})) \\
&= P((N_1^H \neq N_1^D) \vee (N_2^H \neq N_2^D) \vee ...(N_l^H \neq N_l^D)) \\
&= P(N_1^H \neq N_1^D|I_0) + P(N_2^H \neq N_2^D|I_1) + ... + P(N_l^H \neq N_l^D|I_{l-1}) \\
&= \sum_{i=1}^{l} P(N_i^H \neq N_i^D|I_{i-1}) \leq \sum_{i=1}^{l} \delta = \delta l
\end{aligned} \tag{4.4}
$$

Seja $HT_\delta(S)$ a árvore de Hoeffding gerada pela seqüência de exemplos de treinamento S. Então $E[\Delta_i(HT_\delta, DT_*)]$ é a média, considerando-se todas as infinitas seqüências de treinamento S, da probabilidade do caminho de um exemplo na árvore $HT_\delta(S)$ ser diferente do caminho do mesmo exemplo na árvore DT_*:

$E[\Delta_i(HT_\delta, DT_*)]$

$$= \sum_S P(S) \sum_{\mathbf{x}} P(\mathbf{x}) I[Caminho_H(\mathbf{x}) \neq Caminho_D(\mathbf{x})]$$

$$= \sum_{\mathbf{x}} P(\mathbf{x}) P(Caminho_H(\mathbf{x}) \neq Caminho_D(\mathbf{x}))$$

$$= \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x}) P(Caminho_H(\mathbf{x}) \neq Caminho_D(\mathbf{x})) \tag{4.5}$$

onde L_i é o conjunto de exemplos que caem em uma folha de DT_* no nível i. De acordo com a equação 4.4, a probabilidade do caminho de um exemplo na árvore $HT_\delta(S)$ ser diferente do caminho do mesmo exemplo na árvore DT_*, dado que este último tem tamanho i, é no máximo δi (já que $i \geq l$). Então,

$$E[\Delta_i(HT_\delta, DT_*)] \leq \sum_{i=1}^{\infty} \sum_{\mathbf{x} \in L_i} P(\mathbf{x})(\delta i) = \sum_{i=1}^{\infty} (\delta i) \sum_{\mathbf{x} \in L_i} P(\mathbf{x}) \tag{4.6}$$

A soma $\sum_{\mathbf{x} \in L_i} P(\mathbf{x})$ é a probabilidade de um exemplo $\mathbf{x}$ cair em uma folha de DT_* no nível i e é igual a $(1-p)^{i-1}p$, onde p é a probabilidade de folha. Então,

$E[\Delta_i(HT_\delta, DT_*)]$

$$\leq \sum_{i=1}^{\infty} (\delta i)(1-p)^{i-1} p = \delta p \sum_{i=1}^{\infty} i(1-p)^{i-1}$$

$$= \delta p \left[\sum_{i=1}^{\infty} (1-p)^{i-1} + \sum_{i=2}^{\infty} (1-p)^{i-1} + \ldots + \sum_{i=k}^{\infty} (1-p)^{i-1} + \ldots \right]$$

$$= \delta p \left[\frac{1}{p} + \frac{1-p}{p} + \ldots + \frac{(1-p)^{k-1}}{p} + \ldots \right]$$

$$= \delta \left[1 + (1-p) + \ldots + (1-p)^{k-1} + \ldots \right]$$

$$= \delta \sum_{i=0}^{\infty} (1-p)^i = \frac{\delta}{p} \tag{4.7}$$

Isto completa a demonstração do Teorema 4.1.

4.4 VFDT

Vamos finalizar este capítulo explicando algumas diferenças entre o VFDT e o algoritmo de indução de árvores exibido no algoritmo 4.1. O sistema VFDT é uma implementação deste algoritmo e utiliza o ganho de informação como heurística padrão para escolher atributos. Além disso, ele usa algumas técnicas para melhorar o desempenho. A seguir, explicaremos as duas principais técnicas. Para obter maiores informações sobre elas e as demais, veja [11]

O VFDT detecta empates entre atributos. Se dois ou mais atributos têm valores de heurística muito parecidos, é necessário um grande número de exemplos para ser possível escolher qual deles é o melhor atributo. Isto pode ser considerado um desperdício, uma vez que faz pouca diferença se qualquer um deles for escolhido. Por isso, o VFDT pode utilizar um valor τ, definido pelo usuário, para detectar um empate entre atributos. Se $\Delta\overline{G} < \epsilon < \tau$, a folha sofre um *split* e o melhor atributo é escolhido como o teste do nó.

Além disso, o VFDT permite ao usuário definir um número e_{min}, que indica de quantos em quantos exemplos vistos em uma folha o valor da heurística será calculada. Este cálculo é uma parte custosa do algoritmo e, especificando este número, o usuário faz com que o VFDT não precise calcular este valor a cada novo exemplo, fazendo-o melhorar o seu desempenho.

Capítulo 5

HTILDE - Hoeffding TILDE

Neste capítulo, apresentaremos o HTILDE (*Hoeffding TILDE*), que é o sistema proposto nesta dissertação. Na seção 5.1, será feita uma introdução ao método. Já na seção 5.2, exibiremos o algoritmo. Finalizamos o capítulo com a seção 5.3, em que fazemos algumas considerações sobre como representar um problema no HTILDE e como utilizá-lo.

5.1 Introdução

O HTILDE é o algoritmo proposto neste trabalho. Ele é um sistema de programação em lógica indutiva escalável para grandes bases de dados. O HTILDE foi desenvolvido visando unir eficiência e expressividade: de um lado, a eficiência obtida, previamente por sistemas proposicionais, ao se utilizar o limite de Hoeffding para lidar com um número muito grande de exemplos; de outro, a expressividade de sistemas ILP, que utilizam lógica de primeira ordem e, por isso, são mais apropriados para extrair informações a partir de dados relacionais.

O HTILDE é baseado no TILDE [3] e no VFDT [11], que foram explicados nos capítulos 3 e 4. Escolhemos o TILDE para basear o nosso trabalho por ele ser um sistema ILP que realiza a prova de exemplos de maneira eficiente. Testar se uma cláusula cobre um exemplo é uma tarefa custosa em ILP, mas o TILDE conseguiu acelerar este processo utilizando o aprendizado a partir de interpretações [37] e os pacotes de cláusulas (*query packs*) [1].

Vários sistemas proposicionais usaram a metodologia proposta por Domingos e Hulten [13] para tornar algoritmos de aprendizado de máquinas escaláveis para grandes bases de dados. Dentre eles, escolhemos o VFDT [11] por ser uma árvore de decisão, assim como o TILDE.

O HTILDE é, portanto, um sistema que induz, incrementalmente, árvores de decisão lógicas de primeira ordem e utiliza o limite de Hoeffding para ser escalável para grandes bases de dados. Na seção 5.2, entraremos em mais detalhes a respeito do seu algoritmo.

5.2 Algoritmo

O pseudo-código do HTILDE é muito parecido com o do VFDT, mostrado no algoritmo 4.1. A diferença é que, em cada nó, ao invés de se avaliar qual é o melhor atributo que pode ser colocado como teste do mesmo, são considerados os possíveis refinamentos. Além disso, já foram considerados os valores τ e e_{min}, explicados na seção 4.4, assim como no algoritmo do VFDT exposto em [18].

Na tabela 5.1, ressaltamos as diferenças entre os algoritmos do VFDT e do HTILDE. Novamente, assim como nos capítulos 3 e 4, exibimos os pseudo-códigos em alto nível, demonstrando como ocorre a escolha de um atributo para ser colocado como teste de um nó interno nos dois algoritmos. Os passos que não estão exibidos do HTILDE (1 e 5) são iguais aos do VFDT. Já no algoritmo 5.1, detalhamos o pseudo-código do HTILDE, que utiliza de forma auxiliar a função *FazSplit*, mostrada no algoritmo 5.2.

Apesar do algoritmo do HTILDE ser semelhante ao do VFDT, vale ressaltar que ele foi implementado a partir do TILDE, uma vez que desejávamos aproveitar toda a parte relativa à lógica de primeira ordem, como a representação dos exemplos, a geração de refinamentos e os pacotes de cláusulas. A implementação do HTILDE foi, portanto, desenvolvida na linguagem utilizada pelo TILDE, que é um dialeto de Prolog misturado com objetos. Devido à utilização desta linguagem

Algoritmo 5.1 Algoritmo do HTILDE, para indução de uma árvore de decisão de lógica de primeira ordem escalável para grandes bases de dados.

Entradas:

S é uma seqüência de exemplos,

$\rho(.)$ é o operador de refinamento,

$\mathbf{Q}$ é um conjunto de refinamentos,

$G(.)$ é uma função de avaliação de *split*,

δ é um menos a probabilidade desejada de se escolher o refinamento correto em um dado nó.

τ é um valor definido pelo usuário para detectar empates de refinamentos.

e_{min} é um número que indica de quantos em quantos exemplos será verificado se a folha deve sofrer um *split*.

Saída:

HT é uma árvore de decisão de lógica de primeira ordem.

Procedimento HTILDE (S, G, δ, τ, e_{min})

1: Seja HT uma árvore com uma única folha l_1 (a raiz).
2: Seja $\mathbf{Q}_1 = \rho(Verdadeiro)$.
3: Para cada classe c
4: Para cada refinamento $Q_i \in \mathbf{Q}_1$
5: Faça $n[l_1][i][Verdadeiro][c] = 0$.
6: Faça $n[l_1][i][Falso][c] = 0$.
7: Para cada exemplo e em S
8: Desça e até uma folha l usando HT.
9: Faça Q_l ser a consulta associada de l.
10: Seja $\mathbf{Q}_l = \rho(Q_l)$.
11: Para cada refinamento $Q_i \in \mathbf{Q}_l$
12: Se $\leftarrow Q_i$ for verdadeira em $e \wedge B$, então
13: Incremente $n[l][i][Verdadeiro][classe(e)]$ em 1.
14: Senão
15: Incremente $n[l][i][Falso][classe(e)]$ em 1.
16: Faça E_l ser o número de exemplos que chegaram até agora na folha l.
17: Se os exemplos que chegaram até agora em l não forem todos da mesma classe e E_l módulo $e_{min} = 0$, então
18: Calcule $\overline{G_l}(Q_i)$ para cada refinamento $Q_i \in \mathbf{Q}_l$ usando os contadores n.
19: Faça Q_a ser o refinamento com o maior valor de $\overline{G_l}$.
20: Faça Q_b ser o refinamento com o segundo maior valor de $\overline{G_l}$.
21: Calcule ϵ usando a equação 4.1.
22: Faça $\Delta\overline{G_l} = \overline{G_l}(Q_a)$ - $\overline{G_l}(Q_b)$.
23: Se $\Delta\overline{G_l} > \epsilon$ ou $\Delta\overline{G_l} \leq \epsilon < \tau$, então
24: FazSplit(HT, l, Q_a, Q_l).
25: Retorne HT.

Algoritmo 5.2 Função *FazSplit*, chamada pelo HTILDE, cujo pseudo-código é exibido no algoritmo 5.1.

Entradas:

HT é a árvore lógica de primeira ordem que vai ser modificada,
l é a folha de HT que vai sofrer o *split*,
Q_a é o melhor refinamento para o nó l,
Q_l é a consulta associada do nó l.

Saída:

HT, que é a árvore de decisão de lógica de primeira ordem, tendo sofrido o *split* no nó l.

Procedimento FazSplit (HT, l, Q_a, Q_l)

1: Faça $conj = Q_a - Q_l$.
2: Substitua l por um nó interno cujo teste é a conjunção de literais $conj$.
3: Para cada ramo do *split*
4: Adicione uma nova folha l_m.
5: Se for o ramo esquerdo, então
6: Faça $\mathbf{Q}_m = \rho(Q_a)$.
7: Senão (ou seja, se for o ramo direito)
8: Faça $\mathbf{Q}_m = \rho(Q_l)$.
9: Para cada classe c
10: Para cada refinamento $Q_i \in \mathbf{Q}_m$
11: Faça $n[l_m][i][Verdadeiro][c] = 0$.
12: Faça $n[l_m][i][Falso][c] = 0$.
13: Retorne HT.

VFDT	HTILDE
1. Raiz: *Exs* = os N primeiros exemplos de treinamento, N dado pelo limite de Hoeffding ($\Delta\overline{G} > \epsilon$ ou $\Delta\overline{G} < \epsilon < \tau$).	
2. Baseado no conjunto de exemplos *Exs*, escolha o melhor atributo A para ser colocado como teste do nó.	⇒ 2. Baseado no conjunto de exemplos *Exs*, escolha o melhor refinamento R para ser colocado como teste do nó.
3. Crie um ramo para cada valor *vi* do atributo A, correspondente ao teste $A = vi$.	⇒ Crie dois ramos: um correspondente ao teste $R = Verdadeiro$ e o outro a $R = Falso$.
4. Descarte os exemplos já considerados para o *split*. Divida os próximos exemplos entre os nós filhos, de acordo com o valor do atributo A, gerando os conjuntos de exemplos *Exs* de cada filho. O número de exemplos de cada filho também será determinado pelo limite de Hoeffding.	⇒ Descarte os exemplos já considerados para o *split*. Divida os próximos exemplos entre os nós filhos, de acordo com o valor do refinamento R, gerando os conjuntos de exemplos *Exs* de cada filho. O número de exemplos de cada filho também será determinado pelo limite de Hoeffding.
5. Repita 2 para cada filho.	

Tabela 5.1: Diferenças entre os algoritmos em alto nível do VFDT e do HTILDE.

própria do TILDE, a etapa de implementação do HTILDE foi a fase mais difícil e demorada do nosso trabalho.

Finalizamos a seção exibindo o algoritmo 5.3, em que mostramos o pseudo-código utilizado para classificar um exemplo usando o HTILDE. Ele é muito parecido com o algoritmo 3.1, que classifica um exemplo com o TILDE. A diferença agora é que pode ocorrer o caso em que uma folha não tenha exemplos, ou seja, ela foi criada por um *split* do nó pai, mas depois nenhum outro exemplo de treinamento chegou nela. Para contornar estes casos e podermos classificar um exemplo que venha a cair neste tipo de folha, estabelecemos como classe padrão de uma folha a classe que o seu nó pai tinha antes de sofrer o *split*. Este procedimento é semelhante ao utilizado pelo VFDT nesses casos.

Algoritmo 5.3 Algoritmo de classificação de um exemplo usando o HTILDE (com um conhecimento preliminar B)

Entrada:

e é um exemplo.

Saída:

c é uma classe.

Procedimento Classifica (e)

1: Seja Q a consulta Prolog $Verdadeiro$.
2: Seja N o nó raiz da árvore.
3: Enquanto N não for uma folha
4: Seja N = **noInterno**($conj$, esq, dir).
5: Se $Q \wedge conj$ for verdadeira em $e \wedge B$, então
6: Faça $Q = Q \wedge conj$.
7: Faça $N = esq$.
8: Senão
9: Faça $N = dir$.
10: Faça n = número de exemplos de treinamento que caíram na folha N.
11: Se $n = 0$, então
12: Faça c = a classe do nó pai de N, antes de sofrer o *split*.
13: Senão
14: Faça c = a classe mais freqüente entre os exemplos de treinamento que caíram nó N.
15: Retorne c.

5.3 Especificação de um problema

Antes de concluirmos o capítulo, é interessante fazer um comentário sobre como um problema é representado no HTILDE. Como ele foi desenvolvido no mesmo ambiente do TILDE, um problema é especificado de maneira muito semelhante nos dois sistemas: através dos arquivos *.kb*, *.bg* e *.s*, que foram explicados no capítulo 3.

Para utilizar o HTILDE, algumas pequenas mudanças devem ser feita no arquivo *.s*. O usuário deve incluir três linhas específicas para definir os parâmetros utilizados pelo algoritmo: *htilde_delta*, *htilde_emin* e *htilde_tau*. Estes comandos definem, respectivamente, os parâmetros δ, e_{min} e τ do algoritmo. Para começar a executá-lo, deve-se utilizar o comando *execute(induce(htilde))*. Finalmente, vale ressaltar que utilizamos a heurística padrão do VFDT, que é o ganho de informação. Como esta não é a medida padrão do TILDE, que é a taxa de ganho de informação, devemos colocar mais um comando, *heuristic(gain)*, para definir que queremos usar o ganho de informação como heurística. Um exemplo destes comandos podem ser vistos a seguir:

```
htilde_delta(0.000001).
htilde_emin(300).
htilde_tau(0.05).
heuristic(gain).
execute(induce(htilde)).
```

Além das alterações no arquivo *.s*, pode ser necessário incluir novas linhas no arquivo *.bg* para informar quais predicados são simétricos. Esta questão será melhor discutida a seguir.

5.3.1 Predicados simétricos

Se linguagem definida no arquivo *.s* possuir predicados simétricos, pode ocorrer do operador de refinamento do TILDE criar refinamentos que não sejam iguais sintaticamente, mas que tenham o mesmo significado e provem exatamente os mesmos exemplos. Alguns exemplos de refinamentos deste tipo são os seguintes:

nomePessoa(IdA, NA), nomePessoa(IdB, NB), amigo(NA, NB).
nomePessoa(IdA, NA), nomePessoa(IdB, NB), amigo(NB, NA).

Os dois refinamentos exibidos diferem na ordem dos termos do predicado *amigo*. Entretanto, *amigo* é uma relação simétrica: se X é amigo de Y, então Y é amigo de X. Sendo assim, $amigo(NA, NB)$ e $amigo(NB, NA)$ querem dizer a mesma coisa e provam os mesmos exemplos. Devido a isso, os dois refinamentos têm o mesmo ganho de informação.

Este tipo de situação, se não for tratada, pode causar problemas para o funcionamento do HTILDE. Suponha que, em uma folha l, os dois melhores refinamentos (no algoritmo, Q_a e Q_b) sejam os exibidos anteriormente. Como ambos provam os mesmos exemplos, a diferença entre os seus ganhos será sempre zero ($\Delta G_l = 0$) e, portanto, haverá um empate. Se o parâmetro τ for zero, esta folha nunca será transformada em um nó interno, uma vez que ϵ é um número positivo e, portanto, nunca vai ocorrer $\Delta G_l > \epsilon$. Se o usuário definir um valor positivo para τ, esta folha poderá ser transformada em um nó interno, devido à condição $\Delta G_l \leq \epsilon < \tau$. Neste caso, um dos dois refinamentos será escolhido como teste para o nó. Note que, para ocorrer o *split* por esta condição, ϵ deve diminuir muito a ponto de ser menor do que τ. Vale ressaltar que este parâmetro tem tipicamente um valor baixo (por exemplo, o valor padrão para τ no VFDT é 0.05). Como ϵ, dado um problema, é uma função exclusivamente de n (já que, uma vez estabelecido o problema, R e δ não variam), para ϵ diminuir a ponto de ser menor do que τ, a folha deve ter acumulado muitos exemplos.

Suponha que para esta folha, o terceiro melhor refinamento, que chamaremos de Q_c, fosse realmente diferente dos dois primeiros e tivesse ganho menor do que Q_a. Se detectássemos que, devido aos predicados simétricos, os dois primeiros refinamentos são "iguais", poderíamos desconsiderar Q_b e calcular a diferença de ganhos entre Q_a e Q_c. Desta forma, ΔG_l não seria para sempre 0 e poderia vir a ocorrer o *split* pelo critério $\Delta G_l > \epsilon$, sendo necessário acumular bem menos exemplos na folha.

O TILDE não trata o caso de refinamentos que provam os mesmos exemplos devido ao uso de predicados simétricos. Como no TILDE avaliam-se todos os exemplos que caem na folha e então escolhe-se apenas o melhor refinamento para se colocar como teste do novo nó interno, o problema que pode ocorrer devido aos predicados simétricos é existirem mais refinamentos a serem avaliados em cada nó. O TILDE pode ficar mais lento, mas a árvore gerada será a mesma.

Já para o HTILDE, como ele precisa comparar os dois melhores refinamentos para então decidir se vai ou não transformar a folha em nó interno, pode ocorrer diferença na árvore criada. Uma folha pode precisar acumular muitos exemplos para sofrer um *split*, como foi explicado anteriormente, e no caso extremo, isso pode nem vir a ocorrer (por exemplo, se τ for zero).

Por isso, implementamos uma função que verifica se dois refinamentos são "iguais", dados os predicados simétricos, e garantimos que isto não ocorre entre os dois melhores refinamentos Q_a e Q_b. Para esta função funcionar corretamente, o usuário deve colocar no arquivo *.bg* uma linha *symmetric*(P) para cada predicado de nome P que seja simétrico. Exemplos destas linhas podem ser vistos a seguir:

```
symmetric(amigo).
symmetric(igual).
```

Vale ressaltar que *amigo* é uma relação simétrica: se X é amigo de Y, então Y é amigo de X. Outra relação que também é simétrica é *igual*: se X é igual a Y, então Y é igual a X.

Capítulo 6

Resultados experimentais

Neste capítulo, mostraremos os experimentos realizados com o HTILDE. Utilizamos duas bases de dados, o Bongard e o Cora, e comparamos os resultados com os obtidos pelo TILDE. Os resultados dos experimentos que fizemos com o Bongard serão exibidos na seção 6.1; já os dos que realizamos com o Cora serão apresentados na seção 6.2.

Em todos os testes, utilizamos como parâmetros do HTILDE $\delta = 10^{-6}$, $tau = 0.05$ e $e_{min} = 300$. Estes valores foram escolhidos por serem os valores dos mesmos parâmetros utilizados como padrão pelo sistema VFDT.

Os dois algoritmos foram executados usando validação cruzada [19]. As medidas de avaliação e o tempo de aprendizado que são exibidos neste capítulo são as médias dos valores obtidos em cada *fold*. Para comparar se a diferença entre os desempenhos dos dois algoritmos foi estatisticamente significativa, utilizamos o *teste-t corrigido* com 95% de confiança [30].

6.1 Bongard

A base de dados Bongard já foi brevemente explicada no capítulo 3. Ela foi introduzida em [4] e é uma base relacional *benchmark*, cujo objetivo é classificar figuras em *positivas* ou *negativas* de acordo com os objetos que as constituem e das relações entre estes objetos (por exemplo, um objeto pode estar dentro de outro).

Utilizamos um gerador de exemplos, fornecido pelos criadores do TILDE, para produzir duas versões desta base de dados, uma com 500 mil exemplos e outra com 1 milhão. Apresentaremos os resultados da base com 500 mil exemplos na sub-seção 6.1.1; já os resultados da base com 1 milhão de exemplos serão exibidos na sub-seção 6.1.2.

A versão do problema Bongard que foi utilizada nos experimentos difere um pouco da apresentada no capítulo 3, por utilizar alguns predicados a mais, permitindo que seja especificada a posição de um objeto em relação a outro (por exemplo, se um está ao norte, ao sul, ao leste ou ao oeste do outro). Além disso, o gerador introduziu um ruído de 0.1%, ou seja, a cada mil exemplos criados, um era classificado de forma errada pelo gerador, de forma a diminuir a artificialidade da base dados e dificultar o aprendizado. Este ruído foi utilizado por ser o valor utilizado como padrão pelo gerador de exemplos.

O gerador produziu bases de dados desbalanceadas, em que 14% dos exemplos eram positivos e 86% eram negativos, tanto na versão com 500 mil quanto na versão com 1 milhão de exemplos. Devido a isso, utilizamos a medida F para avaliar o desempenho dos algoritmos. Usamos a validação cruzada *k-fold* com $k = 10$ e a proporção de exemplos de cada classe foi mantida, ou seja, foi a mesma tanto no conjunto original de exemplos quanto em cada um dos *folds*.

É importante destacar que, para a execução de ambos os algoritmos, utilizamos computadores *dual core* com 2 Gb de memória RAM.

6.1.1 Bongard com 500 mil exemplos

Nesta sub-seção, apresentaremos os resultados obtidos pelos experimentos que usaram a base de dados Bongard com 500 mil exemplos.

Como utilizamos validação cruzada *k-fold* com $k = 10$, cada *fold* usou 50 mil exemplos para teste. Geramos as curvas de aprendizado de cada sistema, que são exibidas na figura 6.1. Elas mostram as medidas F obtidas pelos algoritmos ao serem treinados por todo o conjunto de treinamento do *fold*, que tem 450 mil

exemplos, e por dois subconjuntos do mesmo: um com 150 mil e outro com 300 mil exemplos. Explicitamos os valores das medidas F exibidos na curva de aprendizado na tabela 6.1.

Observando a curva de aprendizado da figura 6.1 e a tabela 6.1 podemos notar que, para o sistema TILDE, a medida F é praticamente constante para todos os conjuntos de treinamento. Entretanto, para o HTILDE, ela vai subindo, conforme o número de exemplos cresce. Utilizamos o teste-t corrigido com 95% de confiança [30] para comparar as medidas F, avaliadas nos conjuntos de teste, obtidas pelos dois algoritmos para cada conjunto de treinamento. Constatamos que a diferença foi significativa, ou seja, o TILDE gerou um modelo melhor do que o HTILDE, para todos os conjuntos de treinamento considerados. Entretando, percebemos que o fato das medidas F aumentarem conforme o número de exemplos cresce está de acordo com o esperado para o HTILDE, uma vez que a propriedade de árvores de Hoeffding, explicada no capítulo 4, diz que o algoritmo produz árvores assintoticamente próximas às produzidas por um algoritmo *batch*. Em outras palavras, o HTILDE é adequado para aprender a partir de bases muito grandes e é de se esperar que, conforme o número de exemplos aumente, a qualidade da árvore gerada por este sistema melhore e se aproxime da qualidade do modelo criado pelo TILDE. Para estudarmos melhor esta questão, geramos um conjunto ainda maior de exemplos. Falaremos sobre os resultados desta base maior na próxima sub-seção.

	Treinamento		**Teste**	
	TILDE	**HTILDE**	**TILDE**	**HTILDE**
150000	91,8	83,3	91,3	83,2
300000	93,0	84,8	92,3	84,7
450000	92,5	87,3	91,9	87,3

Tabela 6.1: Medidas F (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 500 mil exemplos, a partir de 150000, 300000 e 450000 exemplos de treinamento avaliadas no conjunto de treinamento e no de teste.

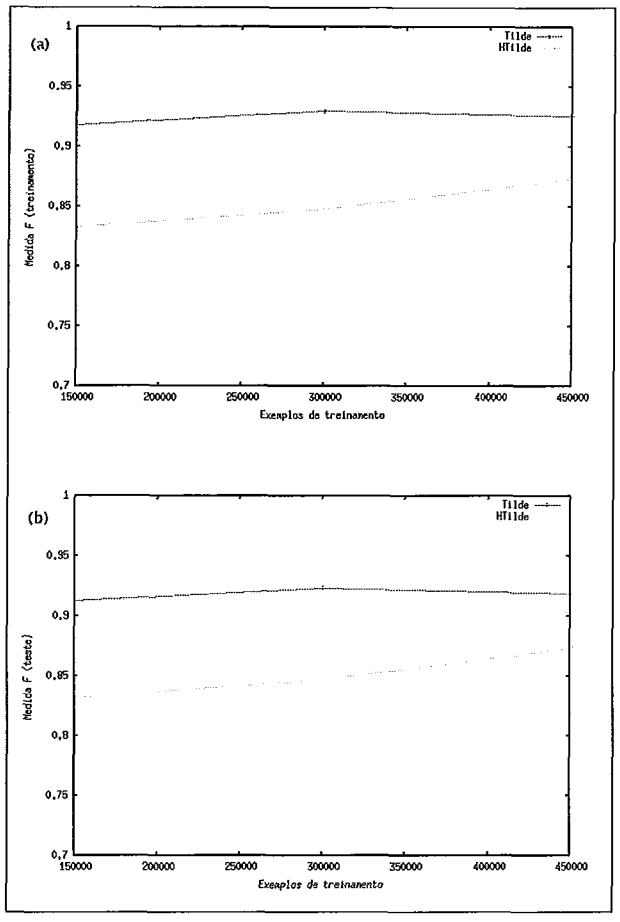


Figura 6.1: Curvas de aprendizado para a base de dados Bongard com 500 mil exemplos. São exibidas as medidas F obtidas pelo TILDE e pelo HTILDE avaliadas no conjunto de treinamento (a) e no de teste (b).

Nas tabelas 6.2, 6.3 e 6.4, exibimos outras medidas obtidas pelos algoritmos. Na tabela 6.2 mostramos a precisão, na tabela 6.3 apresentamos a revocação, já na tabela 6.4 exibimos a acurácia. É importante destacar que apresentamos estes valores mais por curiosidade, uma vez que a medida F é uma função da precisão e da revocação, e a acurácia, apesar de ser uma das medida mais usadas, não é muito apropriada para o problema já que a base de dados é desbalanceada.

Na tabela 6.5, exibimos o tempo de indução de cada algoritmo (ou seja, quanto tempo cada um deles levou para construir o modelo) e o número de regras geradas para cada um dos conjuntos de treinamento que foram indicados na curva de aprendizado. Os tempos foram medidos em segundos. Notamos que o HTILDE

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
150000	99,0	93,0	98,6	92,9
300000	98,8	93,2	98,1	93,2
450000	98,9	94,6	98,3	94,6

Tabela 6.2: Precisão (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 500 mil exemplos, avaliadas no conjunto de treinamento e no de teste.

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
150000	85,6	75,6	85,0	75,5
300000	87,8	78,0	87,1	77,9
450000	86,9	81,2	86,2	81,2

Tabela 6.3: Revocação (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 500 mil exemplos, avaliadas no conjunto de treinamento e no de teste.

foi mais de quatro vezes mais rápido que o TILDE, em todos os conjuntos de exemplos. Além disso, o HTILDE aprendeu teorias menos complexas do que o TILDE, com cerca de quatro vezes menos regras.

Ao observarmos que a medida F do modelo aprendido pelo HTILDE para o Bongard com 500 mil exemplos foi menor do que a do TILDE, geramos uma base de dados Bongard com 1 milhão de exemplos para estudar se, de fato, o sistema proposto por esta dissertação melhora o seu desempenho com um número ainda maior de exemplos. Os resultados obtidos para esta nova versão da base Bongard são exibidos na sub-seção 6.1.2 a seguir.

6.1.2 Bongard com 1 milhão de exemplos

Nesta sub-seção, apresentaremos os resultados obtidos pelos experimentos que usaram a base de dados Bongard com 1 milhão de exemplos.

Como usamos validação cruzada *k-fold* com $k = 10$, cada *fold* utilizou 100 mil exemplos para teste. Geramos as curvas de aprendizado de cada sistema, que

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
150000	97,9	95,9	97,8	95,8
300000	98,2	96,2	98,0	96,1
450000	98,1	96,8	97,9	96,8

Tabela 6.4: Acurácia (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 500 mil exemplos, avaliadas no conjunto de treinamento e no de teste.

	Tempo (s)		Número de regras	
	TILDE	HTILDE	TILDE	HTILDE
150000	1839	478	125,6	37,5
300000	4681	1089	290,7	66,6
450000	7908	1590	393,8	97,4

Tabela 6.5: Tempo de indução, medido em segundos, e o número de regras obtidas pelos modelos aprendidos pelo TILDE e pelo HTILDE para a base Bongard com 500 mil exemplos.

são exibidas na figura 6.2. Elas mostram as medidas F obtidas pelos algoritmos ao serem treinados por todo o conjunto de treinamento do *fold*, que tem 900 mil exemplos, e por vários subconjuntos do mesmo. Na tabela 6.6, explicitamos os valores das medidas F exibidos na curva de aprendizado. É importante ressaltar que geramos modelos a partir de conjuntos de treinamento de diversos tamanhos, mas o interesse é avaliar o desempenho dos algoritmos quando existe um grande número de exemplos. Portanto, focaremos a análise dos resultados dos conjuntos que possuem 100000 exemplos de treinamento ou mais. No final da sub-seção, faremos algumas observações sobre os resultados obtidos com os conjuntos menores.

Observando a curva de aprendizado da figura 6.2 e a tabela 6.6 podemos notar que, a partir de 100000 exemplos de treinamento, a medida F obtida pelo sistema TILDE é praticamente constante para todos os conjuntos de treinamento. Mais precisamente, com 100000 exemplos, a medida F obtida pelo TILDE alcançou o seu valor de máximo e, para os conjuntos maiores, ela diminuiu um pouco, sugerindo a ocorrência de *overfitting*. Entretanto, para o HTILDE, ela vai subindo,

conforme o número de exemplos cresce. Utilizamos o teste-t corrigido com 95% de confiança [30] para comparar as medidas F, avaliadas nos conjuntos de teste, obtidas pelos dois algoritmos para cada conjunto de treinamento. Constatamos que nos dois maiores conjuntos, a diferença entre as medidas F não foi significativa, ou seja, o HTILDE gerou modelos tão bons quanto os aprendidos pelo TILDE. Este resultado está de acordo com o esperado, conforme dito anteriormente, uma vez que o HTILDE melhora a qualidade do modelo aprendido à medida que o número de exemplos de treinamento aumenta, assim como o VFDT.

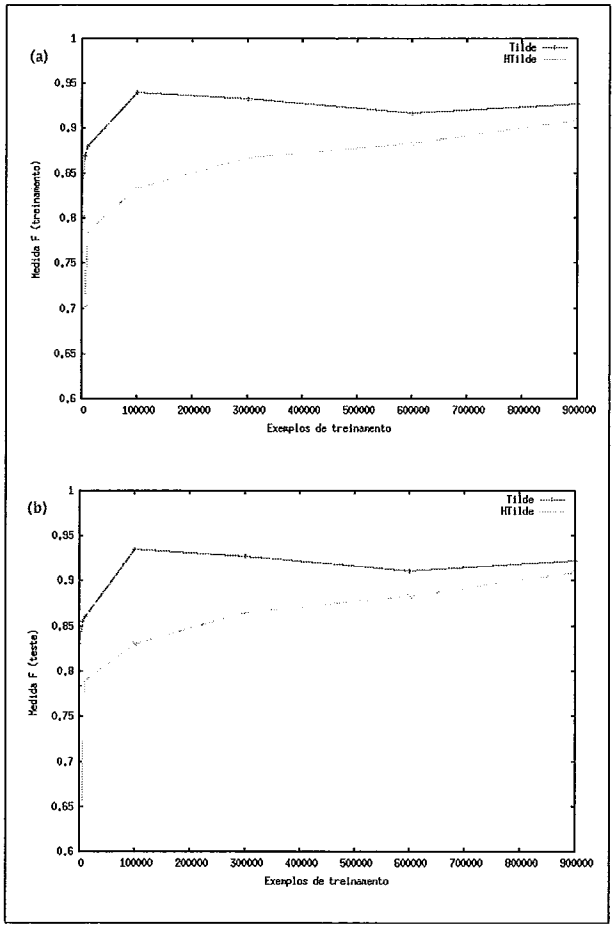


Figura 6.2: Curvas de aprendizado para a base de dados Bongard com 1 milhão de exemplos. São exibidas as medidas F obtidas pelo TILDE e pelo HTILDE avaliadas no conjunto de treinamento (a) e no de teste (b).

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
100	0.0	0.0	0.0	0.0
500	80,3	0.0	78,4	0.0
1000	83,7	0.0	83,1	0.0
5000	87,0	70,2	85,6	71,1
10000	88,1	78,6	86,1	79,0
100000	94,0	83,4	93,5	83,1
300000	93,3	86,8	92,7	86,6
600000	91,7	88,4	91,1	88,4
900000	92,7	90,9	92,2	90,9

Tabela 6.6: Medidas F (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 1 milhão de exemplos, a partir de 100, 500, ..., 900000 exemplos de treinamento avaliadas no conjunto de treinamento e no de teste.

Assim como fizemos para a base Bongard com 500 mil exemplos, nas tabelas 6.7, 6.8 e 6.9, mostramos outras medidas obtidas pelos algoritmos. Na tabela 6.7 exibimos a precisão, na tabela 6.8 apresentamos a revocação, já na tabela 6.9 exibimos a acurácia. Destacamos novamente que exibimos estes valores mais por curiosidade, já que a medida F é uma função da precisão e da revocação e a acurácia não é muito apropriada para o problema, uma vez que a base é desbalanceada.

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
100	0.0	0.0	0.0	0.0
500	82,4	0.0	80,3	0.0
1000	92,0	0.0	92,5	0.0
5000	94,1	67,9	92,5	69,7
10000	95,0	76,8	93,3	77,3
100000	99,8	93,3	99,6	93,3
300000	99,2	94,6	98,7	94,7
600000	99,0	94,6	98,5	94,6
900000	99,0	98,3	98,5	98,3

Tabela 6.7: Precisão (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 1 milhão de exemplos, avaliadas no conjunto de treinamento e no de teste.

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
100	0.0	0.0	0.0	0.0
500	78,3	0.0	76,7	0.0
1000	76,8	0.0	75,4	0.0
5000	80,9	72,7	79,6	72,7
10000	82,2	80,4	80,0	80,7
100000	88,9	75,5	88,1	75,1
300000	88,0	80,2	87,3	79,9
600000	85,3	83,0	84,8	82,9
900000	87,1	84,5	86,7	84,6

Tabela 6.8: Revocação (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 1 milhão de exemplos, avaliadas no conjunto de treinamento e no de teste.

Na tabela 6.10, mostramos o tempo de indução de cada algoritmo, medido em segundos, e o número de regras geradas para cada um dos conjuntos de treinamento que foram usados na curva de aprendizado. Considerando-se os conjuntos de treinamento com 100000 exemplos ou mais, o HTILDE foi mais de quatro vezes mais rápido do que o TILDE. Além disso, ele aprendeu teorias menos complexas do que o TILDE, com menos da metade das regras geradas pelo TILDE para o conjunto com 100000 exemplos e com cerca de quatro vezes menos regras para os três maiores conjuntos.

Finalizamos a sub-seção fazendo algumas considerações a respeito dos modelos aprendidos a partir dos conjuntos com menos de 100000 exemplos de treinamento. Analisando a tabela 6.10, verificamos que com 100 exemplos de treinamento, ambos os algoritmos geram apenas 1 regra, que classifica todos os exemplos como negativos, o que faz com que a medida F, a precisão e a revocação sejam 0% (tabelas 6.6, 6.7 e 6.8), mas a acurácia seja 86% (tabela 6.9), uma vez que a base é desbalanceada. Com 500 exemplos de treinamento, o modelo gerado pelo TILDE já não classifica todos os exemplos como negativos, mas isso só deixa de ocorrer com o HTILDE quando são considerados 5000 exemplos de treinamento. Vale lembrar que, como $e_{min} = 300$, no conjunto com 1000 exemplos, por exemplo,

	Treinamento		Teste	
	TILDE	HTILDE	TILDE	HTILDE
100	86,0	86,0	86,2	86,2
500	94,7	86,2	94,2	86,2
1000	95,9	86,2	95,8	86,2
5000	96,7	93,0	96,3	93,3
10000	96,9	94,0	96,4	94,1
100000	98,4	95,9	98,3	95,8
300000	98,3	96,6	98,1	96,6
600000	97,9	97,0	97,7	97,0
900000	98,1	97,7	98,0	97,7

Tabela 6.9: Acurácia (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE, para a base Bongard com 1 milhão de exemplos, avaliadas no conjunto de treinamento e no de teste.

o HTILDE só havia verificado três vezes se era necessário transformar a raiz em um nó interno. O desempenho dos dois algoritmos vai melhorando, mas nesses conjuntos (com menos de 100000 exemplos), observamos que o HTILDE é pior do que o TILDE, de acordo com o esperado, uma vez que o HTILDE foi desenvolvido para aprender modelos a partir de um número muito grande de exemplos.

6.2 Cora

A base de dados Cora foi apresentada por McCallum em [25]. Ela contém dados sobre vários trabalhos científicos, como o título, o nome do autor, onde ele foi publicado e em que ano. Kok e Domingos adicionaram novos predicados à base original, que medem a porcentagem de palavras que dois campos têm em comum [20]. O objetivo é identificar referências repetidas, ou seja, quando duas referências dizem respeito a uma mesma publicação.

O Cora é uma base de dados bastante desbalanceada, em que apenas 4% dos exemplos são positivos e 96% são negativos. Por isso, assim como no caso dos experimentos com o Bongard apresentados na seção 6.1, utilizamos a medida F para avaliar os desempenhos dos algoritmos.

	Tempo (s)		Número de regras	
	TILDE	HTILDE	TILDE	HTILDE
100	56	71	1,0	1,0
500	56	72	6,7	1,0
1000	55	71	9,0	1,0
5000	67	77	20,1	3,9
10000	87	87	28,9	5,0
100000	1357	334	66,1	25,7
300000	4457	1015	279,3	66,5
600000	10500	2234	470,4	126,2
900000	18221	3944	678,2	178,3

Tabela 6.10: Tempo de indução, medido em segundos, e o número de regras obtidas pelos modelos aprendidos pelo TILDE e pelo HTILDE para a base Bongard com 1 milhão de exemplos.

Nesta base de dados, usamos validação cruzada 5x2 [10]. Este foi o tipo de validação cruzada utilizada porque já obtivemos o Cora separado nestes *folds*. Vale destacar que cada *fold* contém mais de 400 mil exemplos de treinamento.

Pudemos perceber que o Cora é uma base de dados difícil e o aprendizado depende muito do espaço de busca considerado e das restrições a ele impostas. Dizemos isso porque, inicialmente, obtivemos esta base de dados no formato do Aleph [38], que é outro sistema ILP, e convertemos os exemplos para o formato aceito pelo TILDE. Ao executarmos o TILDE com o Cora assim obtido, usando apenas os modos e tipos de variáveis que existiam na base no formato original, o sistema classificava todos os exemplos como negativos. Devido a isso, acreditamos ser necessário considerar *lookaheads* para esta base.

Conforme foi explicado no capítulo 3, para se utilizar *lookaheads* no TILDE, é necessário definir quais as combinações de literais que podem ser consideradas, além da profundidade do *lookahead*, que é dada pelo comando *max_lookahead*. Este parâmetro indica se devem ser considerados só os refinamentos da cláusula associada (para o valor 0), ou os refinamentos dos refinamentos (1), e assim por diante. Inicialmente, como não tínhamos nenhum conhecimento prévio a respeito do problema, consideramos todas as combinações de literais possíveis e utilizamos

max_lookahead = 1. Como a medida F assim obtida foi muito baixa (19%), aumentamos o *max_lookahead* para 2, como forma de aumentar o espaço de busca, mas ele aumentou muito, o que fez os experimentos pararem por falta de memória.

Ao nos depararmos com esta dificuldade, pedimos ajuda aos criadores do TILDE, que então nos enviaram um arquivo de configurações para o Cora, onde é definida a linguagem considerada e, portanto, o espaço de busca. Verificamos que este arquivo direciona bem o espaço de busca, isto é, várias restrições são impostas a ele e são consideradas apenas algumas combinações de literais para o *lookahead*. Devido a esta diminuição do espaço de busca, foi possível executar os algoritmos com *max_lookahead* = 2. Na tabela 6.11, mostramos a medida F, a precisão, a revocação e a acurácia assim obtidas pelos algoritmos TILDE e HTILDE. Já na tabela 6.12, exibimos o tempo de aprendizado do modelo, que foi medido em segundos, e o número de regras geradas por cada sistema. É importante destacar que, para a execução dos algoritmos, utilizamos computadores *dual core* com 1 Gb de memória RAM.

	Treinamento		**Teste**	
	TILDE	**HTILDE**	**TILDE**	**HTILDE**
Medida F	49,5	43,5	41,9	40,9
Precisão	73,4	68,8	49,2	56,0
Revocação	42,8	37,7	40,8	37,8
Acurácia	97,5	97,3	96,7	97,0

Tabela 6.11: Medidas F, precisão, revocação e acurácia (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE para a base Cora avaliadas no conjunto de treinamento e no de teste.

	TILDE	**HTILDE**
Tempo (s)	152	157
Número de regras	46,7	68,0

Tabela 6.12: Tempo de indução, medido em segundos, e o número de regras obtidas pelos modelos aprendidos pelo TILDE e pelo HTILDE para a base Cora.

Observando a tabela 6.11, podemos perceber que as acurácias obtidas pelos algoritmos têm valores bem altos. Entretanto, pelas medidas F obtidas, que foram bem baixas, é possível notar que os dois sistemas tiveram dificuldade para classificar os exemplos positivos.

Usamos o teste-t corrigido com 95% de confiança [30] e verificamos que a diferença entre as medidas F obtidas pelo TILDE e pelo HTILDE, avaliadas no conjunto de teste, não foram significativas.

Analisando a tabela 6.12, percebemos que os tempos de indução de ambos os algoritmos foram bem próximos, mas muito baixos considerando-se o tamanho da base. Note que os sistemas levaram menos de três minutos para aprender as teorias. Entretanto, apesar da rapidez, as qualidades dos modelos não foram boas, como podemos observar pelas medidas F obtidas (tabela 6.11).

Com o objetivo de melhorar os modelos aprendidos, tentamos variar alguns parâmetros do arquivo de configurações da base de dados, de forma a aumentar o espaço de busca considerado. Nas tabelas 6.13 e 6.14, exibimos os resultados obtidos quando aumentamos o parâmetro *max_lookahead* de dois para quatro. Percebemos que o HTILDE levou menos tempo (tabela 6.14) do que o TILDE para aprender o modelo com este espaço de busca maior, mas infelizmente, verificamos que esta tentativa de nada adiantou porque não alterou a qualidade as teorias geradas, como pode ser visto pelas medidas F obtidas, exibidas na tabela 6.13.

	Treinamento		**Teste**	
	TILDE	**HTILDE**	**TILDE**	**HTILDE**
Medida F	49,5	43,5	41,9	40,9
Precisão	73,4	68,5	49,2	56,0
Revocação	42,8	37,8	40,8	37,8
Acurácia	97,5	97,3	96,7	97,0

Tabela 6.13: Medidas F, precisão, revocação e acurácia (em %) dos modelos aprendidos pelo TILDE e pelo HTILDE para a base Cora com *max_lookahead* = 4 avaliadas no conjunto de treinamento e no de teste.

	TILDE	HTILDE
Tempo (s)	1709	601
Número de regras	46,7	67,7

Tabela 6.14: Tempo de indução, medido em segundos, e o número de regras obtidas pelos modelos aprendidos pelo TILDE e pelo HTILDE para a base Cora com *max_lookahead* = 4.

As medidas F que apresentamos nas tabelas 6.11 e 6.13 foram as mais altas que conseguimos alcançar até agora. Com o objetivo de melhorar as teorias geradas pelos sistemas, tentamos variar mais parâmetros do arquivo de configurações que controlam o espaço de busca. Entretanto, até agora não obtivemos resultados melhores do que os exibidos nas tabelas 6.11 e 6.13. Além disso, nos outros experimentos que realizamos, em que retirarmos algumas das restrições do espaço de busca, ou o aprendizado foi ficando extremamente lento sem conseguir melhorar a teoria, ou novamente tivemos problemas dos experimentos pararem por falta de memória.

É importante destacar que Kok e Domingos em [20] também impuseram algumas restrições ao espaço de busca considerado nos seus experimentos. Concluímos então que o Cora é realmente um problema difícil e precisamos impor restrições ao espaço de busca, de forma a ele ser tratável, mas temos que tentar fazer isso de forma a não prejudicar muito a qualidade das teorias aprendidas. Além disso, Kok e Domingos argumentaram que sistemas estritamente ILP têm dificuldades de aprender modelos para a base de dados Cora. Como o TILDE e o HTILDE são sistemas deste tipo, acreditamos que esta possa ser uma das dificuldades.

Apesar disso, ainda estamos realizando experimentos, com o objetivo de tentar melhorar a qualidade dos modelos gerados para a base de dados Cora.

Capítulo 7

Conclusão

Neste trabalho, foi desenvolvido e implementado o HTILDE, um algoritmo de programação em lógica indutiva escalável para grandes bases de dados relacionais. Ele é baseado em dois outros sistemas: o VFDT [11], que é uma árvore de decisão proposicional que utiliza o limite de Hoeffding para amostrar exemplos e lidar com grandes bases de dados, e o TILDE [3], que é uma árvore de decisão em lógica de primeira ordem que realiza a prova de exemplos de uma maneira otimizada.

Realizamos experimentos com duas bases de dados, uma artificial e uma real. Os resultados mostraram que, na base artificial, houve uma melhora significativa de eficiência (o HTILDE foi mais do que quatro vezes mais rápido do que o TILDE), sem que a qualidade do modelo fosse prejudicada. Além disso, o HTILDE aprendeu teorias mais compactas, com cerca de quatro vezes menos regras do que o TILDE. Já para a base real, os resultados foram inconclusivos, uma vez que os dois sistemas geraram modelos muito rapidamente. As teorias obtidas não foram muito boas e, por isso, ainda estamos realizando experimentos na tentativa de melhorá-las.

Além disso, os resultados obtidos para a base de dados artificial confirmam que o HTILDE melhora o desempenho do modelo aprendido e o torna cada vez próximo ao que é gerado pelo TILDE à medida que o número de exemplos de treinamento aumenta.

7.1 Trabalhos futuros

Nesta seção, falaremos brevemente sobre alguns trabalhos futuros que pretendemos fazer com o HTILDE. Apresentamos, a seguir, pontos que temos interesse em estudar e alguns experimentos que desejamos fazer.

Um primeiro trabalho que já estamos fazendo é realizar outros experimentos com a base de dados Cora, com o objetivo de tentar melhorar as teorias obtidas.

Desejamos também realizar experimentos com mais bases de dados, mas acreditamos que isso possa vir a ser uma tarefa complicada. Uma dificuldade que já enfrentamos no decorrer do nosso trabalho foi obter grandes bases de dados relacionais que pudéssemos utilizar nos nossos experimentos. Apesar de hoje em dia, como mencionamos anteriormente, várias organizações possuírem bases de dados com milhões de exemplos, elas não estão disponíveis para o público em geral e também não são cedidas para testes.

Outro trabalho que pretendemos fazer no futuro é realizar experimentos com diferentes valores para os parâmetros para o HTILDE (δ, τ e e_{min}) e não apenas com os definidos como padrão pelo VFDT. Desejamos investigar como outras configurações desses parâmetros podem melhorar os resultados obtidos pelo sistema.

Finalmente, também como trabalho futuro, desejamos comparar o HTILDE com outras abordagens. Desejamos realizar experimentos com o TILDE, aplicado a uma grande base de dados, usando outras técnicas de amostragem de exemplos. Como uma primeira opção, pretendemos aplicar a amostragem progressiva geométrica [31] no TILDE e comparar o desempenho desta abordagem com o obtido pelo HTILDE. Também pretendemos comparar o HTILDE com o sistema VFILPh [7] [6].

Bibliografia

[1] BLOCKEEL, H., DEHASPE, L., DEMOEN, B., *et al.*, "Improving the efficiency of inductive logic programming through the use of query packs", *Journal of Artificial Intelligence Research*, v. 16, pp. 135–166, 2002.

[2] BLOCKEEL, H., RAEDT, L. D., "Top-Down Induction of First-Order Logical Decision Trees", *Artificial Intelligence*, v. 101, n. 1-2, pp. 285–297, 1998.

[3] BLOCKEEL, H., RAEDT, L. D., JACOBS, N., *et al.*, "Scaling Up Inductive Logic Programming by Learning from Interpretations", *Data Mining and Knowledge Discovery*, v. 3, n. 1, pp. 59–93, 1999.

[4] BONGARD, M. M., *Pattern Recognition*. Spartan Books, 1970.

[5] BRATKO, I., MUGGLETON, S., "Applications of inductive logic programming", *Communications of the ACM*, v. 38, n. 11, pp. 65–70, 1995.

[6] CARDOSO, P. M., ZAVERUCHA, G., "Comparative evaluation of approaches to scale up ilp", In: *Short Papers of the 16th International Conference on Inductive Logic Programming (ILP 2006)*, pp. 37–39, UDC Press, 2006.

[7] CARDOSO, P. M., ZAVERUCHA, G., "Tornando a programação em lógica indutiva escalável (ilp) a bases de dados arbitrariamente grandes". Dissertação de Mestrado, COPPE, Universidade Federal do Rio de Janeiro, 2006.

[8] CASANOVA, M. A., GIORNO, F., FURTADO, A., *Programação em Lógica e a Linguagem Prolog*. Edgard Blücher, 1987.

[9] DIAS, M., "Notas de aula - unidade 3 - inferência estatística". `http://www.est.ufba.br/mat027/Mat027-Inferencia.pdf`. Último acesso: junho/2008.

[10] DIETTERICH, T. G., "Approximate Statistical Test For Comparing Supervised Classification Learning Algorithms", *Neural Computation*, v. 10, n. 7, pp. 1895–1923, 1998.

[11] DOMINGOS, P., HULTEN, G., "Mining high-speed data streams", In: *Knowledge Discovery and Data Mining*, pp. 71–80, 2000.

[12] DOMINGOS, P., HULTEN, G., "A general method for scaling up machine learning algorithms and its application to clustering", In: *Proceedings of the Eighteenth International Conference on Machine Learning*, (Williamstown, MA), pp. 106–113, Morgan Kaufmann, 2001.

[13] DOMINGOS, P., HULTEN, G., "A General Framework for Mining Massive Data Stream", *Journal of Computational and Graphical Statistics*, v. 12, pp. 945–949, 2003.

[14] FRIEDMAN, J., "Data mining and statistics: What's the connection?", In: *Proceedings of the 29th Symposium on the Interface Between Computer Science and Statistics*, 1997.

[15] H. BLOCKEEL, L. DEHASPE, J. R. J. S. A. V. A. C. V., FIERENS, D., *The ACE Data Mining System - User's Manual*, December 2005.

[16] HOEFFDING, W., "Probability inequalities for sums of bounded random variables", v. 58, pp. 13–30, 1963.

[17] HULTEN, G., DOMINGOS, P., "Mining complex models from arbitrarily large databases in constant time", In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 525–531, 2002.

[18] HULTEN, G., SPENCER, L., DOMINGOS, P., "Mining time-changing data streams", In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (San Francisco, CA), pp. 97–106, ACM Press, 2001.

[19] KOHAVI, R., "A study of cross-validation and bootstrap for accuracy estimation and model selection", In: *IJCAI*, pp. 1137–1145, 1995.

[20] KOK, S., DOMINGOS, P., "Learning the structure of markov logic networks", In: *Proceedings of the Twenty-Second International Conference on Machine Learning*, (Bonn, Germany), pp. 441–448, ACM Press, 2005.

[21] KRAMER, S., LAVRAC, N., FLACH, P., *Propositionalization Approaches to Relational Data Mining*, pp. 262–291. Springer-Verlag, September 2001.

[22] LAVRAC, N., DZEROSKI, S., *Inductive Logic Programming: Techniques and Applications*. New York, Ellis Horwood, 1994.

[23] LAVRAČ, N., ŽELEZNÝ, F., FLACH, P., "Rsd: Relational subgroup discovery through first-order feature construction", In: *Proceedings of the 12th International Conference on Inductive Logic Programming*, Springer, 2002.

[24] LEITE, R., BRAZDIL, P., "Predicting relative performance of classifiers from samples", In: *ICML '05: Proceedings of the 22nd international conference on Machine learning*, (New York, NY, USA), pp. 497–503, ACM, 2005.

[25] MCCALLUM, A., NIGAM, K., RENNIE, J., *et al.*, "Building domain-specific search engines with machine learning techniques", In: *Proc. AAAI-99 Spring Symposium on Intelligent Agents in Cyberspace, 1999*, 1999.

[26] MITCHELL, T., *Machine Learning*. New York, McGraw-Hill, 1997.

[27] MUGGLETON, S., "Inverse Entailment and Progol", *New Generation Computing Journal*, v. 13, pp. 245–286, 1995.

[28] MUGGLETON, S., "Inductive logic programming", *New Generation Computing*, v. 8, n. 4, pp. 295–318, 1991.

[29] MUGGLETON, S., RAEDT, L. D., "Inductive Logic Programming: Theory and Methods", *Journal of Logic Programming*, v. 19/20, pp. 629–679, 1994.

[30] NADEAU, C., BENGIO, Y., "Inference for the Generalization Error", *Machine Learning*, v. 52, n. 3, pp. 239–281, 2003.

[31] PROVOST, F. J., JENSEN, D., OATES, T., "Efficient progressive sampling", In: *Knowledge Discovery and Data Mining*, pp. 23–32, 1999.

[32] PROVOST, F. J., KOLLURI, V., "A Survey of Methods for Scaling Up Inductive Algorithms", *Data Min. Knowl. Discov.*, v. 3, n. 2, pp. 131–169, 1999.

[33] QUINLAN, J. R., "Learning Logical Definitions from Relations", *Machine Learning*, v. 5, n. 3, pp. 239–266, 1990.

[34] QUINLAN, J. R., "Induction of Decision Trees", *Machine Learning*, v. 1, n. 1, pp. 81–106, 1986.

[35] QUINLAN, J. R., *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[36] QUINLAN, J. R., CAMERON-JONES, R. M., "Induction of Logic Programs: FOIL and Related Systems", *New Generation Computing*, v. 13, pp. 287–312, 1995.

[37] RAEDT, L. D., "Logical settings for concept-learning", *Artificial Intelligence*, v. 95, n. 1, pp. 187–201, 1997.

[38] SRINIVASAN, A., "The Aleph manual", tech. rep., Oxford University, 2001.

[39] VAN RIJSBERGEN, C. J., *Information Retrieval, 2nd edition*. Dept. of Computer Science, University of Glasgow, 1979.