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.

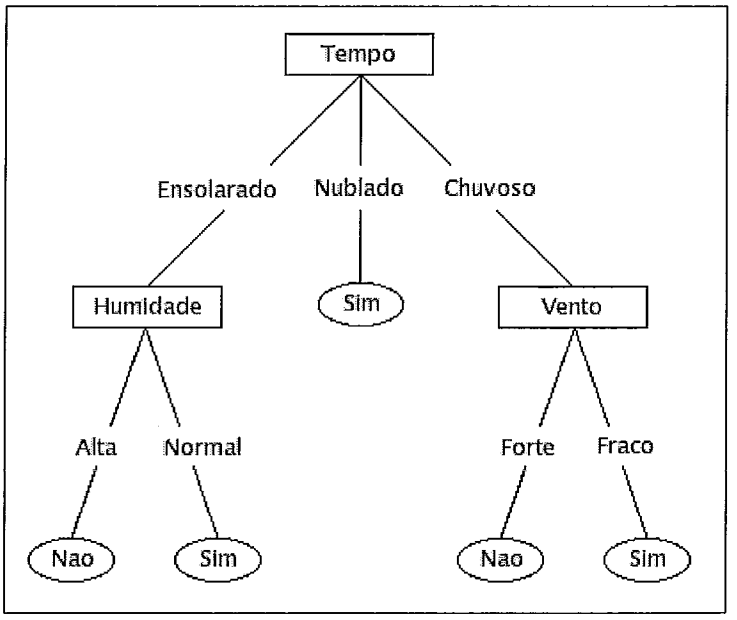


Figura 2.1: Exemplo de uma árvore de decisão para o conceito *JogarTênis*.

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

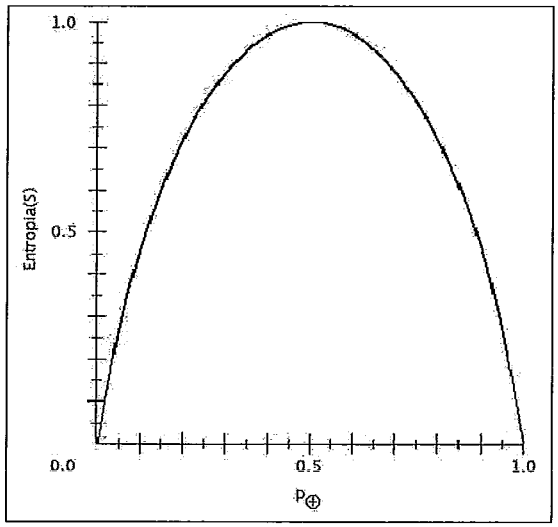


Figura 2.2: Gráfico da entropia relativa a uma classificação booleana. $p_\oplus$ é a proporção de exemplos positivos e varia de 0 a 1.

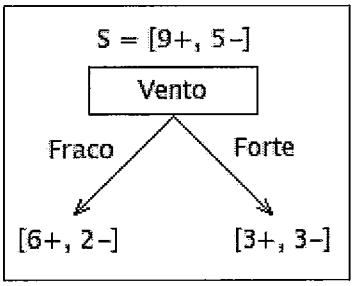
é dada pela equação 2.2, onde p_i é a proporção de exemplos em S pertencentes à classe i. Neste caso, o valor máximo da entropia não é mais 1, como no caso booleano, mas $\log_2 c$.

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

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

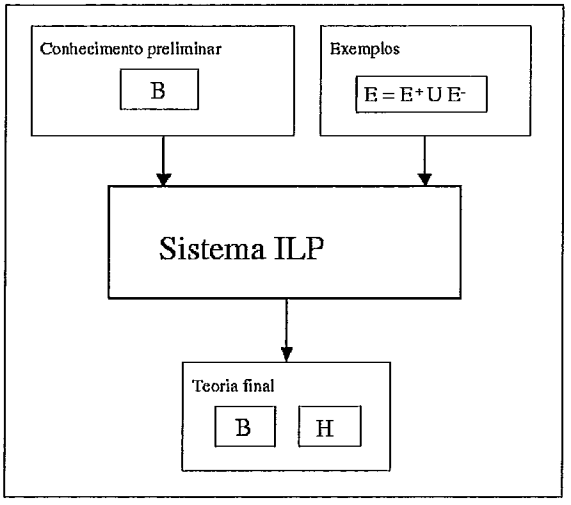


Figura 2.4: Esquema do aprendizado em programação em lógica indutiva (*ILP*), onde B é o conhecimento preliminar, E é o conjunto de exemplos, formado pelos exemplos positivos (E^+) e negativos (E^-), e H é a teoria aprendida pelo sistema.

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.

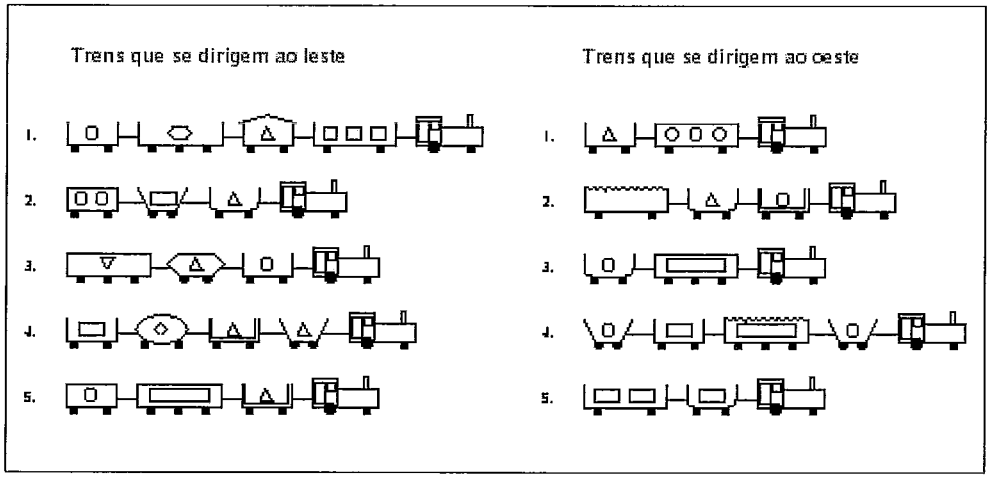


Figura 2.5: Exemplos considerados no problema dos trens.

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.

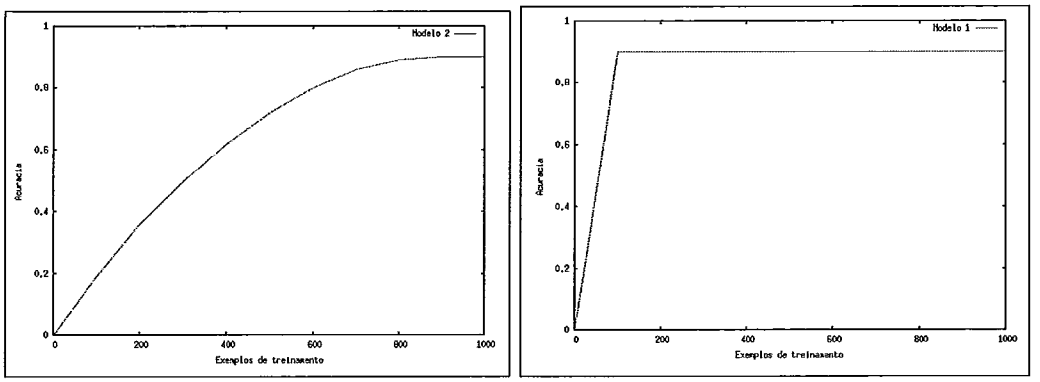


Figura 2.6: Exemplos de curvas de aprendizado, uma com convergência lenta e a outra com convergência rápida.

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].

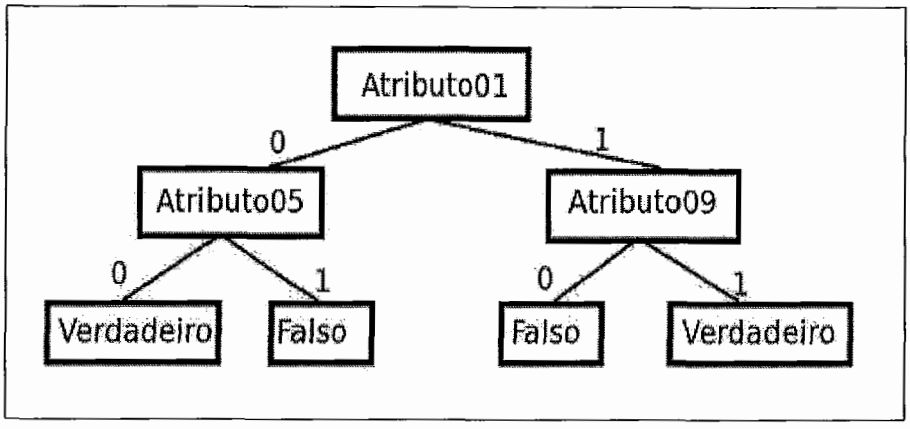


Figura 2.7: Exemplo de uma árvore de decisão gerada pelo terceiro módulo do sistema VFILPh (figura extraída de [7]).

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.

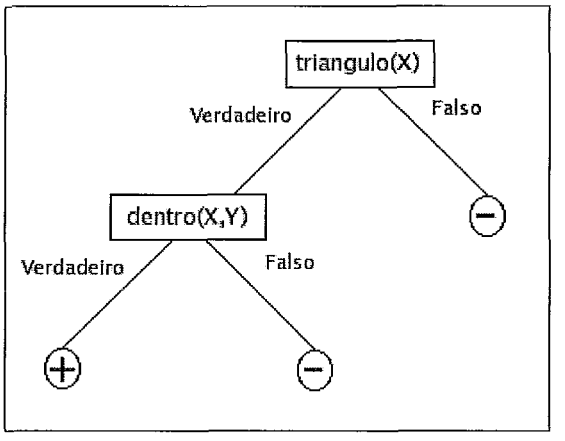


Figura 3.1: Exemplo de uma árvore gerada pelo TILDE.

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-

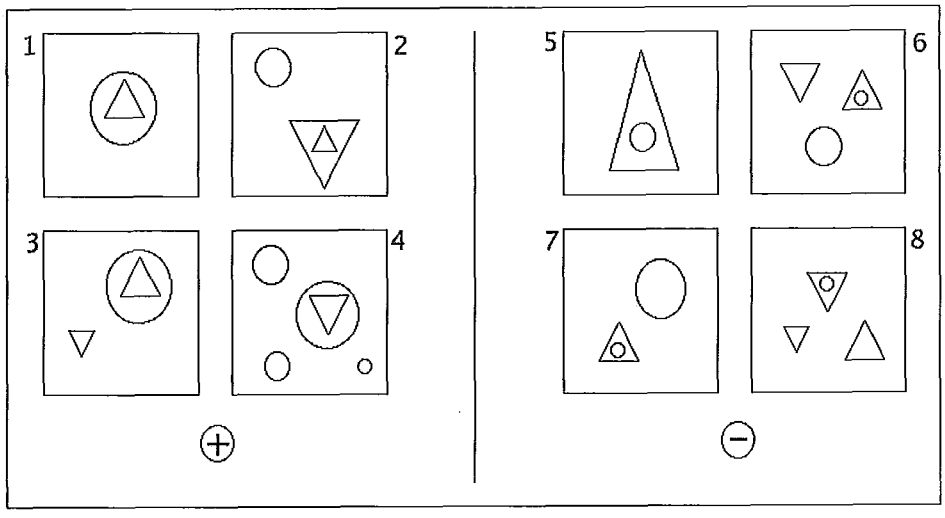


Figura 3.2: Exemplos do problema Bongard.

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.