Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2585

Uma Introdução ao Aprendizado Simbólico de Máquina por Exemplos

Maria Carolina Monard
Gustavo E. de Almeida Prado Alves Batista
Sandra Kawamoto
Jaqueline Brigladori Pugliesi

Nº 29

NOTAS DIDÁTICAS DO ICMC

São Carlos

Outubro/1997

Uma Introdução ao Aprendizado Simbólico de Máquina por Exemplos*

Maria Carolina Monard
Gustavo E. de Almeida Prado Alves Batista
Sandra Kawamoto
Jaqueline Brigladori Pugliesi

Universidade de São Paulo/ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP, Brasil
e-mail: {mcmonard,gbatista,jbpuglie}@icmsc.sc.usp.br

Resumo

Aprendizado de Máquina – AM – é uma importante área em Inteligência Artificial já que a capacidade de aprender é essencial para um comportamento inteligente. A área de AM estuda métodos computacionais apropriados para a aquisição de novos conhecimentos, novas habilidades e novas formas de organização do conhecimento já existente.

Este trabalho didático tem como objetivo contextualizar os diversos paradigmas de Aprendizado de Máquina bem como descrever algumas implementações de algoritmos de Aprendizado Simbólico de Máquina por exemplos. Nessas implementações é dada ênfase à compreensão e solução dos problemas que eles resolvem e não, necessariamente, á eficiência das implementações.

Versão 1.0
Outubro 1997

*Trabalho realizado com auxílio parcial do CNPq, CAPES, FAPESP e Programa PAE — Programa de Aperfeiçoamento de Ensino — da USP.

Material didático visual, relacionado com esta Nota, está sendo desenvolvido em PowerPoint dentro do projeto *Desenvolvimento de Material Didático na Área de Inteligência Artificial* apoiado pela Próreitoria de Graduação da USP.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Aprendizado de Máquina – AM – é uma sub-área de pesquisa muito importante em Inteligência Artificial – IA –, pois a capacidade de aprender é essencial para um comportamento inteligente. AM estuda métodos computacionais para adquirir novos conhecimentos, novas habilidades e novos meios de organizar o conhecimento já existente. O estudo de técnicas de aprendizado baseado em computador também pode fornecer um melhor entendimento de nosso próprio processo de raciocínio.

Uma das críticas mais comuns à IA é que as máquinas só podem ser consideradas inteligentes quando forem capazes de aprender novos conceitos e se adaptarem a novas situações, em vez de simplesmente fazer o que lhes for mandado. Não há muita dúvida de que uma importante característica das entidades inteligentes é a capacidade de adaptar-se a novos ambientes e de resolver novos problemas. É possível incorporar tais habilidades em programas? Ada Augusta, uma das primeiras filósofas em computação, escreveu

> A Máquina Analítica (*Analytical Engine*) não tem qualquer pretensão de *originar* nada. Ela pode fazer qualquer coisa desde que nós *saibamos como mandá-la* executar.

Esse comentário foi interpretado por vários críticos de IA como uma indicação de que os computadores não são capazes de aprender. Entretanto, nada impede que digamos a um computador como interpretar as informações recebidas, de uma maneira que melhore gradualmente seu desempenho.

Em vez de perguntar antecipadamente se é possível ou não que os computadores "aprendam", é muito mais esclarecedor tentar descrever exatamente a quais atividades nos referimos quando falamos em "aprender"e quais mecanismos podem ser usados para que seja possível executar essas atividades.

Este trabalho tem como objetivo contextualizar o Aprendizado de Máquina, bem como descrever algumas implementações de algoritmos de AM. Deve ser salientado que na implementação desses algoritmos é dada ênfase à compreensão e solução de problemas que eles resolvem e não, necessariamente, à eficiência das implementações.

O trabalho está organizado da seguinte forma: na seção 2 são descritas algumas características de aprendizado humano e de Aprendizado de Máquina. Na seção 3 são apresentados brevemente alguns dos paradigmas mais difundidos de AM. Na seção 4 é discutida uma classificação das estratégias de AM. A seção 5 trata de argumentos dedutivos e indutivos. Na seção 6 é descrito aprendizado por analogia e um algoritmo para resolver alguns problemas de analogia é apresentado em detalhe. A seção 7 descreve mais formalmente Aprendizado de Máquina indutivo e na seção 8 são apresentados dois algoritmos de AM por exemplos. Na seção 9 é discutido aprendizado utilizando árvores de decisão; o algoritmo geral para construir árvores de decisão na forma "top down" é apresentado junto com a implementação. Finalmente, na seção 10 são apre-

sentas as conclusões deste trabalho bem como vários sites, na Web, relacionados com Aprendizado de Máquina.

2 Aprendizado

Segundo Herbert Simon [Simon 83]

> *Aprendizado denota mudanças no sistema, que são adaptáveis no sentido de que elas possibilitam que o sistema faça a mesma tarefa ou tarefas sobre uma mesma população, de uma maneira mais eficiente a cada vez.*

Definida dessa maneira, o aprendizado abrange uma ampla escala de fenômenos. Em uma extremidade do espectro está o *refinamento de habilidades.* Em geral, as pessoas melhoram a execução de tarefas através da prática. Usualmente, quanto mais uma pessoa anda de bicicleta ou joga tênis, melhor fica seu desempenho. Na outra extremidade do espectro está a *aquisição de conhecimentos.* Sistemas de IA dependem fortemente de conhecimento que é normalmente adquirido através da experiência.

O estudo e a atividade de modelar o processo de aprendizado e suas múltiplas manifestações são as principais metas de AM. Sistemas de AM são aqueles capazes de extrair conhecimento de novos dados. Eles devem ter a habilidade de não apenas adquirir conhecimento de forma acumulativa, mas também de absorvê-lo.

2.1 Características do Aprendizado Humano

O homem é capaz de aprender habilidades motoras e cognitivas. A visão é o principal meio para adquirir habilidades motoras, mas mesmo se não tivermos a visão, nós ainda somos capazes de aprender as mesmas habilidades motoras. A linguagem é muito importante na aquisição de habilidades cognitivas. A fala é uma característica particular do homem, apesar de não ser inata: a fala tem que ser aprendida. Aprendizado é a chave da superioridade da inteligência humana. Um aumento significativo da inteligência da máquina será resultado do aumento das capacidades de aprendizado da mesma. Aprendizado é a essência da inteligência.

2.2 Aprendizado de Máquina

Aprendizado de Máquina é fundamental em IA e sua metodologia tem se desenvolvido de acordo com os principais interesses do ramo. Em resposta às dificuldades de codificar volumes de conhecimento sempre crescentes em modernos sistemas de IA, muitos pesquisadores têm voltado sua atenção para AM como meio de vencer o gargalo da aquisição de conhecimento.

O processo de aprendizado inclui a aquisição de novos conhecimentos, o desenvolvimento das habilidades motoras e cognitivas através de instrução e prática, a organização de novos conhecimentos em geral ou formas efetivas de representar o conhecimento, e a descoberta de novos fatos e teorias por meio de observações e experimentações. O estudo e a modelagem computacional dos processos de aprendizado em suas múltiplas manifestações constituem o principal objetivo do AM.

A habilidade de aprender por observação e experiência parece ser crucial para qualquer criatura inteligente. É difícil dar uma definição geral sobre aprendizado, mas comecemos com a seguinte

Aprendizado é a habilidade de aperfeiçoar-se em uma determinada questão.

Por exemplo, a questão pode ser reconhecer uma determinada fisionomia, fazer um prognóstico temporal, ou obter a derivada de uma função. Essa definição é comportamental na medida que o ator da questão é como uma caixa preta, julgando o seu desempenho apenas pelo seu exterior. Faremos aqui uma suposição adicional de que o ator consiste de um Motor de Inferência procedimental e uma Base de Conhecimento declarativa, e que o seu desempenho é aperfeiçoado melhorando a Base de Conhecimento ao invés do Motor de Inferência. Por exemplo, o conhecimento necessário para reconhecer fisionomias pode ser expresso em termos de regras de classificação, listando as características de uma fisionomia específica; a habilidade de reconhecer fisionomias pode ser aperfeiçoada melhorando as regras de classificação. Nesse ponto de vista

Aprendizado é a habilidade de adquirir novos conhecimentos.

Nem todos os paradigmas de AM realizam tal separação entre o conhecimento e o motor de inferência, como exemplo pode-se citar as redes neurais. Esse paradigma é brevemente decrito na seção 3.4.

3 Paradigmas de Aprendizado de Máquina

Atualmente já foram propostos diversos paradigmas de AM. Esta seção procura apresentar brevemente alguns desses paradigmas, tais como o paradigma simbólico, estatístico, Instance-based, conexionista e genético.

3.1 Paradigma Simbólico

Os sistemas de aprendizado simbólico buscam aprender construindo representações simbólicas de um conceito através da análise de exemplos e contra-exemplos desse conceito. As representações simbólicas estão tipicamente na forma de alguma expressão lógica, árvore de decisão, regras de produção ou rede semântica.

Atualmente, entre as representações simbólicas mais estudadas estão as árvores e regras de decisão. É atribuído a Morgan e Messeger [Morgan 73], o desenvolvimento original do programa para a indução de árvores de decisão. Métodos de indução de árvores de decisão a partir de dados empíricos, conhecido como *particionamento recursivo*, foram estudados por pesquisadores da área de Inteligência Artificial e Estatística. Os sistemas ID3 [Quinlan 86] e C4 [Quinlan 87b] para indução de árvores de decisão tiveram uma importante contribuição sobre a pesquisa em Inteligência Artificial. O sistema de classificação de árvores de regressão [Breiman 84] foi desenvolvido por estatísticos, durante praticamente o mesmo período que o ID3, no final dos anos 70.

Os trabalhos com indução de regras de decisão surgiram com a simples tradução das árvores de decisão para regras, com a poda realizada sobre as regras, tal abordagem surgiu em [Quinlan 87a, Quinlan 87b]. Posteriormente, foram criados métodos que induziam regras diretamente a partir dos dados, um exemplo deste trabalho pode ser encontrado em [Michalski 86].

Discutir métodos e técnicas de aprendizado simbólico é o maior objetivo deste trabalho, dessa forma aprendizado simbólico é melhor detalhado nas seções seguintes.

3.2 Paradigma Estatístico

Pesquisadores em estatística têm criado muitos métodos de classificação, muitos deles semelhantes aos métodos empregados em aprendizado de máquina. Por exemplo, o método CART [Breiman 84], um sistema muito conhecido para montar árvores de decisão, foi desenvolvido por estatísticos. Como regra geral, técnicas estatísticas tendem a focar tarefas em que todos os atributos têm valores contínuos ou ordinais. Muitos deles também são paramétricos, assumindo alguma forma de modelo, e então encontrando valores apropriados para os parâmetros do modelo a partir de dados. Por exemplo, um classificador linear assume que classes podem ser expressas como combinação linear dos valores dos atributos, e então procurar uma combinação linear particular que fornece a melhor aproximação sobre o conjunto de dados. Os classificadores estatísticos

freqüentemente assumem que valores de atributos estão normamente distribuídos, e então usam os dados fornecidos para determinar média, variância e co-variância da distribuição.

Alguns autores têm considerado redes neurais como métodos estatísticos paramétricos uma vez que treinar uma rede neural geralmente significa encontrar valores apropriados para pesos e *bias* pré-determinados.

3.3 Paradigma Instance-based

Uma forma de classificar um caso é lembrar de um caso similar cuja classe é conhecida e assumir que o novo caso terá a mesma classe. Esta filosofia exemplifica os sistemas instance-based, que classificam casos nunca vistos através de casos similares conhecidos [Quinlan 88].

As características principais dos sistemas instance-based são

- Quais casos de treinamento devem ser lembrados? Se todos os casos forem memorizados, o classificador pode se tornar lento e difícil de manusear. O ideal é reter casos prototípicos que juntos resumem toda a informação importante, esta abordagem pode ser observada em livros médicos e legais. Em [Aha 91] estão descritas algumas estratégias para decidir quando um novo caso deve ser memorizado.
- Como deve ser medida a similaridade entre os casos? Se todos os atributos forem contínuos, pode-se calcular a distância entre dois casos como a raiz quadrada da soma dos quadrados da diferença dos atributos. Quando alguns atributos não são ordinais, esta interpretação de distância se torna mais problemática. Além do mais, se existem muitos atributos irrelevantes, dois casos similares podem aparentar serem muito diferentes pois eles possuem valores diferentes em atributos sem importância. Em [Stanfill 86] foi desenvolvido um método sensível ao contexto para alterar a escala dos atributos de forma que as medidas de distância fiquem mais robustas.
- Como um novo caso deve ser relacionado com casos armazenados? Existem duas alternativas que são usar um simples caso armazenado o qual é o mais próximo do novo caso, ou usar vários casos levando-se em consideração os diferentes graus de similaridade entre cada caso.

3.4 Paradigma Conexionista

Redes neurais são construções matemáticas relativamente simples que foram inspiradas no modelo biológico do sistema nervoso. Sua representação envolve unidades altamente interconectadas, no qual o nome *conexionismo* é utilizado para descrever a área de estudo.

A metáfora biológica com as conexões neurais do sistema nervoso tem interessado muitos pesquisadores, e tem fornecido muitas discussões sobre os méritos e as limitações dessa abordagem de aprendizado. Em particular, as analogias com a biologia têm levado muitos pesquisadores a acreditar que as redes neurais possuem um grande potencial na resolução de problemas que requerem intenso processamento sensorial humano, tal como visão e reconhecimento de voz.

As pesquisas em redes neurais foram iniciadas com o trabalho pioneiro de McCulloch e Pitts em [McCulloch 43]. McCulloch era um psiquiatra e pesquisou por 20 anos uma forma de representar um evento no sistema nervoso. Pitts era um jovem pesquisador e começou a trabalhar com McCulloch em 1942. Praticamente 15 anos após a publicação de McCulloch e Pitts, Rosenblatt em [Rosenblatt 58] apresentou o perceptron, cuja grande contribuição foi a prova do teorema de convergência. Mas no livro *Perceptrons*, [Minsky 69], Minsky e Papert demonstraram a existência de limites fundamentais nos perceptrons de uma camada. A pesquina na área ficou praticamente estática até que Hopfield em [Hopfield 82] utilizou a idéia de uma função de energia para formular uma nova forma de compreender os cálculos realizados em redes recorrentes com conexões sinápticas simétricas.

Talvez mais que qualquer outra publicação, o artigo de Hopfield em 1982 e o livro de Rumelhart e McLelland [Rumelhart 86] foram as publicações que mais influenciaram para o resurgimento do interesse sobre redes neurais na década de 80. Redes neurais tiveram um longo caminho desde McCulloch e Pitts, e continuarão a crescer em teoria, projetos e aplicações [Haykin-94].

3.5 Paradigma Genético

Este formalismo de classificação é derivado do modelo evolucionário de aprendizado [Holland 86]. Um classificador genético consiste de uma população de elementos de classificação que competem para fazer a predição. Elementos que possuem uma performance fraca são descartados, enquanto os elementos mais fortes proliferam, produzindo variações de si mesmos. Este paradigma possue uma analogia direta com a teoria de Darwin, onde sobrevivem os mais bem adaptados ao ambiente. Segundo [Freitas 92]

> *Um algoritmo genético é um procedimento iterativo que mantém uma população de indivíduos, cada um dos quais é um candidato à solução de algum problema específico. A cada iteração (denominada geração), os indivíduos da população atual são avaliados quanto a sua aptidão para a solução do problema. Com base nessas avaliações aplicam-se alguns operadores genéticos aos indivíduos, formando-se uma nova população, que substituirá a população atual. Isto é feito de modo que, quanto maior a aptidão de um indivíduo da população atual, maior a sua influência na formação dos indivíduos da nova geração. Assim, com o passar do tempo, a seleção natural tende a fazer com que a população seja formada por indivíduos cada vez melhores (soluções cada vez mais próximas da solução ótima para o proble-*

ma). Como critério de parada do algoritmo genético, geralmente define-se um limite no número de gerações. Dentre os indivíduos da última geração, aqueles mais aptos representam a melhor solução encontrada pelo algoritmo. Pode-se também especificar que o algoritmo encerrará quando for gerado algum indivíduo que satisfaça alguma condição mínima de aptidão.

Alguns operadores genéticos básicos que aplicados a população geram novos indivíduos são discutidos a seguir. Esses operadores são denominados

1. Reprodução
2. Cruzamento
3. Mutação
4. Inversão

1. Reprodução. É a quantidade de cópias que um indivíduo produz, proporcional à sua aptidão. Essas cópias serão submetidas à ação de outros operadores genéticos na geração de novos indivíduos.
2. Cruzamento[†]. É a troca de material genético entre indivíduos. Por exemplo, considerando os seguintes indivíduos representados por uma cadeia de símbolos de comprimento 6

   Indivíduo 1: $\boxed{x1\ x2\ x3\ x4\ x5\ x6}$

   Indivíduo 2: $\boxed{y1\ y2\ y3\ y4\ y5\ y6}$

   aplicando o operador de cruzamento na sexta posição, são gerados os seguintes novos indivíduos

   Indivíduo 3: $\boxed{x1\ x2\ x3\ x4\ x5\ \mathbf{y6}}$

   Indivíduo 4: $\boxed{y1\ y2\ y3\ y4\ y5\ \mathbf{x6}}$
3. Mutação. Como em Biologia, a mutação, geralmente, é importante para uma espécie sobreviver. Aplicada aos algoritmos genéticos, a mutação modifica o valor de uma posição aleatória da cadeia de símbolos que representa o indivíduo.

   Por exemplo, dado um indivíduo representado por uma cadeia de símbolos binários

   Indivíduo 1: $\boxed{0\ 1\ 0\ \mathbf{0}\ 1\ 0\ 1\ 0}$

[†]Crossover

aplicando o operador de mutação na quarta posição da cadeia é gerado o seguinte indivíduo

Indivíduo 2: 0 1 0 **1** 1 0 1 0

A vantagem deste operador é que a probabilidade de pesquisar pela solução de um problema em qualquer região do espaço nunca será zero, virtude que os outros operadores não possuem.

4. Inversão. É a troca de símbolos na cadeia que representa o indivíduo. Símbolos em posições mais próximas têm maior probabilidade de serem rearranjados dentro da cadeia.

   Por exemplo, dado o Indivíduo 1, o operador de inversão (posições 3 e 5) gera o Indivíduo 2

Indivíduo 1: *x1 x2* **x3** *x4* **x5** *x6*

Indivíduo 2: *x1 x2* **x5** *x4* **x3** *x6*

Segundo [De Jong 88] os algoritmos genéticos é uma técnica utilizada para solucionar problemas de otimização e de aprendizado de máquina, utilizando simulações e regras probabilísticas (e não determinísticas). O sistema trabalha na forma de tentativa e erro com alguma forma de retroalimentação do erro, onde consequentemente "aprende" soluções melhores para os problemas. O algoritmo nem sempre encontra uma solução correta, mas procura adquirir novos conhecimentos e chegar o mais próximo possível da solução.

4 Classificação das Estratégias de Aprendizado de Máquina

Em qualquer processo de aprendizado, o aprendiz usa o conhecimento que possui para obter novo conhecimento. Este novo conhecimento é então lembrado para uso posterior. O aprendizado de um novo conceito pode ser realizado de várias maneiras.

O tipo de inferência que um estudante desempenha sobre as informações disponíveis define uma estratégia de aprendizado e pode constituir um bom critério de classificação dos processos de aprendizado de máquina.

As estratégias básicas de aprendizado de conceito são apresentadas a seguir em ordem crescente de complexidade de inferência desempenhada pelo aprendiz. De modo geral, essa ordem reflete uma dificuldade crescente para o aprendiz aprender um conceito bem como a dificuldade decrescente para o instrutor ensinar o conceito, como mostrado na Figura 1.

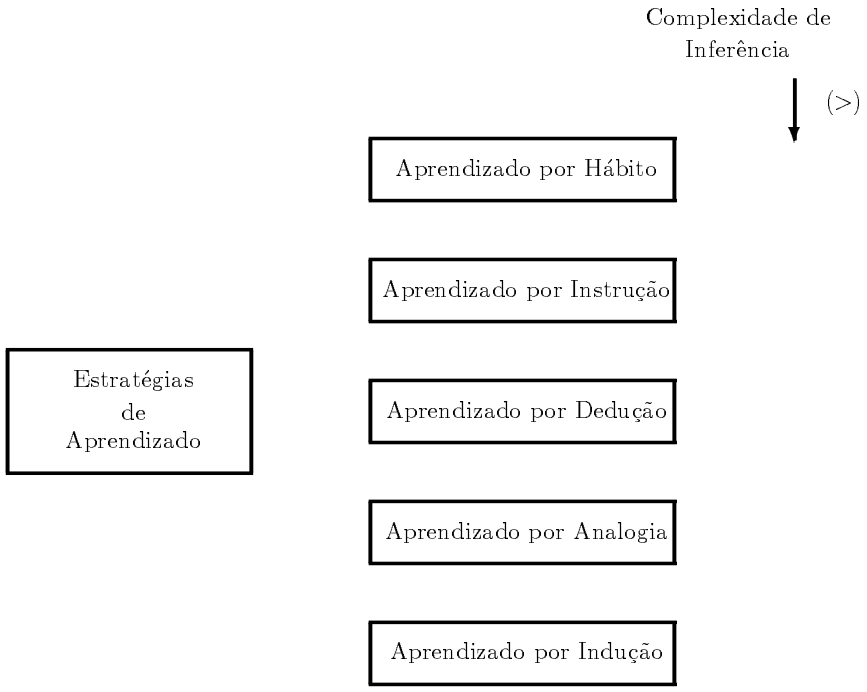


Figura 1: Classificação das Estratégias de AM

Deve-se notar que essa classificação aplica-se não somente ao aprendizado de conceitos, mas também a qualquer modo de adquirir conhecimento. A seguir são discutidas brevemente cada uma dessas estratégias básicas de aprendizado.

4.1 Aprendizado por Hábito

Nesse tipo de aprendizado, o aprendiz não precisa desempenhar nenhuma inferência sobre a informação fornecida. O conhecimento é diretamente assimilado pelo aprendiz. Essa estratégia inclui o aprendizado por memorização direta de descrições de um conceito. Por exemplo, em AM, essa estratégia é empregada quando um algoritmo específico para reconhecer um conceito é programado em um computador, ou quando é construída uma base de dados fatuais sobre o conceito.

4.2 Aprendizado por Instrução

Aqui, o aprendiz adquire conceitos de uma fonte, como por exemplo, um professor, uma publicação ou um livro texto, mas não copia diretamente a informação fornecida para a memória. O aprendizado engloba a seleção dos fatos mais relevantes e/ou a transformação da informação fonte em formas mais apropriadas.

4.3 Aprendizado por Dedução

Dedução é definida como *inferência logicamente correta*, isto é, a conclusão obtida de várias premissas iniciais verdadeiras sempre preserva a verdade. A inferência atua somente nas informações contidas no conhecimento, nada além disso [Raggett 92].

No Aprendizado por Dedução, o aprendiz adquire um conceito através de dedução sobre o conceito já adquirido. Isto é, essa estratégia inclui qualquer processo no qual o conhecimento aprendido é o resultado de uma transformação sobre um conhecimento já possuído, que preserva veracidade. Em geral, no aprendizado dedutivo realiza-se uma seqüência de deduções ou cálculos sobre a informação presente, memorizando o resultado. Uma forma de dedução é o Aprendizado Baseado em Explicação — ABE — que transforma a definição de um conceito para uma definição operacional utilizando um exemplo como guia. ABE é descrito em detalhes em [DeJong 86].

4.4 Aprendizado por Analogia

Nessa estratégia, o aprendiz adquire um novo conceito modificando a definição de um conceito semelhante já conhecido. Em vez de formular ao acaso uma regra para um novo conceito, adapta-se uma regra existente modificando-a apropriadamente para que possa ser aplicada ao novo conceito. Por exemplo, se alguém tem o conceito de uma laranja, o aprendizado do conceito de uma tangerina pode ser muito simples, apenas notando as semelhanças e as diferenças entre as duas.

O Aprendizado por Analogia pode ser visto como a combinação do aprendizado indutivo com o dedutivo. Através de inferência indutiva, determinam-se características gerais unindo os conceitos que estão sendo comparados. Depois, por inferência dedutiva e

a partir dessas características, determina-se o conceito a ser aprendido. Esse tipo de aprendizado é melhor descrito na seção 6, pg. 17.

4.5 Aprendizado por Indução

A habilidade dos seres humanos de fazer generalizações a partir de alguns fatos, ou descobrir padrões em coleções de observações aparentemente caóticas, é atingida através do aprendizado por indução.

Indução é a forma de inferência lógica que permite que conclusões gerais sejam obtidas de exemplos particulares. É caracterizado como o raciocínio que parte do específico para o geral, do particular para o universal, da parte para o todo. No Aprendizado por Indução, o aprendiz adquire um conceito fazendo inferências indutivas sobre os fatos apresentados. Hipóteses geradas pela inferência indutiva podem ou não preservar a verdade.

A seguir é abordado o problema de indução lógica, suas características e diferenças em relação à dedução lógica. Ambos os conceitos são de fundamental importância para a compreensão de aprendizado indutivo e dedutivo.

5 Dedução e Indução

A lógica trata de argumentos e inferências. Um de seus propósitos básicos é apresentar métodos capazes de identificar os argumentos logicamente válidos. Um argumento é uma coleção de enunciados que estão relacionados uns com os outros. Um desses enunciados é a conclusão e o restante, um ou mais, constituem a evidência corroborativa e são chamados de premissas.

Considere o seguinte exemplo de argumento lógico, independentemente dos estudantes de Ciência de Computação – CC – gostarem ou não de Inteligência Artificial.

Todos os estudantes de CC gostam de IA
Sandra é uma estudante de CC
Sandra gosta de IA

Nesse caso, o argumento é formado por três enunciados. Os dois primeiros enunciados são as premissas e o terceiro é a conclusão. No que segue, os argumentos são apresentados de uma forma padronizada, escrevendo as premissas em primeiro lugar e a seguir a conclusão, identificando-a com o símbolo $\rightarrow$.

Todos os estudantes de CC gostam de IA
Sandra é uma estudante de CC
$\rightarrow$ *Sandra gosta de IA*

Em geral, os argumentos lógicos são divididos em duas grandes classes

1. argumentos dedutivos e
2. argumentos indutivos

Exemplo de argumento dedutivo

Todo mamífero tem coração
Todos os felinos são mamíferos
$\rightarrow$ *Todos os felinos têm coração*

Exemplo de argumento indutivo

Todos os felinos que foram observados tinham coração.
$\rightarrow$ *Todos os felinos têm coração.*

Há certas características básicas, discutidas a seguir, que distinguem argumentos dedutivos de argumentos indutivos.

5.1 Argumentos Dedutivos

O argumento dedutivo destina-se a deixar explícito o conteúdo das premissas, enquanto que o argumento indutivo destina-se a ampliar o alcance de nossos conhecimentos. A relação que há entre a generalização científica e a observação de evidências é de tipo indutivo.

A validade de um argumento dedutivo depende, exclusivamente, da relação que se estabelece entre as premissas e a conclusão. Dizer que um argumento dedutivo é válido significa dizer que as premissas estão de tal modo relacionadas com a conclusão que a conclusão é verdadeira quando as premissas são verdadeiras. Validade é uma propriedade dos argumentos — que são uma coleção de enunciados — e não uma propriedade dos enunciados isoladamente considerados. Verdade, por outro lado, é uma propriedade de enunciados isolados, não de argumentos.

5.2 Argumentos Indutivos

Os argumentos indutivos, ao contrário do que sucede com os dedutivos, levam a conclusões cujos conteúdos excedem os das premissas. Indução é a forma de inferência lógica que permite que conclusões gerais sejam obtidas de exemplos particulares. É caracterizada como o raciocínio que parte do específico para o geral, do particular para o universal, da parte para o todo. Hipóteses geradas pela inferência indutiva podem ou não preservar a verdade. É esse traço característico da indução que torna os argumentos indutivos indispensáveis para a fundamentação de uma significativa porção de nossos conhecimentos. Entretanto, é esse mesmo fato que levanta questões extremamente complicadas, dificultando a análise de resultados obtidos com auxílio de métodos indutivos.

Ao contrário do que sucede com um argumento dedutivo e válido, um argumento indutivo e correto pode, perfeitamente, admitir uma conclusão falsa, ainda que suas premissas sejam verdadeiras. Mesmo não podendo garantir que a conclusão de um argumento seja verdadeira quando as premissas são verdadeiras, pode-se afirmar que as premissas de um argumento indutivo correto sustentam ou atribuem certa verossimilhança a sua conclusão. Quando as premissas de um argumento indutivo são verdadeiras, o melhor que pode ser dito é que a sua conclusão é *provavelmente* verdadeira. Uma exceção disso é a indução matemática. Em um argumento matemático indutivo correto, partindo de premissas verdadeiras obtem-se, invariavelmente, conclusões verdadeiras.

Há certos enganos que podem tornar os argumentos indutivos completamente inúteis ou inúteis de um ponto de vista prático. Enganos desse gênero são denominados *falácias indutivas*. Quando um argumento indutivo é falaz, as premissas não sustentam a conclusão. Entre os argumentos indutivos corretos, porém, pode-se cogitar um grau de

sustentação ou de apoio. As premissas de um argumento indutivo correto podem tornar a conclusão extremamente provável, moderadamente provável ou provável com certo grau de certeza.

Há uma segunda diferença entre os argumentos indutivos e os dedutivos. Dado um argumento dedutivo válido, é possível acrescentar novas premissas, colocando-as com as já existentes, sem afetar a validade do argumento. Em contraste, o grau de sustentação que as premissas de um argumento indutivo conferem à conclusão pode ser alterado por evidências adicionais, acrescentadas ao argumento sob a forma de novas premissas que figurem ao lado das premissas inicialmente consideradas. Como a conclusão de um argumento indutivo pode ser falsa mesmo quando as premissas forem verdadeiras, a evidência adicional, admitindo que relevante, pode nos capacitar a determinar, com mais precisão, se a conclusão é de fato verdadeira. A evidência adicional pode afetar o grau de sustentação da conclusão.

5.3 Relação entre Indução e Dedução

As inferências feitas pelos métodos de indução e dedução relacionadas com aprendizado necessitam, em sua base, de um conhecimento de fundo. O conhecimento de fundo contém conceitos gerais e específicos do domínio para interpretar as premissas na realização da tarefa de inferência. Isto inclui conceitos aprendidos anteriormente, restrições do domínio, hipóteses candidatas, metas a serem inferidas, métodos para avaliar as hipóteses candidatas, etc. A relação entre os métodos de indução e dedução é mostrada na Figura 2 [Michalski 87].

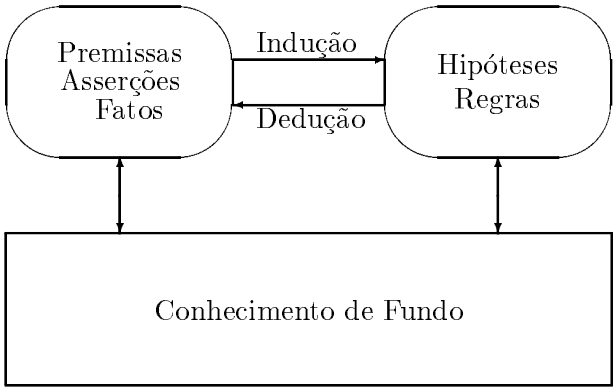


Figura 2: Relação entre Dedução e Indução

A Tabela 1 mostra um resumo das características básicas que distinguem os argumentos dedutivos dos indutivos [Castineira 90].

Argumentos Dedutivos	Argumentos Indutivos
Se todas as premissas forem verdadeiras, a conclusão será verdadeira.	Se todas as premissas forem verdadeiras, a conclusão será provavelmente verdadeira — mas não necessariamente verdadeira — com exceção dos argumentos matemáticos indutivos.
Toda a informação do conteúdo factual da conclusão já está, pelo menos implicitamente, nas premissas.	A conclusão contém informação que excede a das premissas.

Tabela 1: Argumentos Dedutivos *versus* Indutivos

6 Aprendizado por Analogia

O Aprendizado Analógico consiste, basicamente, na transferência de conhecimento de como realizar uma tarefa bem compreendida a outra menos compreendida. É difícil ensinar a um programa de IA que não possui a capacidade de fazer analogias. Nossa linguagem e raciocínio estão carregados de analogias. Na maioria das vezes, solucionamos problemas fazendo analogias com situações já conhecidas. Por exemplo, considere as seguintes sentenças

Marcelo parece uma chaminé (Marcelo fuma muito)

Fernanda é um palito (Fernanda é muito magra)

Aparentemente, cada uma dessas sentenças mistura conceitos que não possuem relação. Entretanto, por trás de cada um desses exemplos há um mapeamento complicado entre os objetos *Marcelo* e *chaminé* bem como *Fernanda* e *palito*. Por exemplo, para entender a segunda sentença deve ser feito o seguinte

1. pegar uma propriedade chave de *palito*, seja "ser fino"
2. perceber que há uma analogia entre 'magreza' e "ser fino"

Esta tarefa não é fácil. O espaço de possíveis analogias é muito grande. Não estamos interessados em considerar situações do tipo

Fernanda é um palito porque é de madeira

Assim, dadas duas ou mais situações, a dificuldade está em distinguir quais aspectos são semelhantes e quais não. Existem dois métodos de solução de problemas por analogia estudados em IA

1. *analogia transformacional* e
2. *analogia derivacional.*

discutidos a seguir.

6.1 Analogia Transformacional e Derivacional

Na *Analogia Transformacional*, a idéia é transformar a solução de um problema anterior em solução do problema atual. Carbonell [Carbonell 83] descreve um método para transformar soluções antigas em soluções novas da seguinte forma — Figura 3 [Rich 93].

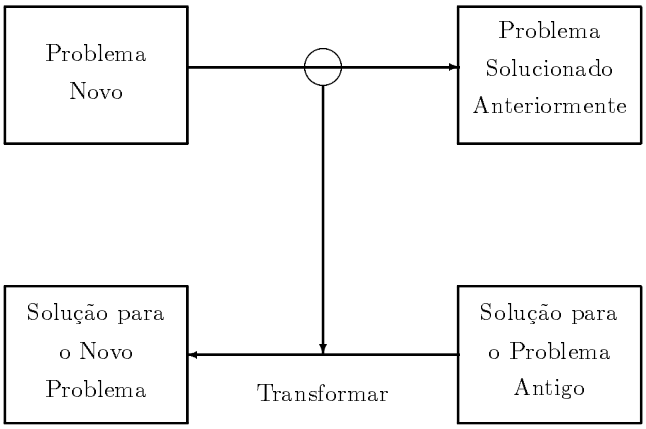


Figura 3: Analogia Transformacional

> *Soluções completas são vistas como estados em um espaço de problemas chamado espaço T. Operadores T prescrevem métodos para transformar uma solução (estado) em outra. O raciocínio por analogia transforma-se em uma busca no espaço T: a partir de uma solução antiga, usamos a análise de meios-fins (ou algum outro método) para encontrar uma solução para o problema atual.*

Note que esse tipo de analogia não analisa *como* o problema foi resolvido, e sim, apenas a solução final. Entretanto, nem sempre é suficiente só a análise da solução final.

Na *Analogia Derivacional*, leva-se em consideração os passos utilizados para solucionar um problema antigo — Figura 4. Seguem algumas definições [Rich 93].

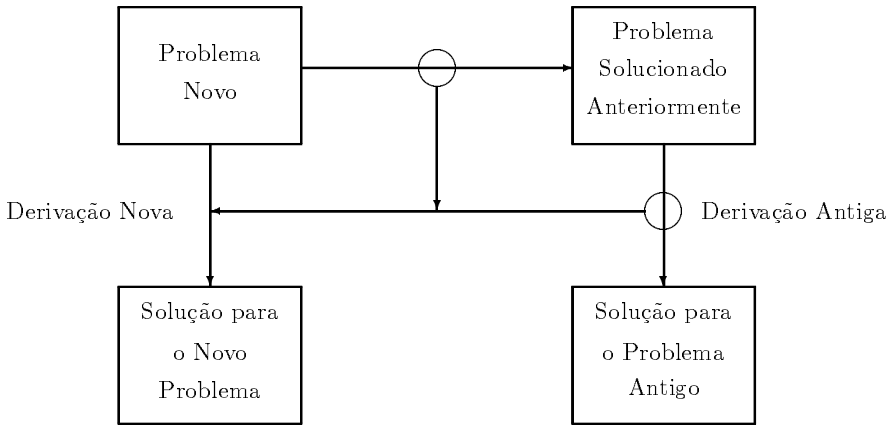


Figura 4: Analogia Derivacional

- *derivação*: é a história detalhada de um episódio de solução de problema.

- *analogia derivacional*: é o raciocínio analógico que leva essas histórias em consideração.

Carbonell afirma que a analogia derivacional é um componente necessário na transferência de habilidades (ou técnicas) em domínios complexos.

6.2 Dificuldades na Resolução de Problemas Análogos

Em geral, uma dificuldade encontrada na resolução de problemas análogos é que esse tipo de mapeamento não é unicamente invertível — conseqüência do não determinismo da generalização. Dois exemplos para resolver problemas de geometria, encontrados em [Anderson 83] e [Anderson 79] respectivamente, ilustram bem essa dificuldade. No primeiro exemplo — Figura 5 — o aprendiz resolve o Problema I.1, isto é, prova que $\overline{RN} = \overline{OY}$, usando o seguinte argumento

$$\begin{array}{rcl} \overline{RO} & = & \overline{NY} \\ \overline{ON} & = & \overline{ON} \\ \overline{RO} + \overline{ON} & = & \overline{ON} + \overline{NY} \longrightarrow \overline{RN} = \overline{OY} \end{array}$$

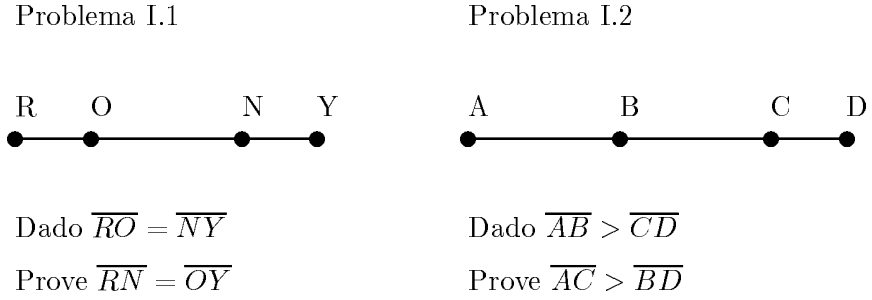


Figura 5: Problemas de Geometria com Segmentos

Quando ao mesmo aprendiz é apresentado o Problema I.2, com base na resolução do Problema I.1, o aprendiz faz o seguinte mapeamento

$$\begin{array}{rcl} R & \longrightarrow & A \\ O & \longrightarrow & B \\ N & \longrightarrow & C \\ Y & \longrightarrow & D \\ = & \longrightarrow & > \end{array}$$

e tenta, por analogia, resolver o problema da seguinte forma

$$\begin{array}{rcl} \overline{AB} & > & \overline{CD} \\ \overline{BC} & > & \overline{BC} \end{array}$$

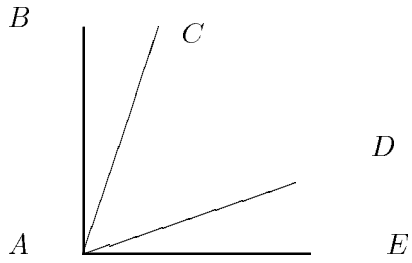
Então ...

Esse raciocínio é logo descartado por absurdo — proveniente da correspondência entre o sinal "=" e o ">". Para chegar na solução é necessário mapear somente "uma" ocorrência do sinal "=" no ">". Assim como é difícil estabelecer as correspondências válidas em um mapeamento por analogia, é mais difícil ainda estabelecer a sua inversão válida.

No segundo exemplo é apresentado ao estudante o mesmo Problema I.1 — Figura 5 — e, logo a seguir, o Problema II ilustrado na Figura 6. Nesse caso, a resolução do problema por analogia faz, diretamente, a correspondência de retas com ângulos, bem como de igualdade com congruência

$$\begin{array}{rcl} \widehat{BAC} & = & \widehat{DAE} \\ \widehat{CAD} & = & \widehat{CAD} \\ \widehat{BAC} + \widehat{CAD} & = & \widehat{CAD} + \widehat{DAE} \longrightarrow \widehat{BAD} = \widehat{CAE} \end{array}$$

Problema II



Dado $\widehat{BAC} = \widehat{DAE}$

Prove $\widehat{BAD} = \widehat{CAE}$

Figura 6: Problema de Geometria com Ângulos

6.3 ANALOGY

Um outro exemplo clássico é o relacionado a questões de analogia geométrica de testes de inteligência, nos quais várias figuras são apresentadas ao aprendiz. Por exemplo — veja Figura 7 — as figuras A, B e C são escolhidas de uma lista de respostas possíveis e a seguinte questão é colocada

A está para B como C está para ?

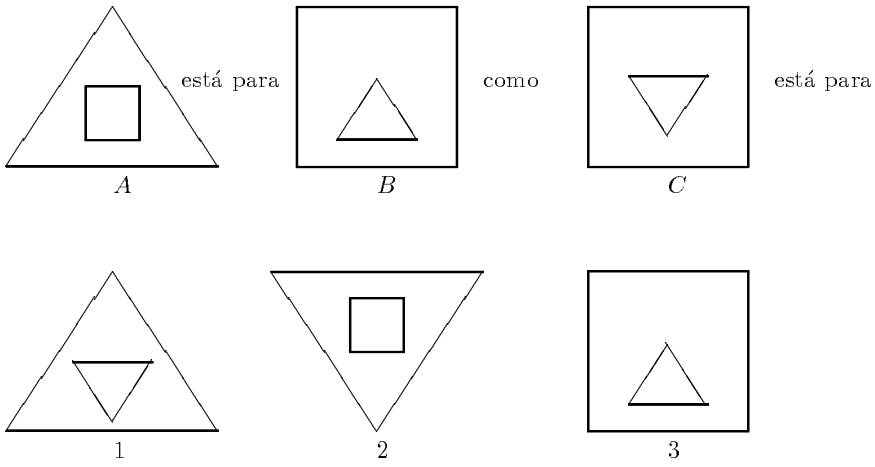


Figura 7: Teste de Inteligência

As possíveis respostas são 1, 2 ou 3. O algoritmo simplificado para solucionar esse problema é o seguinte

Encontrar uma regra que relaciona A com B
Aplicar a regra a C para obter a figura X
Encontrar X (ou semelhante) entre as possíveis respostas.

No problema ilustrado na Figura 7, pode ser observado que a posição dos quadrados e triângulos do tipo $\triangle$ das figuras A e B — dimensionados apropriadamente — está trocada. A resposta óbvia é trocar o quadrado e o triângulo do tipo ∇ na figura C. A figura resultante corresponde à resposta 2.

O programa ANALOGY de Evans, descrito a seguir, implementa na linguagem de programação lógica Prolog[‡] uma solução simples para resolver o problema de analo-

[‡]As referências [Hogger 90, Kowalsky 79] são apropriadas para um estudo sobre programação lógica. Referente à linguagem Prolog, vários livros têm sido publicados, entre eles recomenda-se a leitura de [Bratko 90, Marcus 86, O'Keefe 90, Rowe 88, Sterling 86, Shoham 94]. Diversos programas Prolog para processamento de listas e árvores podem ser encontrados em [Monard 93a, Monard 93b].

gia. Este programa soluciona questões de analogia geométrica de testes de inteligência semelhantes ao descrito anteriormente [Sterling 86].

6.3.1 Descrição da Implementação

A implementação do programa ANALOGY realizada — veja listagem na seção 6.3.3 pg. 26 — resolve o problema de analogia da Figura 7 e foi expandida para resolver também os três problemas ilustrados na Figura 8 pg. 22.

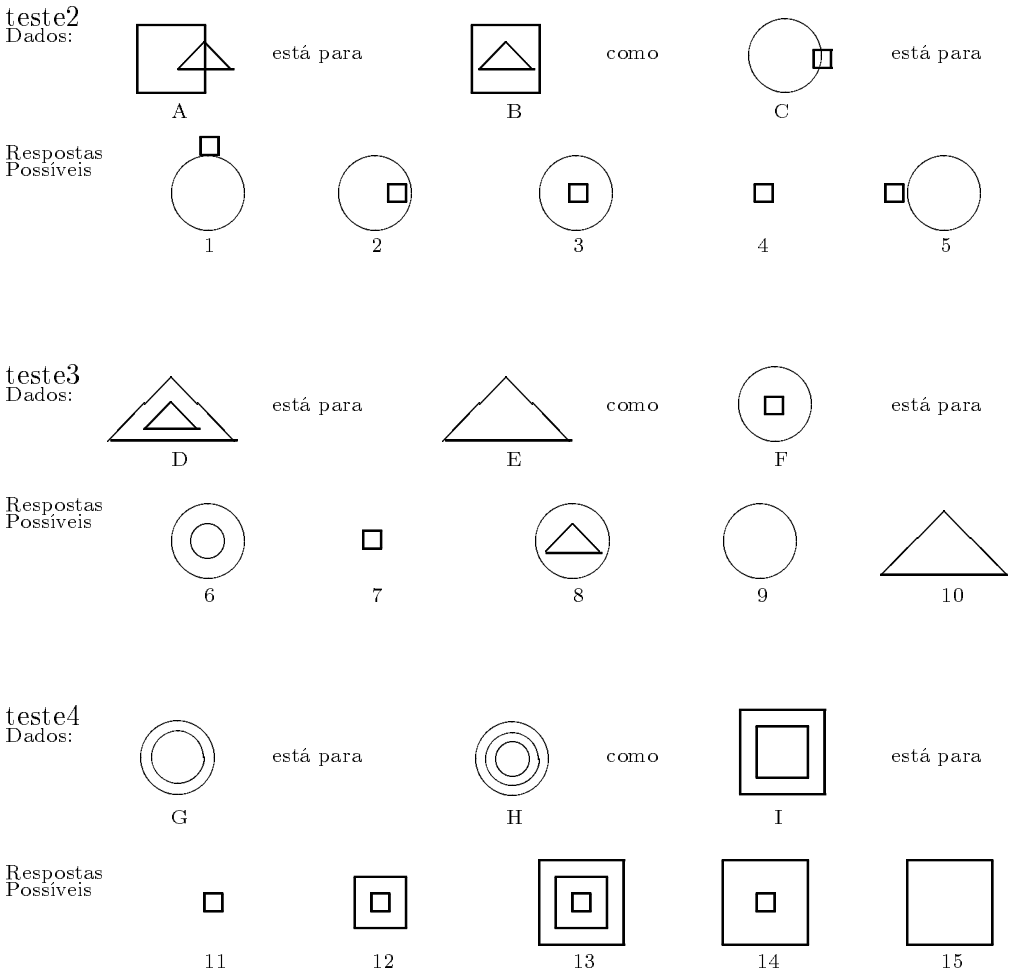


Figura 8: Três Problemas de Analogia

Uma decisão fundamental é como representar as figuras. Nesta implementação, as figuras são representadas como termos Prolog. Por exemplo, a figura A no diagrama da Figura 7 representa um quadrado dentro de um triângulo; ela é representada pelo termo

```
dentro(quadrado, triangulo)
```

O procedimentos principal deste programa é

```
analogia(esta_para(A,B), esta_para(C,X), Respostas):-
      casa(A,B,Regra),
      casa(C,X,Regra),
      membro(X,Respostas).
```

Onde A, B e C são as figuras sobre as quais se pretente realizar as analogias, e X é uma variável livre que irá instanciar com a figura resposta. Esta figura resposta deve ainda pertencer a `Resposta`, pois se trata de um teste de múltipla escolha.

A relação entre as figuras é encontrada pelo procedimento

```
casa(A, B, Operacao)
```

O procedimento `casa` é usado de duas maneiras distintas. Primeiramente, ele produz uma operação de casamento entre as duas figuras. Depois, dadas uma operação e uma figura, ele fornece uma outra figura que obedece a operação de casamento. Esse procedimento, para os exemplos considerados — Figura 7 e 8 — está definido da seguinte forma

```
casa(dentro(Fig1, Fig2), dentro(Fig2, Fig1), inverte).
casa(lado_dir(Fig1, Fig2), dentro(Fig1, Fig2), centraliza).
casa(dentro(Fig1, Fig2), dentro(nada, Fig2), apaga_dentro).
casa(dentro(nada, Fig, Fig), dentro(Fig, Fig, Fig), poe_dentro).
```

Observe que a função de `inverte` é a de trocar as figuras. Finalmente, o procedimento `membro` simplesmente verifica se a figura dada aparece na lista de respostas.

O Aprendizado por Analogia é muito útil na resolução de problemas. Entretanto, para que a analogia possa ser estabelecida diretamente, os problemas precisam ser suficientemente semelhantes. Se isto não ocorrer, esse tipo de aprendizado servirá apenas para fornecer idéias iniciais sobre um conceito ou solução.

6.3.2 Exemplo de Execução

A fim de executar o programa, é necessário fornecer os dados. Eles consistem de

1. a descrição da figura original e
2. a descrição das possíveis respostas.

Nomeando o caso ilustrado na Figura 7 como `teste1` e os casos ilustrados na Figura 8 como `teste2`, `teste3` e `teste4` respectivamente, a descrição das figuras e possíveis respostas para cada um desses casos pode ser realizada através das seguintes cláusulas unitárias

- Dados para `teste1`

```
figuras(teste1, dentro(quadrado, triangulo), dentro(triangulo, quadrado),
                                             dentro(circulo, quadrado)).
respostas(teste1,
     [dentro(circulo, triangulo), dentro(quadrado, circulo),
      dentro(triangulo, quadrado)]).
```

- Dados para `teste2`

```
figuras(teste2, lado_dir(triangulo, quadrado), dentro(triangulo, quadrado),
                                               lado_dir(quadrado, circulo)).
respostas(teste2,
     [acima(quadrado, circulo), interior_dir(quadrado, circulo),
      dentro(quadrado, circulo), dentro(quadrado, nada),
      exterior_esq(quadrado, circulo)]).
```

- Dados para `teste3`

```
figuras(teste3, dentro(triangulo, triangulo), dentro(nada, triangulo),
                                              dentro(quadrado, circulo)).
respostas(teste3,
     [dentro(circulo, circulo), dentro(quadrado, nada),
      dentro(triangulo, circulo), dentro(nada, circulo),
      dentro(nada, triangulo)]).
```

- Dados para `teste4`

```
figuras(teste4, dentro(nada,circulo,circulo), dentro(circulo,circulo,circulo),
                                              dentro(nada, quadrado, quadrado)).
respostas(teste4,
      [dentro(quadrado,nada,nada), dentro(quadrado,quadrado,nada),
      dentro(quadrado, quadrado, quadrado), dentro(quadrado, nada, quadrado),
      dentro(nada, nada, quadrado)]).
```

O procedimento `teste(Nome, X)` definido como

```
teste(Nome, X):-  figuras(Nome, A, B, C),
                  respostas(Nome, Respostas),
                  analogia(esta_para(A,B), esta_para(C,X), Respostas).
```

pode ser utilizado para executar um dos possíveis casos se o seu primeiro argumento estiver instanciado com o nome do caso específico, bem como para executar todos eles se o seu primeiro argumento for uma variável livre. Segue o exemplo de execução deste último caso.

```
?teste(Nome, X).

Nome= teste1
X= dentro(quadrado,circulo) ->;

Nome= teste2
X= dentro(quadrado,circulo) ->;

Nome= teste3
X= dentro(nada,circulo) ->;

Nome= teste4
X= dentro(quadrado,quadrado,quadrado) ->;
no
```

6.3.3 Listagem do Programa

```
%////////////////////////////////////////////////////////////////////////%
%/       Para executar o programa:                                      %
%/            ?teste(Nome, X).                                          %
%/   onde Nome pode ser teste1, teste2, teste3 ou teste4 e X e' a resposta%
%////////////////////////////////////////////////////////////////////////%

%************* PROGRAMA PARA SOLUCIONAR ANALOGIAS GEOMETRICAS ************%
    % A esta para B assim como C esta para qual das possiveis respostas?
analogia(esta_para(A,B), esta_para(C,X), Respostas):-
      casa(A,B,Regra),     % encontre uma regra que relacione A com B
      casa(C,X,Regra),     % aplique a mesma regra para C para dar a fig. X
      membro(X,Respostas). % encontre X ou seu equivalente mais proximo
                           % entre as respostas

   % Definicao de algumas regras
casa(dentro(Fig1, Fig2), dentro(Fig2, Fig1), inverte).
casa(lado_dir(Fig1, Fig2), dentro(Fig1, Fig2), centraliza).
casa(dentro(Fig1, Fig2), dentro(nada, Fig2), apaga_dentro).
casa(dentro(nada, Fig, Fig), dentro(Fig, Fig, Fig), poe_dentro).

   % Verifica se um Elemento e' membro de uma lista
membro(Elemento ,[Elemento|_]).
membro(Elemento,[_|Cauda]):- membro(Elemento, Cauda).

%***************************** TESTE E DADOS ****************************%
teste(Nome, X):-  figuras(Nome, A, B, C),
                  respostas(Nome, Respostas),
                  analogia(esta_para(A,B), esta_para(C,X), Respostas).

   % A= quadrado dentro do triangulo
   % B= triangulo dentro do quadrado
   % C= circulo dentro do quadrado
figuras(teste1, dentro(quadrado, triangulo), dentro(triangulo, quadrado),
                                             dentro(circulo, quadrado)).

   % A= triangulo no lado direito do quadrado
   % B= triangulo dentro do quadrado
   % C= quadrado no lado direito do circulo
figuras(teste2, lado_dir(triangulo, quadrado), dentro(triangulo, quadrado),
                                               lado_dir(quadrado, circulo)).

   % A= triangulo dentro do triangulo
   % B= nada dentro do triangulo
   % C= quadrado dentro do circulo
figuras(teste3, dentro(triangulo, triangulo), dentro(nada, triangulo),
                                              dentro(quadrado, circulo)).

    % A= nada dentro do circulo que esta dentro de outro circulo
    % B= circulo dentro do circulo dentro do circulo
    % C= nada dentro do quadrado que esta dentro de outro quadrado
figuras(teste4, dentro(nada,circulo,circulo), dentro(circulo,circulo,circulo),
                                              dentro(nada, quadrado, quadrado)).
```

```
                % Respostas possiveis para cada teste


respostas(teste1,
     [dentro(circulo, triangulo),        % circulo dentro do triangulo
      dentro(quadrado, circulo),         % quadrado dentro do circulo
      dentro(triangulo, quadrado)]).     % triangulo dentro do quadrado

respostas(teste2,
     [acima(quadrado, circulo),          % quadrado acima do circulo
      interior_dir(quadrado, circulo),   % quadrado no interior a direita
                                         % do circulo
      dentro(quadrado, circulo),         % quadrado dentro do circulo
      dentro(quadrado, nada),            % quadrado dentro de nada
      exterior_esq(quadrado, circulo)]). % quadrado fora do circulo,
                                         % a esquerda

respostas(teste3,
     [dentro(circulo, circulo),      % circulo dentro de circulo
      dentro(quadrado, nada),        % quadrado dentro de nada
      dentro(triangulo, circulo),    % triangulo dentro de circulo
      dentro(nada, circulo),         % nada dentro de circulo
      dentro(nada, triangulo)]).     % nada dentro de triangulo

respostas(teste4,
     [dentro(quadrado,nada,nada),           % quadrado dentro de nada
      dentro(quadrado,quadrado,nada),       % quadrado dentro de quadrado
      dentro(quadrado, quadrado, quadrado), % quadrado dentro de quadrado
                                            % dentro de quadrado
      dentro(quadrado, nada, quadrado),,    % quadrado dentro de nada
                                            % dentro de quadrado
      dentro(nada, nada, quadrado)]).       % nada dentro de nada dentro
                                            % de quadrado
```

7 Aprendizado por Indução

A inferência indutiva é um dos principais meios de criar novos conhecimentos e prever eventos futuros. Como já foi mencionado, o processo para usar observações a fim de descobrir regras e procedimentos é denominado indução. O processo de indução é indispensável na obtenção de novos conhecimentos pelo ser humano.

Foi através de induções que Kepler descobriu as leis do movimento planetário, que Mendel descobriu as leis da genética e que Arquimedes descobriu o princípio da alavanca. Pode-se ousar em afirmar que a indução é o recurso mais utilizado pelos seres humanos para obter novos conhecimentos. Apesar disto, esse recurso deve ser utilizado com os devidos cuidados pois, se o número de observações for insuficiente ou se os dados relevantes forem mal escolhidos, as regras obtidas podem ser de pouco ou nenhum valor. Para exemplificar esse fato, basta lembrar a história de um homem que ia a uma cidadezinha e encontrou um bêbado. Não deu muita importância ao fato, até deparar-se com outro bêbado. Nesse momento, concluiu sem pestanejar *Só dá bêbado nesta cidade!* Esse é um caso de indução feita com um número insuficiente de observações.

Há também a anedota do cientista que estava estudando a audição das aranhas. A aranha está parada e ele dá um grito. A aranha foge. Ele então lhe arranca duas pernas e dá um outro grito. A aranha foge mas não tão depressa. Ele arranca as outras pernas e dá um outro grito. A aranha não se move. Ele conclui, então, que *As aranhas usam as pernas para ouvir.* A conclusão dele foi errada porque ele escolheu mal os dados relevantes referentes à audição nas aranhas.

Existem duas formas de Aprendizado por Indução

1. *Aprendizado por Exemplos* e
2. *Aprendizado por Observação e Descoberta.*

No *Aprendizado por Exemplos*, o aprendiz induz a descrição de um conceito formulando uma regra geral a partir dos exemplos e dos contra-exemplos fornecidos pelo professor ou pelo ambiente. O professor já tem o conhecimento do conceito e, assim, ele pode ajudar o aprendiz selecionando exemplos relevantes para aprender um determinado conceito. A tarefa do aprendiz é determinar a descrição geral de um conceito, analisando exemplos individuais a ele fornecidos. Essa estratégia também é conhecida como aprendizado supervisionado.

No *Aprendizado por Observação e Descoberta*, o aprendiz analisa entidades fornecidas ou observadas e tenta determinar se alguns subconjuntos dessas entidades podem ser agrupados em certas classes (i.e. conceitos) de maneira útil. Como não há um professor que já tenha o conhecimento do conceito para fornecer exemplos significativos ao conceito a ser aprendido, essa estratégia é também chamada de aprendizado não supervisionado.

Na seção 7.2 pg. 30 Aprendizado por Exemplos é discutido com mais detalhes.

7.1 Linguagem de Descrição

Qualquer que seja o tipo de aprendizado, é necessário uma linguagem para descrever objetos (ou possíveis eventos) e uma linguagem para descrever conceitos. Em geral, é possível distinguir dois tipos de descrições

1. descrições estruturais
2. descrições de atributos

Em uma descrição estrutural, um objeto é descrito em termos de seus componentes e a relação entre eles. Por exemplo, a descrição estrutural de um arco pode ser

> *Um arco consiste de três componentes — dois postes e um lintel*[§] *— tal que, cada um deles é um bloco; os dois postes suportam o lintel; os postes estão na vertical, são paralelos, não se tocam e o lintel está na horizontal.*

Em uma descrição de atributos, um objeto é descrito em termos de suas características globais como um vetor de valores de atributos. Uma descrição de atributos de um arco pode ser

> *Seu tamanho é 8cm, sua altura é 5cm e sua cor é amarelo.*

Alguns formalismos freqüentemente usados em AM para descrever objetos e conceitos são

- vetores de atributos para representar objetos
- regras *if-then* para representar conceitos
- árvores de decisão para representar conceitos
- redes semânticas
- lógica de predicados

Linguagens específicas para descrever objetos e conceitos que podem ser usadas em programas de aprendizado são semelhantes àquelas que podem ser usadas para representação de conhecimento em geral.

[§] palavra usada em Arquitetura que denomina a parte superior de um arco; vergo

7.2 Aprendizado Indutivo por Exemplos

A inferência indutiva e a estrutura básica para guiar a busca em aprendizado indutivo são descritas em [Shaw 90] da seguinte forma

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

Como já mencionado, Aprendizado Indutivo é o processo de inferência indutiva através de fatos fornecidos por um professor ou ambiente. Um tipo especial de aprendizado indutivo é o *aprendizado de conceitos através de exemplos*, cuja tarefa é induzir descrições gerais de conceitos através de instâncias específicas desses conceitos [Michalski 83]. O problema de aprendizado de conceitos através de exemplos pode ser formalizado da seguinte forma [Bratko 89]

> *Seja $\mathcal{U}$ o conjunto universal dos objetos, isto é, todos os objetos que o aprendiz pode encontrar. Não existe limite, a princípio, no tamanho de $\mathcal{U}$. Um conceito $\mathcal{C}$ pode ser formalizado como um subconjunto de objetos em $\mathcal{U}$*
>
> $$\mathcal{C} \subset \mathcal{U}$$
>
> *aprender um conceito $\mathcal{C}$ significa aprender a reconhecer objetos em $\mathcal{C}$. Ou seja, uma vez que o conceito $\mathcal{C}$ é aprendido, para qualquer objeto $\mathcal{X}$ em $\mathcal{U}$, o sistema é capaz de reconhecer se $\mathcal{X}$ está em $\mathcal{C}$.*

7.3 Aprendizado Incremental e Não Incremental

Os algoritmos de aprendizado indutivo podem ser classificados de duas formas segundo o modo em que os exemplos são apresentados

1. *incremental*
2. *não incremental.*

Um algoritmo não incremental necessita de que todos os exemplos de treinamento, simultaneamente, estejam disponíveis para que seja induzido um conceito. É vantajoso usar esses algoritmos para problemas de aprendizado onde todos os exemplos estão disponíveis e, provavelmente, não irão ocorrer mudanças.

Um algoritmo incremental revê a definição do conceito corrente, se necessário, em resposta a cada nova instância de treinamento observada. Os exemplos observados são considerados um a um pelo sistema, isto é, o sistema considera o primeiro exemplo e, de acordo com este exemplo constrói uma determinada hipótese; a seguir considera um segundo exemplo, que pode ou não modificar a primeira hipótese, baseando-se em como esta classifica o segundo exemplo. À medida que mais exemplos são apresentados, o sistema continua modificando o conceito.

Por exemplo, o programa ARCHES [Winston 75] — descrito na seção 8.1 pg. 36 — utiliza processamento incremental para aprender um conceito. Ele aprende descrições estruturais de conceitos no mundo dos blocos (o conceito de um arco) através de exemplos positivos e negativos fornecidos por um "professor". O programa representa exemplos e conceitos na forma de uma rede semântica. A cada passo do aprendizado é mantida uma hipótese corrente do conceito. A hipótese inicial assume as descrições do primeiro exemplo que é, necessariamente, positivo. Se o próximo exemplo for positivo e não satisfizer a descrição do conceito corrente, o programa então generaliza a descrição para que inclua esse exemplo. Se o próximo exemplo for negativo e satisfizer a descrição corrente, então a descrição é especializada para que exclua o exemplo do conceito. O programa processa todos os exemplos restantes, individualmente, até que o conceito englobe todos os exemplos positivos e nenhum negativo.

Uma das vantagens de usar um algoritmo incremental é que o conhecimento pode ser rapidamente atualizado a cada nova observação. É mais eficiente revisar uma hipótese existente do que gerar uma hipótese cada vez que uma nova instância é observada [Utgoff 89].

7.4 O Problema do Aprendizado por Exemplos

Aprender um conceito pode ser visto como um processo de busca. Para cada classe de objetos, o aprendiz "navega" através de um espaço de descrições de conceitos até encontrar um conceito apropriado [Mitchell 82]. No caso de aprendizado por exemplos, a meta é encontrar a descrição de um conceito sendo que a fonte de informação para o aprendizado consiste de exemplos.

Um exemplo para aprender um conceito $\mathcal{C}$ é um par

$$(objeto, classe)$$

onde *objeto* é uma descrição do objeto e *classe* é "+" ou "–". Tal que

> *Se o objeto pertence a* $\mathcal{C}$
> *então classe = "+"*
> *caso contrário, classe = "–"*

É dito que o exemplo é positivo ($\mathcal{E}^+$) ou negativo ($\mathcal{E}^-$). O problema de aprendizado de um conceito $\mathcal{C}$ através de exemplos pode ser formulado da seguinte forma [Bratko 89]

> *Dado um conjunto S de exemplos, encontrar uma fórmula ou hipótese $\mathcal{H}$ expressa na linguagem de descrição do conceito, tal que*
>
> *Para todo objeto X*
>
> 1. *se X é um exemplo positivo no conjunto de treinamento S, então X "casa" com $\mathcal{H}$*
> 2. *se X é um exemplo negativo em S, então X não "casa" com $\mathcal{H}$*

Como resultado do aprendizado, $\mathcal{H}$ é a "compreensão do sistema" do conceito $\mathcal{C}$ obtido através de exemplos. De acordo com essa definição, $\mathcal{H}$ e $\mathcal{C}$ concordam em todos os exemplos dos objetos em S. Em outras palavras, $\mathcal{H}$ é completa (cobre todos os exemplos positivos) e consistente (não cobre nenhum exemplo negativo). Não existe, entretanto, nenhuma garantia que $\mathcal{H}$ corresponderá também a $\mathcal{C}$ em outros objetos. A principal meta do aprendizado indutivo é aprender a classificar objetos não vistos[¶] (aqueles contidos em $\mathcal{U}$ mas não contidos em S) em relação a $\mathcal{C}$. Assim, um critério importante para o sucesso do aprendizado é a precisão da classificação de $\mathcal{H}$ em objetos não vistos.

A hipótese $\mathcal{H}$ pode não classificar corretamente alguns objetos em S. Isto ocorre quando os dados de aprendizado contêm erros ou outros tipos de incertezas. Essas propriedades dos dados são usualmente denominadas ruído[‖]. Ruído é típico de alguns domínios de

[¶] objetos não utilizados para o aprendizado do conceito
[‖] noise

aplicações como medicina. Desde que os dados de aprendizado não são confiáveis, o "casamento" exato entre a hipótese $\mathcal{H}$ e o conceito $\mathcal{C}$ no conjunto de aprendizado S não garante a correção e pode, entretanto, ser abandonada em favor de alguma outra vantagem, tal como simplicidade conceitual de $\mathcal{H}$.

A Figura 9 mostra graficamente os conceitos de completude e consistência de uma hipótese. Esta figura utiliza a função cobre, a qual pode ser definida da seguinte forma

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

no qual $\mathcal{H}$ é a hipótese e e um exemplo positivo ou negativo.

A função cobre pode ser redefinida para conjunto de exemplos $S' \subseteq S$ cobertos por $\mathcal{H}$ da seguinte forma

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

Dependendo de como a hipótese cobre os exemplos positivos e negativos, quatro situações podem acontecer

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

7.5 Critérios de Sucesso

Há vários critérios para medir o sucesso de um sistema de aprendizado. Os critérios mais usuais são

1. *Precisão da Classificação*
2. *Transparência da Descrição do Conceito Induzido*
3. *Complexidade Computacional*

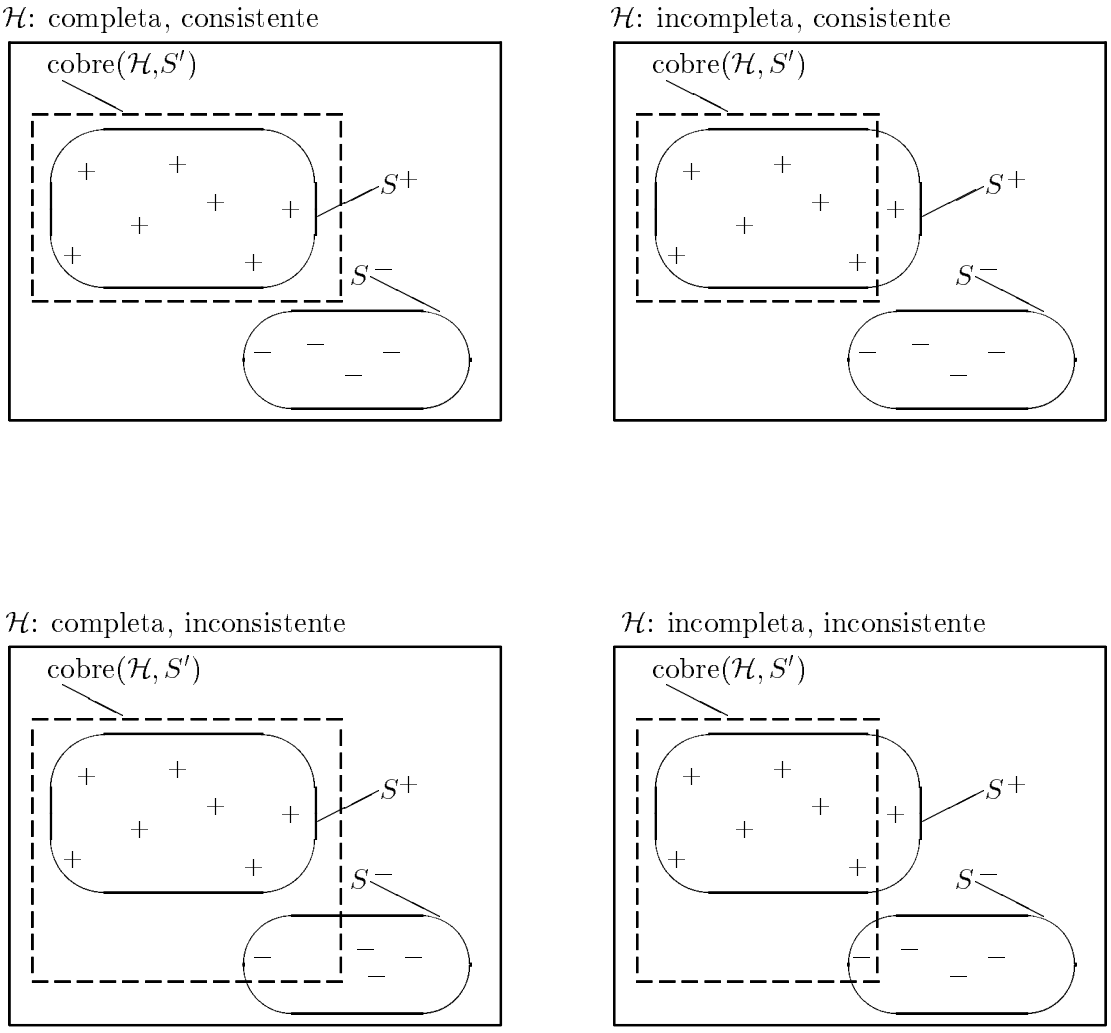


Figura 9: Completeza e Consistência de uma Hipótese [Lavrač 94]

1. *Precisão da Classificação.* É geralmente definida como a porcentagem de objetos corretamente classificados pela hipótese aprendida $\mathcal{H}$. É possível distinguir dois tipos de precisão da classificação

   - Precisão em objetos não vistos — aqueles que não estão contidos no conjunto de treinamento S.
   - Precisão nos objetos em S — esse tipo de precisão é interessante somente se o algoritmo de aprendizado relaxar as condições para $\mathcal{H}$ ser completa e consistente.

2. *Transparência da Descrição do Conceito Induzido $\mathcal{H}$.* É importante que a descrição gerada seja compreensível por um ser humano para que possa mostrar ao usuário alguma coisa interessante sobre o domínio da aplicação. Tal descrição pode então ser usada diretamente pelo usuário sem ajuda da máquina, como uma intensificação ao seu próprio conhecimento. Esse critério é também muito importante quando descrições induzidas são usadas em um Sistema Especialista cujo comportamento tem que ser transparente, tal como explicar como o sistema chegou a uma conclusão.

3. *Complexidade Computacional.* Está relacionada com os recursos computacionais necessários (tempo e espaço) para realizar o aprendizado. É possível distinguir dois tipos de complexidade

   - Complexidade de Geração: recursos necessários para induzir uma descrição do conceito através de exemplos.
   - Complexidade de Execução: complexidade em classificar um objeto usando a hipótese induzida.

8 Aprendizado Indutivo por Exemplos

Nesta seção são discutidos vários algoritmos e suas implementações na linguagem de programação lógica Prolog, para Aprendizado por Exemplos.

8.1 O Sistema ARCHES

Um exemplo de Aprendizado por Exemplos incremental é o programa de Winston que aprende conceitos estruturais. O programa ARCHES aprende o conceito de um arco através de exemplos e contra-exemplos fornecidos por um professor como mostrado na Figura 10.

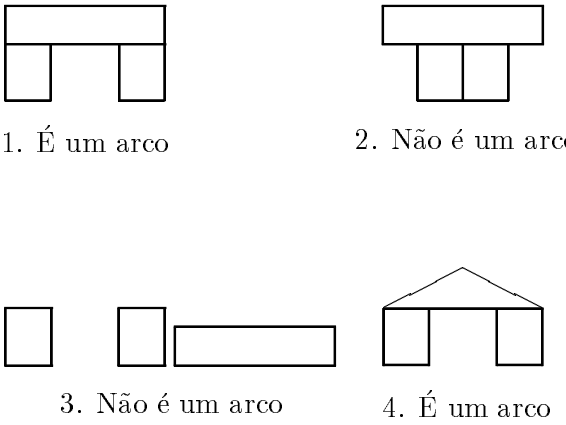


Figura 10: Seqüência de Exemplos e Contra-exemplos para o Aprendizado do Conceito de Arco

Os exemplos fornecidos pelo professor são processados seqüencialmente pelo aprendiz que, a cada novo exemplo, "atualiza" o conceito que está aprendendo. No caso específico considerado, após o aprendiz ter processado os quatro exemplos ilustrados na Figura 10, o conceito de arco por ele aprendido seria, informalmente, o seguinte

1. Um arco consiste de três partes, dois postes e um lintel
2. Os dois postes têm forma retangular enquanto que o lintel pode ser uma classe de polígono. Isto pode ser concluído dos exemplos 1 e 4 — Figura 10
3. Os dois postes não podem se tocar. Isto pode ser concluído do exemplo negativo 2 — Figura 10
4. Os dois postes devem suportar o lintel. Isto pode ser concluído do exemplo negativo 3 — Figura 10

Em geral, quando um conceito é aprendido através do processamento seqüencial de exemplos — aprendizado incremental —, o processo de aprendizado procede através de

uma seqüência de hipóteses, H_1, H_2, ... etc., sobre o conceito que está sendo aprendido. Quando um exemplo é processado, a hipótese corrente é atualizada resultando na próxima hipótese. O algoritmo seguinte mostra este processo [Bratko 90]

> Para aprender um conceito C de uma dada seqüência de exemplos E_1, E_2,..., E_n, onde o primeiro exemplo E_1 deve ser obrigatoriamente um exemplo positivo de C, faça
>
> 1. Adote E_1 como hipótese inicial H_1 sobre C.
> 2. Processe todos os exemplos restantes da seguinte forma:
>   Para cada E_i (i= 2,3,...) faça
>   (a) Compare a hipótese corrente H_{i-1} com E_i; seja D o resultado dessa comparação (diferença entre H_{i-1} e E_i)
>   (b) Atualize H_{i-1} de acordo com D, verificando também se E_i é um exemplo positivo ou negativo. O resultado desta atualização é uma hipótese refinada H_i sobre C.
>
> O resultado final deste procedimento é H_n, que representa o entendimento do sistema sobre o conceito C.

As duas operações principais desse algoritmo são a comparação e a atualização da hipótese corrente que, para serem implementadas, precisam ser refinadas. Deve ser observado que ambas as operações não são fáceis de implementar e variam bastante entre os diferentes sistemas de AM que as utilizam. A fim de ilustrar essas dificuldades, serão examinados com mais detalhes os exemplos e contra-exemplos ilustrados na Figura 10.

Referente à representação, o programa ARCHES utiliza redes semânticas para representar tanto os exemplos como as descrições do conceito. Redes semânticas são essencialmente grafos nos quais os nós correspondem a entidades, e as ligações indicam as relações entre essas entidades. A Figura 11 pg. 38 ilustra vários exemplos de redes semânticas.

O primeiro exemplo, representado por uma rede semântica, torna-se a hipótese corrente do que um arco é — H_1 na Figura 11.

O segundo exemplo — E_2 na Figura 11 — é um contra-exemplo de um arco. Como as duas redes são muito parecidas, é fácil estabelecer a correspondência entre os nós e as ligações de H_1 e E_2. A diferença D entre H_1 e E_2 é a relação extra, **encosta**, em E_2. Como essa é a única diferença, o sistema conclui que essa relação extra é a razão de E_2 não ser um arco. O sistema atualiza então a hipótese corrente H_1 aplicando o seguinte princípio de aprendizado heurístico

> *Se*
> o exemplo é negativo e
> o exemplo contém a relação R que não está na hipótese corrente H
> *Então*

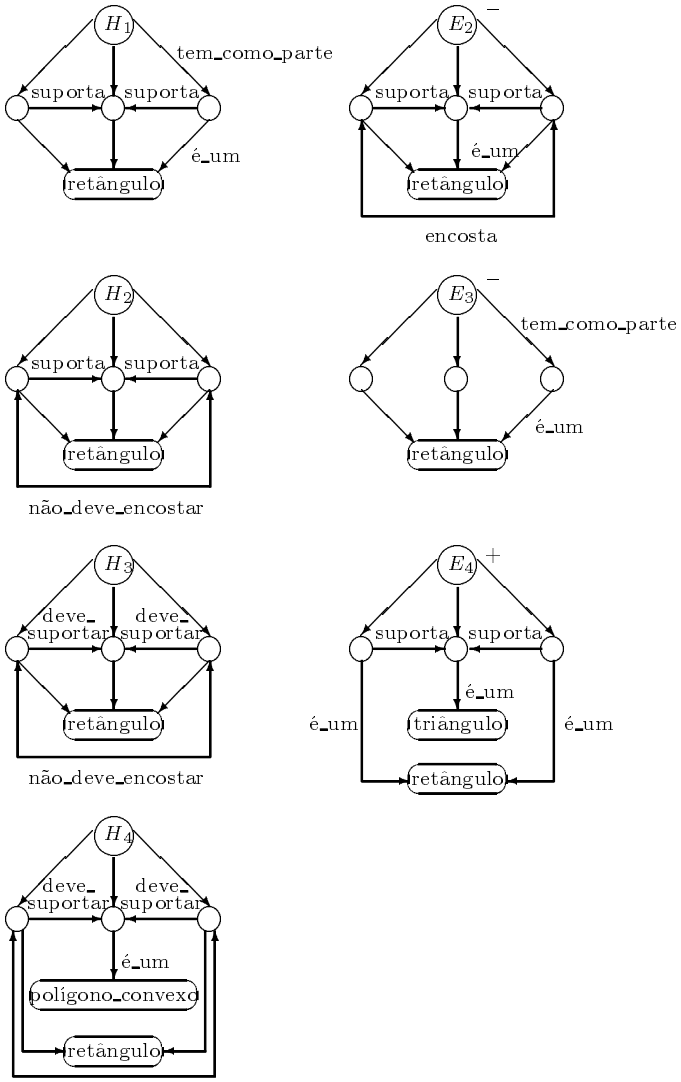


Figura 11: Seqüência de Hipóteses sobre um Arco

proiba R em H (adicione **não_deve_**R em H)

Aplicando essa regra a H_1, obtém-se uma nova hipótese H_2. Essa nova hipótese H_2 possui uma ligação extra **não_deve_encostar** — veja Figura 11. Neste caso, diz-se que H_2 é uma hipótese *mais específica* que H_1.

O próximo exemplo negativo da Figura 10 pg. 36 é representado pela rede semântica E_3 na Figura 11. Comparando-o com a hipótese corrente H_2, observam-se duas diferenças: as duas ligações **suporta**, presentes em H_2, não estão presentes em E_3. Agora, é necessário que o sistema escolha uma das seguintes explicações

1. E_3 não é um arco porque a ligação **suporta** à esquerda está faltando, ou

2. E_3 não é um arco porque a ligação **suporta** à direita está faltando, ou

3. E_3 não é um arco porque ambas as ligações **suporta** estão faltando.

Portanto, o aprendiz precisa escolher qual delas utilizar para atualizar a hipótese corrente. Supondo que a mentalidade do aprendiz é para mudanças mais radicais, a explicação escolhida é a 3. Neste caso, o aprendiz assumirá que ambas as ligações **suporta** são necessárias e converterá ambas as ligações **suporta** em H_2 em ligações **deve_suportar** na nova hipótese H_3 — Figura 11. A situação de falta de ligações pode ser manipulada pela seguinte regra condição-ação que é outra heurística utilizada em aprendizado

Se
o exemplo é negativo e
o exemplo não contém a relação R que está presente
na hipótese corrente H
Então
R deve ser uma condição necessária na nova hipótese.
(adicione **deve_**R em H).

Neste ponto, é bom observar que a mentalidade do aprendiz pode ser modelada, tanto pela eleição das explicações referentes ás diferenças entre hipóteses, quanto pela eleição do tipo de regra condição-ação a ser utilizada. Variando essas regras, o estilo de aprendizado pode variar entre dois extremos. Em um deles tem-se um aprendizado cauteloso, prudente ou precavido, no outro extremo tem-se um aprendizado tipo "radical".

Voltando ao caso sobre consideração, o último exemplo E_4 é novamente positivo. A comparação de E_4 com H_3 mostra uma diferença: a parte de cima é um triângulo em E_4 e um retângulo em H_3. O aprendiz deve trocar a correspondente ligação **é_um** na hipótese, de retângulo para uma nova classe de objetos **retângulo_ou_triângulo**. Uma forma alternativa e mais comum de reagir em programas de aprendizado é baseada numa hierarquia pré-definida de conceitos. Supondo que o aprendiz tenha como conhecimento de fundo, no domínio tratado, uma taxonomia de conceitos e observe que retângulo e triângulo são ambos do tipo **polígono_convexo**, ele deve então atualizar a hipótese corrente para obter H_4 — Figura 11.

Note que o processamento desse exemplo positivo resultou em uma nova hipótese mais geral (**polígono_convexo** em vez de **retângulo**). Neste caso, é dito que a hipótese corrente foi *generalizada*.

A nova hipótese agora permite que a parte de cima seja um trapézio, embora nenhum exemplo de um arco com um trapézio foi mostrado ao aprendiz. Se for mostrado ao sistema o desenho da Figura 12, ele será classificado pelo sistema como arco, pois sua representação em rede semântica satisfaz completamente o conceito de arco aprendido pelo sistema — a hipótese H_4.

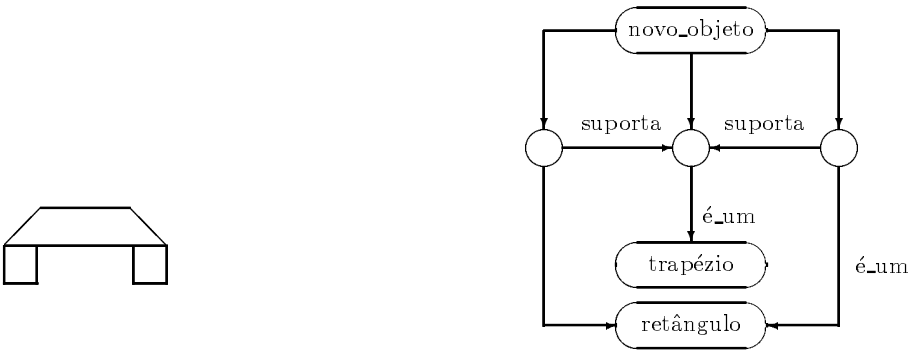


Figura 12: Um Novo Objeto e sua Representação

8.1.1 Descrição da Implementação

A implementação simplificada do programa ARCHES de Winston, aqui apresentada, utiliza a seguinte representação Prolog para objetos e exemplos [Bratko 90]

- `Objeto = objeto(ListaDePartes, ListaDeRelacoes)`
- `Exemplo = + Objeto` (exemplo positivo)
- `Exemplo = - Objeto` (exemplo negativo)

A descrição de conceitos é mais complexa. Eles também possuem partes e relações, e as relações são classificadas em 3 categorias

- relações **deve** — aquelas que devem estar presentes,
- relações **desejáveis** e
- relações **não deve** — aquelas que não devem estar presentes

A representação utilizada no programa, apresentado na seção 8.1 pg. 36, é a seguinte

```
DescConc = conceito(Partes, Deve, Relacoes, NaoDeve)
```

onde

- `Deve` é a lista das relações que devem estar presentes,
- `Relacoes` é a lista das relações desejáveis e
- `NaoDeve` é a lista das relações que não devem estar presentes.

O predicado mais complexo no programa implementado é

```
casa(Objeto, Conceito, Diferenca)
```

definido de tal modo que ele é verdadeiro se

1. Todas as relações `deve` em `Conceito` aparecem em `Objeto`.
2. Nenhuma relação `nao_deve` em `Conceito` aparece em `Objeto`.
3. `Objeto` e `Conceito` têm igual número de partes em suas listas de partes
4. `Diferenca` contém as seguintes informações

- as relações que estão faltando — aquelas relações desejáveis que não estão em `Objeto`
- as relações extras — aquelas relações em `Objeto` que não aparecem como desejáveis em `Conceito`.

`Diferenca` é, portanto, um par de listas

```
Diferenca = Faltando + Extras
```

O procedimento `casa` deve minimizar essas diferenças. O número de diferenças depende de como são escolhidas as partes em `Objeto` para corresponder às partes em `Conceito`. Por isso, é importante encontrar uma boa correspondência; o procedimento `casa` é responsável por encontrar essa correspondência. Para minimizar as diferenças representadas pelo par de listas `Faltando + Extras`, o procedimento `casa` procura por uma correspondência entre as partes do objeto e a descrição do conceito de tal maneira que faça com que as listas `Faltando` e `Extra` fiquem tão reduzidas quanto possível.

Tanto os objetos quanto os conceitos são representados por redes semânticas. O processo de casamento precisa estabelecer a correspondência entre os nós nos dois grafos; em particular, ele precisa encontrar que parte do objeto corresponde a qual parte do conceito. Na representação utilizada, há uma lista de partes para o objeto e uma para a descrição do conceito. O processo de casamento deve associar a cada elemento em uma lista um elemento na outra lista. Essa associação é simples de implementar em Prolog através de instanciação de variáveis. Portanto, uma lista contém apenas variáveis enquanto a outra lista contém apenas constantes. Por exemplo, considerando que as duas listas de partes são

```
L1 = [parte1, parte2, parte3]        L2 = [A, B, C]
```

Os possíveis pares correspondem às permutações da lista `L1` de partes podem ser realizadas através das seguintes instanciações

```
A = parte1, B = parte2, C = parte3
A = parte1, B = parte3, C = parte2
A = parte2, B = parte1, C = parte3
                .....
```

Em princípio, não importa qual das duas listas possui variáveis e qual possui constantes. Como neste caso uma lista pertence à descrição do conceito e a outra a de um objeto, e uma descrição do conceito é uma definição geral de um objeto, seria mais natural ter variáveis na descrição do conceito e constantes na descrição do objeto. Entretanto, esta alternativa é menos prática pelos motivos expostos a seguir.

Após o casamento, a descrição do conceito é atualizada, produzindo uma nova descrição do conceito. Se a descrição do conceito contém variáveis, estas se tornariam constantes durante o casamento, e as constantes teriam que se tornar variáveis novamente quando

a nova descrição do conceito fosse construída. É, portanto, mais prático ter variáveis na descrição do objeto e não na descrição do conceito.

A seguir são apresentadas as representações correspondentes a alguns exemplos concretos. O primeiro exemplo na Figura 10 pg. 36 é representado como

```
+objeto([A, B, C],
             [suporta(A, C), suporta(B, C),
              e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])
```

onde `A` e `B` são os dois postes e `C` é o lintel. A descrição final induzida do conceito de arco — H_4 na Figura 11 pg. 38 — é

```
conceito([parte1, parte2, parte3],
              [suporta(parte1, parte3), suporta(parte2, parte3)],
              [e_um(parte1, retangulo), e_um(parte2, retangulo),
               e_um(parte3, poligono_convexo)],
              [encosta(parte1, parte2)])
```

O conceito tem três partes. No caso de estar procurando pelo melhor casamento entre um conceito e um objeto, o procedimento `casa` considera todas as permutações das partes. Isto é realizado pelo procedimento `lista_dif`.

Regras para atualizar a descrição do conceito corrente de acordo com as diferenças são implementadas através da relação

```
atualiza(TipoExemplo, Diferenca, DescAtual, DescNova)
```

onde `Diferenca = RelacoesFaltando + RelacoesExtras`. Por exemplo, uma regra de atualização heurística é

Se
um exemplo negativo contém uma relação extra
Então
esta relação deve ser proibida no conceito
(isto é, esta relação é incluída entre as `nao_deve`).

Esta regra está implementada da seguinte maneira

```
atualiza(negativo, _ + [RelacaoExtra],
              conceito(Partes, Deve, Rels, NaoDeve),
              conceito(Partes, Deve, Rels, [RelacaoExtra | NaoDeve])).
```

Finalmente, o nível mais alto do programa está implementado através do seguinte predicado `aprende`

```
aprende([PrimeiroExemplo | Exemplos], DescConceito):-
         inicializa(PrimeiroExemplo, HipoteseInicial),
         processa_exemplos(HipoteseInicial, Exemplos, DescConceito).
```

O primeiro exemplo fornecido, que deve ser sempre positivo, é usado para construir uma hipótese inicial sobre o conceito a ser aprendido. A descrição final do conceito é obtida processando os outros exemplos. O predicado responsável pelo processamento dos outros exemplos está definido da seguinte forma

```
processa_exemplos(DescAtual, [Exemplo | Exemplos], DescFinal):-
       tipo_objeto(Exemplo, Objeto, Tipo),
       casa(Objeto, DescAtual, Diferenca),
       atualiza(Tipo, Diferenca, DescAtual, DescNova),
       processa_exemplos(DescNova, Exemplos, DescFinal).
```

onde cada exemplo é comparado com a descrição atual do conceito. A descrição corrente é então atualizada de acordo com as diferenças entre a descrição corrente e o exemplo, dependendo também se o exemplo é positivo ou negativo.

A abordagem utilizada para o aprendizado de conceitos estruturais tem alguns problemas, o principal deles é que o instrutor deve guiar o programa de aprendizado através de uma seqüência de exemplos cuidadosamente selecionada.

8.1.2 Exemplo de Execução

Para executar o programa é necessário fornecer o conjunto de exemplos para o aprendizado do conceito, bem como a taxonomia correspondente aos objetos considerados através da relação `ako`. O conjunto de exemplos utilizados para ilustrar a execução deste programa utiliza figuras geométricas. Assim, a relação `ako` está definida da seguinte forma

```
ako(figura, poligono).
ako(figura, circulo).
ako(poligono, poligono_convexo).
ako(poligono, poligono_concavo).
ako(poligono_convexo, triangulo).
ako(poligono_convexo, retangulo).
ako(poligono_convexo, trapezio).
ako(poligono_convexo, hexagono).
```

O conjunto de exemplos é o ilustrado na Figura 10 pg. 36 para aprender o conceito de arco. Esses exemplos estão definidos pelo predicado `exemplo` da seguinte forma

```
exemplo(+objeto([A, B, C],
                [suporta(A, C), suporta(B, C),
                e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo(-objeto([A, B, C],
                [suporta(A, C), suporta(B, C), encosta(A, B),
                e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo(-objeto([A, B, C],
                [e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo(+objeto([A, B, C],
                [suporta(A, C), suporta(B, C),
                e_um(A, retangulo), e_um(B, retangulo), e_um(C, triangulo)])).
```

A seguinte interrogação deve ser usada para aprender o conceito de arco através desses exemplos

```
?- bagof( E, exemplo(E), Exemplos), aprende(Exemplos, DescArco).
```

que tem como resultado

```
DescArco= conceito([parte1,parte2,parte3],
                   [suporta(parte1,parte3),suporta(parte2,parte3)],
                   [e_um(parte3,poligono_convexo),
                    e_um(parte1,retangulo),e_um(parte2,retangulo)],
                   [encosta(parte1,parte2)])

Exemplos= [+objeto([parte1,parte2,parte3],
                   [suporta(parte1,parte3),suporta(parte2,parte3),
```

```
        e_um(parte1,retangulo),e_um(parte2,retangulo),
        e_um(parte3,retangulo)]),
-objeto([parte1,parte2,parte3],
        [suporta(parte1,parte3),suporta(parte2,parte3),
        encosta(parte1,parte2),e_um(parte1,retangulo),
        e_um(parte2,retangulo),e_um(parte3,retangulo)]),
-objeto([parte1,parte2,parte3],
        [e_um(parte1,retangulo),e_um(parte2,retangulo),
        e_um(parte3,retangulo)]),
+objeto([parte1,parte2,parte3],
        [suporta(parte1,parte3),suporta(parte2,parte3),
        e_um(parte1,retangulo),e_um(parte2,retangulo),
        e_um(parte3,triangulo)])]
```

8.1.3 Listagem do Programa

Segue a listagem da implementação de uma versão simplificada do algoritmo de aprendizado ARCHES de Winston

```
%//////////////////////////////////////////////////////////////////////
%//                                                                  //
%//        Implementacao de uma versao simplificada e modificada do  //
%//   procedimento de aprendizagem de Winston chamado ARCHES         //
%//                                                                  //
%//////////////////////////////////////////////////////////////////////

   % Representacoes:
   % Objeto = objeto(ListaDePartes, ListaDeRelacoes)
   % Conceito = conceito(ListaDePartes, DeveRelacoes, Relacoes, NaoDeveRelacoes)
   % Exemplo positivo = + Objeto
   % Exemplo negativo = - Objeto
   % As partes de um objeto sao denotadas por variaveis
   % As partes de uma descricao de um conceito sao denotadas por constantes


aprende([PrimeiroExemplo | Exemplos], DescConceito):-
        inicializa(PrimeiroExemplo, HipoteseInicial),
        processa_exemplos(HipoteseInicial, Exemplos, DescConceito).

processa_exemplos(DescConceito, [], DescConceito).
processa_exemplos(DescAtual, [Exemplo | Exemplos], DescFinal):-
        tipo_objeto(Exemplo, Objeto, Tipo),
        casa(Objeto, DescAtual, Diferenca),
        atualiza(Tipo, Diferenca, DescAtual, DescNova),
        processa_exemplos(DescNova, Exemplos, DescFinal).

inicializa(+ objeto(Partes, Rels), conceito(Partes, [], Rels, [])):-
       nome_vars(Partes, [parte1, parte2, parte3, parte4, parte5, parte6]).
   %   Torna constantes as variaveis do objeto, assumindo no maximo 6 partes

   %   Acoes para generalizar ou especializar uma hipotese corrente:
   %      atualiza(TipoDeExemplo, Diferenca, DesCor, DescNova)

   %   A regra da relacao proibida: uma relacao extra em um exemplo negativo
   % deve ser proibida na descricao do conceito

atualiza(negativo, _ + [RelacaoExtra],
                conceito(Partes, Deve, Rels, NaoDeve),
                conceito(Partes, Deve, Rels, [RelacaoExtra | NaoDeve])).

   %   A regra da relacao requerida: relacoes que estao faltando devem ser
   % requeridas na descricao do conceito

atualiza(negativo, Faltando +_,
                conceito(Partes, Deve, Rels, NaoDeve),
                conceito(Partes, NovosDeve, RelNovas, NaoDeve)):-
        Faltando = [_|_],                       % Faltando nao-vazio
        conc(Faltando, Deve, NovosDeve),        % Adiciona Faltando em Deve
        lista_dif(Rels, Faltando, _+RelNovas).  % Remove Faltando das Rels
```

```
   %   Uma relacao faltando e uma extra em um exemplo negativo podem tambem
   % ser manipulados tanto pelo "extra proibido" como pelo "faltando requerido"

atualiza(negativo, [RFal] + [RExtra], DescAtual, DescNova):-
        atualiza(negativo, [] + [RExtra], DescAtual, DescInter),   % Proibida
        atualiza(negativo, [RFal] + [], DescInter, DescNova).      % e requerida

   %   A regra da taxonomia de subida: generaliza uma relacao e_um subindo um
   % tipo de taxonomia

atualiza(positivo, [e_um(Objeto, Classe1)] + [e_um(Objeto, Classe2)],
                conceito(Partes, Deve, Rels, NaoDeve),
                conceito(Partes, Deve, RelNovas, NaoDeve)):-
   %   Encontre  Classe, a superclasse mais especifica de Classe1 e Classe2
        suba(Classe1, Classe),
        suba(Classe2, Classe), !,
        troca(e_um(Objeto, Classe1), Rels, e_um(Objeto, Classe), RelNovas).

   %   casa(Objeto, DescricaoConceito, Diferenca)

casa(objeto(OPartes, ORels),
                conceito(CPartes, Deve, Rels, NaoDeve), Faltando + Extras):-
        lista_dif(ORels, Deve, [] + RestoRels),        % Casa Deve
        listas_curtas(Faltando + Extras),              % Generaliza listas curtas
        lista_dif(OPartes, CPartes, [] + []),          % Casa partes
        lista_dif(RestoRels, Rels, Faltando + Extras), % Casa outras relacoes
        lista_dif(Extras, NaoDeve, NaoDeve + _).

   %   lista_dif(Lista1, Lista2, Lista2MenosLista1 + Lista1MenosLista2)

lista_dif(Lista1, [], [] + Lista1).
lista_dif(Lista1, [X|Lista2], Falta + Extras):-
        apaga(Lista1, Lista11, X, Falta11, Falta),
        lista_dif(Lista11, Lista2, Falta11 + Extras).

  %  apaga(Lista, ListaPossivelSemX, X, ListaApagada, ListaApagadaPossivelComX)
  %  Se X eh apagado de Lista, entao ListaApagadaPossivelComX = ListaApagada,
  % se nao, entao ListaPossivelSemX = Lista e ListaApagadaPossivelComX =
  %      [X|ListaApagada]
  %  (Se X nao eh apagado entao esta faltando em Lista)

apaga([], [], X, Apagados, [X|Apagados]).
apaga([Y|L], L, X, Apagados, Apagados):-
        X == Y, !      % Literalmente igual: apagar necessariamente
        ;
        X = Y.         % X, Y casam: apagar possivelmente

apaga([Y|L], [Y|L1], X, Apagados, Apagados1):-
       apaga(L, L1, X, Apagados, Apagados1).

tipo_objeto(+Objeto, Objeto, positivo).   % Exemplo positivo
tipo_objeto(-Objeto, Objeto, negativo).   % Exemplo negativo

    %   troca(Item, Lista, NovoItem, NovaLista):
    % retire Item de Lista e adicione ItemNovo produzindo NovaLista
```

```
troca(Item, Lista, NovoItem, [NovoItem|Lista1]):-
         apaga(Lista, Lista1, Item, _, _).

   %   suba(Classe1, Classe2):-
   % suba um tipo de relacao de Classe1 para Classe2

suba(Classe, Classe).
suba(Classe, SuperClasse):-
        ako(Classe1, Classe),              % Classe eh um tipo de Classe1
        suba(Classe1, SuperClasse).

   %   nome_vars(Lista, ListaNomes)
   % instancia vaiaveis em uma lista de variaveis a nomes em ListaNomes

nome_vars(Lista, ListaNomes):-
       conc(Lista, _, ListaNomes).

conc([], L, L).
conc([X|L1], L2, [X|L3]):-
       conc(L1, L2, L3).

   %   gerador de diferencas  Lista1 + Lista2: listas curtas sao geradas
   % primeiramente para forcar o encontro de bom casamento
   %   ordem de geracao: []+[], []+[_], [_]+[], []+[_,_], [_]+[_], ...

listas_curtas(L1 + L2):-
conc(L, _, [_, _, _]),             % Ate 3 elementos em ambas as listas
conc(L1, L2, L).


    %******* Exemplos para aprendizado do conceito de arco **********%

exemplo(+ objeto([A, B, C],
               [suporta(A, C), suporta(B, C),
               e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo(- objeto([A, B, C],
               [suporta(A, C), suporta(B, C), encosta(A, B),
               e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo( -objeto([A, B, C],
               [e_um(A, retangulo), e_um(B, retangulo), e_um(C, retangulo)])).

exemplo( +objeto([A, B, C],
               [suporta(A, C), suporta(B, C),
               e_um(A, retangulo), e_um(B, retangulo), e_um(C, triangulo)])).


   % Taxonomia
ako(figura, poligono).
ako(figura, circulo).
ako(poligono, poligono_convexo).
ako(poligono, poligono_concavo).
ako(poligono_convexo, triangulo).
ako(poligono_convexo, retangulo).
```

```
ako(poligono_convexo, trapezio).
ako(poligono_convexo, hexagono).
```

8.2 Aprendizado de Descrições de Atributos

Nesta seção é descrito um programa de aprendizado onde os exemplos e conceitos são descritos em termos de um conjunto de atributos. A princípio, um atributo pode ser de qualquer tipo, numérico ou não numérico. Se o atributo for não numérico, seu conjunto de valores pode ser ordenado ou desordenado.

O programa implementado considera um pequeno conjunto de atributos não numéricos onde seu conjunto de valores não tem ordem estabelecida [Bratko 90]. Cada objeto é descrito em termos de atributos especificando valores concretos dos atributos do objeto.

A Figura 13 pg. 52 mostra o perfil de alguns objetos que serão utilizados como ilustração. Esses exemplos pertencem a cinco classes: porca, parafuso, chave, caneta, tesoura. Os atributos para esses elementos são: o tamanho, a forma e o número de buracos no objeto. Os possíveis valores para esses atributos são os seguintes

tamanho: pequeno, grande
formato: longo, compacto, outro
buracos: nenhum, 1, 2, 3, muitos

O objetivo é desenvolver um programa para aprender o conceito de porca, parafuso, chave, caneta e tesoura, utilizando os exemplos fornecidos. O resultado desse aprendizado será a descrição dessas classes em forma de regras, para serem utilizadas na classificação de novos objetos.

8.2.1 Descrição da Implementação

A implementação realizada utiliza a seguinte representação Prolog para descrever objetos

```
exemplo(Classe, [Atributo1=Val1, Atributo2=Val2, ...]).
```

Por exemplo,

```
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = longo, furos = nenhum]).
exemplo(chave, [tamanho = pequeno, formato = longo, furos = 1]).
...
```

O objetivo de um algoritmo de aprendizado é, a partir de exemplos fornecidos como os descritos acima, induzir conceitos que descrevem classes e podem ser utilizados para classificar novos objetos. Tais conceitos podem ser descritos através de regras, como foi adotado nesta implementação. Por exemplo, o formato adotado para decrever as regras, nas classes chave e tesoura, onde `<==` é declarado como operador infixo de tipo `xfx`, é o seguinte

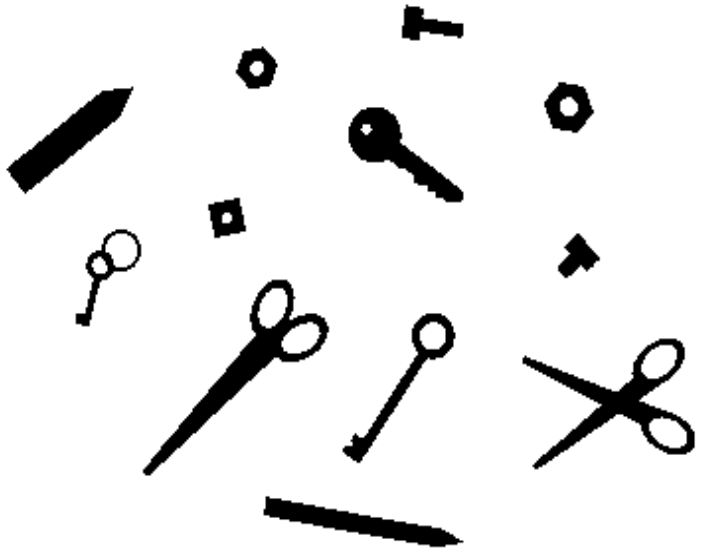


Figura 13: Perfil de Alguns Objetos[Bratko 90]

```
tesoura <== [buracos = 2, tamanho = grande]
chave <== [[formato = longo, tamanho = pequeno] [buracos = 1, formato = longo]]
```

cujo significado é

Se
tem 2 buracos e
seu tamanho é grande.
Então
O objeto é uma tesoura

Se
seu formato é longo e
seu tamanho é pequeno
ou
tem 1 buraco e
seu formato é longo.
Então
O objeto é uma chave

A forma geral de tais regras é

```
Classe <== [Conj1, Conj2, ...]
```

onde `Conj1`, `Conj2`, etc. são listas de valores de atributos da forma

```
[Atri1 = Val1, Atri2 = Val2, ...]
```

A descrição da classe `[Conj1, Conj2, ...]` é interpretada da seguinte forma

1. um objeto casa com a descrição se o objeto satisfaz pelo menos um dos `Conj1, Conj2`, etc.

2. um objeto satisfaz uma lista de valores de atributos `Conj` se todos os valores dos atributos em `Conj` são iguais aos do objeto.

Por exemplo, um objeto descrito por

```
[buracos = 1, formato = longo, tamanho = grande]
```

casa com a regra para chave satisfazendo a segunda lista de valores de atributos na regra. Assim, os valores de atributos em `Conj` são relacionados conjuntivamente, isto é, todos eles devem ser satisfeitos. Por outro lado, as listas `Conj1`, `Conj2`, etc., são relacionadas disjuntivamente, isto é, pelo menos uma delas deve ser satisfeita.

O casamento entre um objeto e uma descrição do conceito é realizado pelas seguintes cláusulas Prolog

```
casa(Obj, Descricao):-
        membro(Conjuncao, Descricao),
        satisfaz(Obj, Conjuncao).

satisfaz(Obj, Conjuncao):-
        not (membro(Atri = Val, Conjuncao),
             membro(Atri = ValX, Obj),
             ValX \== Val).
```

Deve ser observado que esta definição permite uma especificação parcial dos objetos. Se o valor de um atributo não é especificado, a definição assume que o valor não especificado satisfaz a requisição em `Conjuncao`.

O problema a ser tratado agora é a construção de descrições de Classes, ou seja, como construir as descrições das classes dados os exemplos. O algoritmo de aprendizado implementado é não incremental, todos os exemplos são processados de uma vez.

O principal requerimento é que a descrição de uma classe deve casar exatamente com todos os exemplos dessa classe. Assim, é necessário construir a descrição de uma dada classe tal que sejam cobertos todos os exemplos correspondentes a essa descrição. Em outras palavras, essa descrição é completa — cobre todos os exemplos positivos — e consistente — não cobre exemplos negativos. O procedimento

```
aprende(Exemplos, Classe, Descricao)
```

é responsável por encontrar essa descrição. A idéia do algoritmo é a seguinte

> Para cobrir todos os exemplos da classe `Classe` em `Exemplos` faça
> se nenhum exemplo em `Exemplos` pertence a `Classe`

então `Descricao = []`,
caso contrário `Descricao = [Conj|Conjs]`

onde `Conj` e `Conjs` são obtidos da seguinte forma

1. construa uma lista `Conj` de valores de atributos que cobre pelo menos um exemplo positivo de `Classe` e nenhum exemplo de qualquer outra classe
2. remova de `Exemplos` todos os objetos cobertos por `Conj` e cubra os restantes pela descrição `Conjs`.

Cada lista de valores é construída pelo procedimento

```
aprende_conj(Exemplos, Classe, Conjuncao)
```

A lista de valores de atributos `Conjuncao` é construída gradualmente, começando com a lista vazia, e adicionando a esta lista condições da forma

```
Atributo = Valor
```

Desta maneira, a lista de valores de atributos torna-se cada vez mais específica (cobre menos objetos). A lista de valores de atributos está pronta quando se torna tão específica que somente cobre exemplos positivos da classe.

O processo de construção de tal conjunção possui uma alta complexidade combinacional, pois são muitos os possíveis candidatos atributo-valor a serem adicionados na lista. O aprendizado pode ser visto como uma busca entre as possíveis descrições com o objetivo de minimizar o comprimento da descrição do conceito. Por causa da complexidade dessa busca, o programa utiliza uma função heurística. Em cada caso, somente o valor do atributo melhor estimado é adicionado à lista, descartando imediatamente todos os outros candidatos. A busca é então reduzida a um procedimento determinístico sem qualquer *backtracking*. Porém, ainda que esta busca seja eficiente, há o risco de não encontrar a descrição mais curta do conceito.

A heurística utilizada é simples e está baseada na seguinte idéia intuitiva

> *Uma condição valor-atributo útil deve fazer uma boa distinção entre exemplos positivos e negativos.*

Portanto, ela deve cobrir o maior número possível de exemplos positivos e o menor número possível de exemplos negativos. Essa função heurística é implementada pelo procedimento

```
score(Exemplos, Classe, ValorAtributo, Score)
```

8.2.2 Exemplo de Execução

O programa, cuja listagem se encontra na seção 8.2.3, pode ser executado para construir a descrição de alguma classe da seguinte maneira

```
?- aprende(porca), aprende(chave).

porca <== [formato = compacto,furos = 1]

chave <== [formato = outro,tamanho = pequeno]
          [furos = 1,formato = longo]
```

que representa as sguintes regras

1. *Se formato = compacto*
   e furos = 1
   então é uma porca

2. *Se formato = outro*
   e tamanho = pequeno *ou*
   furos = 1
   e formato = longo
   então é uma chave

8.2.3 Listagem do Programa

```
%////////////////////////////////////////////////////////////////////////////
%/    Implementacao de um programa de aprendizado onde exemplos e conceitos //
%/  sao descritos em termos de um conjunto de atributos.                     //
%////////////////////////////////////////////////////////////////////////////

% possiveis valores do atributo tamanho: pequeno, grande
atributo(tamanho, [pequeno, grande]).

% possiveis valores do atributo formato: longo, compacto, outro
atributo(formato, [longo, compacto, outro]).

% possiveis valores do atributo furos: nenhum, 1, 2, 3, muitos
atributo(furos, [nenhum, 1, 2, 3, muitos]).


% Representacao dos exemplos:
% exemplo( Classe, [Atributo1=Val1, Atributo2=Val2, ...]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = longo, furos = nenhum]).
exemplo(chave, [tamanho = pequeno, formato = longo, furos = 1]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(chave, [tamanho = grande, formato = longo, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = compacto, furos = nenhum]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = longo, furos = 2]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = outro, furos = 2]).
exemplo(chave, [tamanho = pequeno, formato = outro, furos = 2]).

:- op(300, xfx, <==).

% aprende(classe) : coleta exemplos em uma lista, constroi e escreve
% a descricao de Classe, e grava a regra correspondente sobre  a Classe.
aprende(Classe):-
        bagof(exemplo(ClasseX, Obj), exemplo(ClasseX, Obj), Exemplos),
        aprende(Exemplos, Classe, Descricao),
        nl,
        write(Classe),                      %imprime resultado
        write('<=='),
        nl,
        escreve_lista(Descricao),
        assert(Classe <== Descricao).      %grava regra

% aprende(Exemplos, Classe, Descricao):
% Descricao cobre exatamente os exemplos da classe Classe na lista de Exemplos
aprende(Exemplos, Classe, []):-
        not membro(exemplo(Classe, _), Exemplos).  % nao existe exemplo
aprende(Exemplos, Classe, [Conj|Conjs]):-
        aprende_conj(Exemplos, Classe, Conj),
        remove(Exemplos, Conj, RestExemplos),  % remove exemplos que casam
                                               % com Conj
        aprende(RestExemplos, Classe, Conjs).  % cobre os demais exemplos
```

```
% aprende_conj(Exemplos, Classe, Conj):
% Conj e' uma lista de valores de atributos satisfeitos por alguns exemplos
% da classe Classe e de nenhuma outra classe
aprende_conj(Exemplos, Classe, []):-
       not (membro(exemplo(ClasseX, _), Exemplos),   % nao ha exemplo
       ClasseX \== Classe), !.                       % de uma classe diferente
aprende_conj(Exemplos, Classe, [Cond|Conds]):-
        escolhe_cond(Exemplos, Classe, Cond),  % escolhe valores dos atributos
        filtra(Exemplos, [Cond], Exemplos1),
        aprende_conj(Exemplos1, Classe, Conds).

escolhe_cond(Exemplos, Classe, ValAtri):-
        findall(VA/Score, score(Exemplos, Classe, VA, Score), VAs),
        melhor(VAs, ValAtri).

melhor([ValAtri/_], ValAtri).
melhor([VA0/S0, VA1/S1| VASlista], ValAtri):-
        S1 > S0, !,                           % VA1 melhor que VA0
        melhor([VA1/S1|VASlista], ValAtri)
        ;
        melhor([VA0/S0|VASlista], ValAtri).

% filtra(Exemplos, Condicao, Exemplos1):
% Exemplos1 contem elementos de Exemplos que satisfazem a condicao
filtra(Exemplos, Cond, Exemplos1):-
        findall(exemplo(Classe, Obj),
                (membro(exemplo(Classe, Obj), Exemplos), satisfaz(Obj, Cond)),
                Exemplos1).

% remove(Exemplos, Conj, Exemplos1):
% Remove de Exemplos aqueles exemplos que casam com Conj e coloca resultado em
% Exemplos1
remove([],_,[]).
remove([exemplo(Classe, Obj)|Es], Conj, Es1):-
        satisfaz(Obj, Conj), !,           % primeiro exemplo casa com Conj
        remove(Es, Conj, Es1).            % remova-o
remove([E|Es], Conj, [E|Es1]):-
        remove(Es, Conj, Es1).

satisfaz(Obj, Conj):-
        not (membro(Atri = Val, Conj),    % Valor no conceito
        membro(Atri = ValX, Obj),         % e valor no objeto
            ValX \== Val).                % sao  diferentes

casa(Obj, Descricao):-
        membro(Conj, Descricao),
        satisfaz(Obj, Conj).

score(Exemplos, Classe, ValAtri, Score):-
        candidato(Exemplos, Classe, ValAtri),    % um valor de atributo adequado
        filtra(Exemplos, [ValAtri], Exemplos1),  % Exemplos1 satisfaz a
                                                 % condicao Atri = Val
        length(Exemplos1, N1),                   % tamanho da lista
        conta_pos(Exemplos1, Classe, NPos1),     % numero de exemplos positivos
        NPos1 > 0,          % Pelo menos um exemplo positivo casa com ValAtri
```

```
        Score is 2*NPos1 - N1.

candidato(Exemplos, Classe, Atri = Val):-
        atributo( Atri, Valores),               % obtem um atributo
        membro(Val, Valores),                   % obtem um valor
        adequado(Atri = Val, Exemplos, Classe).

adequado(ValAtri, Exemplos, Classe):-
     % pelo menos um exemplo negativo nao deve casar com ValAtri
        membro(exemplo(ClasseX, ObjX), Exemplos),
        ClasseX \== Classe,                     % exemplo negativo
        not satisfaz(ObjX, [ValAtri]), !.       % que nao casa


% conta_pos(Exemplo, Classe, N):
% N e' o numero de exemplos positivos de Classe
conta_pos([], _, 0).
conta_pos([exemplo(ClasseX, _)|Exemplos], Classe, N):-
        conta_pos(Exemplos, Classe, N1),
        (ClasseX = Classe, !, N is N1+1; N=N1).

membro(X, [X|_]).
membro(X, [_|L]):-
        membro(X, L).

escreve_lista([]).
escreve_lista([X|L]):-
        tab(2),
        write(X),
        nl,
        escreve_lista(L).

% procedimento de reconhecimento que utiliza as descricoes aprendidas
classifica(Objeto, Classe):-
        Classe <== Descricao,
        casa(Objeto, Descricao).
```

9 Aprendizado de Árvores de Decisão

Indução de árvores de decisão é um método muito utilizado em AM para aprender conceitos quando objetos são descritos através de atributos. A seguir é descrito o processo de construção de uma árvore de decisão, a partir de exemplos, de tal forma que a árvore classifique corretamente os exemplos, ou observações, fornecidos.

9.1 Construindo Árvores de Decisão

Para ilustrar tal processo, será considerado o seguinte exemplo apresentado originalmente em [Castineira 90]. Considere um programa de auditório que realiza o "namoro na TV". O telespectador quer descobrir um processo para prever se um candidato vai ou não arrumar uma namorada. Ele pode consultar o apresentador do programa, o qual tem muita experiência no assunto. Considere o seguinte diálogo entre o telespectador e o apresentador do programa

apresentador	—	O candidato é bonito ou feio?
telespectador	—	Ele é feio.
apresentador	—	Ele é inteligente?
telespectador	—	Não.
apresentador	—	O candidato tem dinheiro (pouco, mais ou menos ou muito)?
telespectador	—	Ele tem muito dinheiro.
apresentador	—	Pelo que você me fala o candidato provavelmente arrumará namorada.

que evidencia o uso de uma certa estrutura de decisão por parte do apresentador. Tal estrutura pode ser uma árvore de decisão. A Figura 14 pg. 60 mostra como este diálogo pode ser representado na forma de árvore de decisão.

Essa árvore de decisão parcial representa apenas a situação resultante do diálogo acima descrito. Uma árvore completa poderia ser construída representando todas as possíveis perguntas com todas as possíveis respostas. A partir da árvore de decisão pode ser formulado um conjunto equivalente de regras. Por exemplo, percorrendo a árvore da Figura 14 da raiz até o nó final (folha) pode-se deduzir a seguinte regra

Se	*o candidato é feio*
e	*não é inteligente*
e	*tem muito dinheiro*
então	*ele arrumará namorada.*

Neste exemplo inteligência, beleza e situação financeira são chamados de atributos ou indicadores. A Tabela 2 mostra, para o exemplo considerado, os possíveis valores desses atributos.

Suponha agora que o telespectador quer achar mais regras para predizer se um determinado candidato será capaz ou não de arrumar namorada. Para isso, ele confecciona

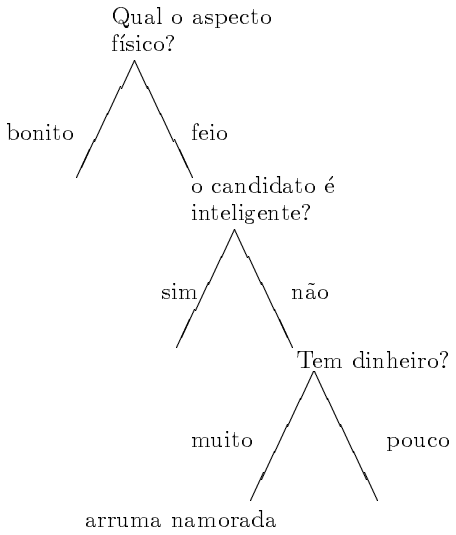


Figura 14: Árvore de Decisão Incompleta

Atributo	**Valor**
inteligência	{sim, não}
beleza	{bonito, feio}
situação financiera	{rico, médio, pobre}

Tabela 2: Atributo-Valor

uma tabela com uma lista de alguns candidatos que já passaram pelo programa, alguns fatos sobre eles (suas características ou atributos) e seu desempenho (se arrumaram ou não namorada). Uma amostra destes dados é ilustrada na Tabela 3.

inteligência	beleza	sit.financeira	arrumou namorada
sim	bonito	rico	+
não	feio	pobre	−
sim	feio	pobre	+
sim	feio	médio	+
não	bonito	pobre	−
não	bonito	médio	+
não	bonito	rico	+
não	feio	rico	+

Tabela 3: Conjunto de Observações ou Exemplos

Cada linha dessa tabela é um exemplo ou observação, os rótulos das colunas (inteligência, beleza, etc.), como já mencionado, são os atributos, sendo que o último atributo (arrumou namorada) é chamado atributo de classe. É esse atributo que divide os exemplos nas diferentes classes — neste caso uma classe positiva e outra negativa. Os candidatos que arrumaram namorada formam a classe positiva (+) e os que não arrumaram namorada formam a classe negativa (−). O objetivo é determinar uma relação entre as classes e os outros atributos.

Por exemplo, a primeira observação da tabela diz que um candidato muito inteligente, bonito e rico arrumou namorada. Porém, essa tabela não diz nada a respeito do desempenho de futuros candidatos. Para isto constrói-se uma árvore de decisão completa representando esses dados. Para construir essa árvore de decisão, primeiramente deve ser selecionado um atributo para ser a raiz da árvore e então dividir as observações segundo os valores desse atributo; seja ele o atributo inteligência. Deve-se, então, dividir as observações em dois grupos: um com os candidatos inteligentes e outro com os não inteligentes; coloca-se cada um desses grupos no extremo correspondente dos ramos da árvore, como mostrado na Figura 15. Assim, todas as observações nas quais o candidato é inteligente são colocadas no ramo rotulado com inteligência:sim. Nessas observações foi retirado o atributo inteligência, pois sabe-se que todos estes candidatos são inteligentes. De maneira análoga, todas as observações de candidatos pouco inteligentes vão para o outro ramo.

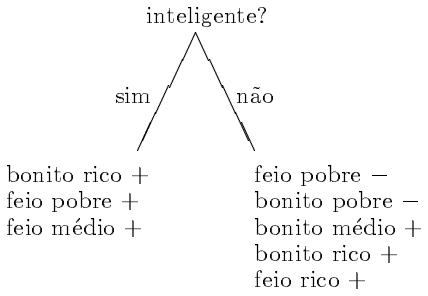


Figura 15: Construindo a Árvore de Decisão – Passo 1

Todas as observações do ramo inteligência:sim pertencem à mesma classe, ou seja, todas indicam que o candidato arrumou namorada. Quando isso acontece, as observações devem ser substituídas pela classe comum a todas elas. No presente caso, a árvore adquire a forma ilustrada na Figura 16.

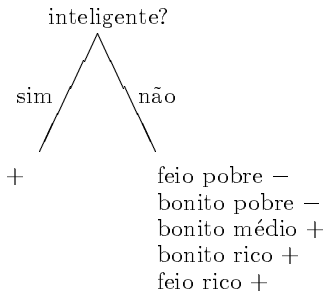


Figura 16: Construindo a Árvore de Decisão – Passo 2

No ramo dos pouco inteligentes ainda existem observações de duas classes diferentes. Quando isso acontece, escolhe-se outro atributo e repete-se a divisão em ramos. Suponha que agora é considerado o atributo situação financeira para realizar a próxima ramificação. Desta vez deve-se separar os dados que estão no ramo dos "não inteligentes" em três sub-ramos, segundo a situação financeira

- tem muito dinheiro (rico)
- situação financeira média (médio)
- tem pouco dinheiro (pobre).

A Figura 17 mostra o resultado desta ramificação.

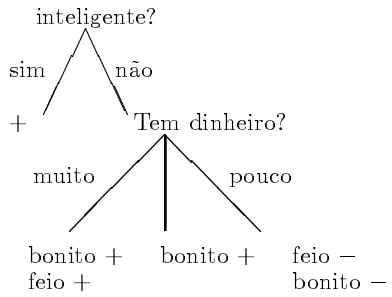


Figura 17: Construindo a Árvore de Decisão – Passo 3

Agora, cada sub-ramo tem observações que representam classes homogêneas. Pode-se então substituir as observações por suas classes. O resultado é mostrado na Figura 18.

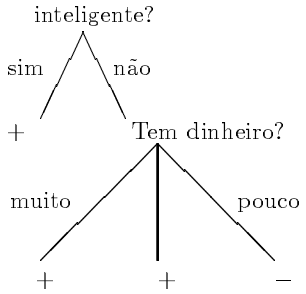


Figura 18: Árvore de Decisão

Como cada folha da árvore contém uma única classe, o processo de ramificação ou construção da árvore está completo. Assim, se seus ramos forem percorridos primeiro em profundidade até chegar nas folhas, a árvore representa as seguintes regras

1. *Se o candidato é inteligente*
   então arruma namorada.
2. *Se ele não é inteligente*
   e é rico
   então arruma namorada.
3. *Se ele não é inteligente*
   e sua situação financeira é média
   então arruma namorada.

4. *Se ele não é inteligente*
   e é pobre
   então não arruma namorada.

Deve ser observado que é produzida uma regra para cada nó terminal da árvore.

A Figura 18 mostra uma característica interessante do processo de construção da árvore de decisão. Apesar do conjunto de observações originais conter três atributos (beleza, inteligência e situação financeira), o atributo beleza não foi necessário para classificar as observações do conjunto. Neste caso, pode-se dizer que o atributo beleza é um atributo *irrelevante*.

Certamente, a árvore da Figura 18 não é a única árvore que pode ser construída a partir desse conjunto de exemplos. Dependendo do atributo escolhido em cada passo, podem ser geradas diferentes árvores representando os mesmos dados. Por exemplo, se tivesse sido escolhido o atributo beleza para começar a dividir o conjunto amostra da Tabela 3 pg. 60, teria sido gerada uma árvore como a da Figura 19.

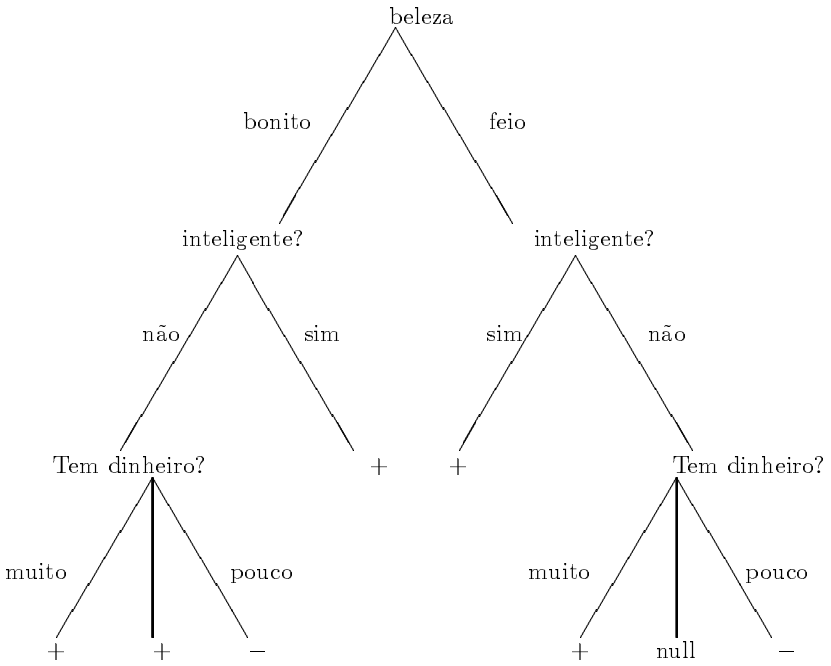


Figura 19: Árvore de Decisão com Atributo Inicial beleza

Essa árvore é mais complexa e ramificada do que a da Figura 18, embora cubra o mesmo conjunto de exemplos. Essa árvore também gera regras mais complexas, pois as regras possuem mais condições e são mais numerosas — nesse caso tem-se 7 regras contra 4 da árvore anterior. No caso de problemas reais, com grande número de atributos e maior número de observações, é importante gerar regras concisas e no menor número possível.

Do que foi dito, conclui-se que é necessário um método para descobrir qual é o atributo mais relevante (ou melhor atributo) a ser utilizado para ramificar cada nó da árvore de decisão. Existem diversos métodos para encontrar o atributo mais relevante. Um deles, utilizado no algoritmo ID3 [Quinlan 79] e baseado na teoria da informação, utiliza a entropia para escolher o atributo mais relevante.

9.2 Entropia

Entropia é uma grandeza que mede desordem, tanto de objetos físicos quanto de informações. Quanto maior a entropia maior a desordem. Arariboia explica o conceito de entropia de uma forma muito clara [Arariboia 89]

> *Para entender o que é ordem, imagine que haja várias gavetas contendo objetos embalados em caixas. Se cada uma das gavetas contém apenas objetos de um só tipo, não precisamos abrir as caixas para saber que objetos estão dentro delas. Em outras palavras, se a caixa foi retirada de uma gaveta de lápis, não precisamos abrir a referida caixa para saber que contém lápis. Não precisamos, em suma, de nenhuma informação para descobrir o que está dentro da caixa. E por que não foi necessária nenhuma informação? Porque os objetos estavam adequadamente organizados dentro das gavetas. Se, porém, cada gaveta contivesse objetos dos mais variados tipos, teríamos que abrir a caixa para saber qual objeto foi retirado de cada gaveta. Precisaríamos de informação para classificar o objeto. Um grupo de objetos está tanto mais organizado quanto menos informação externa for preciso para classificá-lo. A propósito, a entropia, no caso em que cada gaveta contém apenas um tipo de objeto, é nula, pois tudo está em ordem.*
>
> *É mais ou menos intuitivo que devemos escolher o atributo que produza a ramificação de menor entropia (e, portanto, de maior organização). De preferência, gostaríamos de ter entropia nula. Isto significa que, em um dado ramo, todas as classes são iguais e, portanto, não precisamos de mais nenhuma informação para descobrir uma determinada classe, basta saber em que ramo ela está. Neste caso os ramos são como gavetas, contendo, cada uma, apenas um tipo de objeto.*

Para calcular a entropia de um determinado atributo é necessário calcular a entropia de uma ramificação. A fim de exemplificar, é considerado o caso do atributo inteligência. Esse atributo produz dois ramos, como mostrado na Figura 15 pg. 61. O ramo dos candidatos inteligentes contém observações que pertencem a uma mesma classe (+), portanto, ele está ordenado e possui entropia zero — a desorganização dele é nula. O ramo inteligência: não contém observações de duas classes diferentes; em outras palavras, ele ainda está desorganizado. Neste caso, é necessário calcular a entropia dos atributos participantes a fim de determinar o melhor atributo a ser utilizado na ramificação. O cálculo da entropia de um determinado ramo ainda desorganizado é descrito a seguir.

Se uma observação pode ser classificada em n classes diferentes c_1, ...c_n, e a probabilidade de um objeto pertencer a classe c_i é $p(i)$, então, a entropia de classificação do ramo é dada por

$$E(A = v_j) = -\sum_{i=1}^{n} p(i) \log_2 p(i) \tag{1}$$

onde $A = v_j$ significa que o atributo A tem o valor v_j. Isto é, $E(A = v_j)$ é a entropia do ramo correspondente ao valor v_j do atributo A. No caso do atributo inteligência — Tabela 3 pg. 60 — tem-se que a entropia correspondente ao ramo de inteligência:não é

$$\begin{aligned} E(inteligência = nao) &= -2/5 \log_2(2/5) - 3/5 \log_2(3/5) \\ &= 0.9709505 \end{aligned}$$

pois 2/5 das observações no ramo correspondente a pouca inteligência pertencem a classe negativa e 3/5 das observações pertencem a classe positiva. Analogamente, pode-se verificar que a entropia do ramo inteligência:sim é zero

$$\begin{aligned} E(inteligência = sim) &= -3/3 \log_2(3/3) \\ &= -1 \log_2 1 \\ &= 0 \end{aligned}$$

Para calcular a entropia total de um atributo deve-se relacionar a entropia de cada um dos ramos correspondentes a esse atributo. Seja

M o número de observações totais,

$E_1, E_2, E_3, ...E_k$ as entropias de cada ramo pertencente ao atributo escolhido, e

$N_1, N_2, N_3 ...N_k$ o número de elementos de cada ramo.

Neste caso, a entropia de toda a ramificação, ou seja, a entropia do atributo A é dada por

$$E(A) = \sum_{i=1}^{k} E_i * N_i/M \tag{2}$$

Por exemplo, no caso do atributo inteligência tem-se dois ramos, um contém três das oito observações totais e o outro possui cinco dessas observações, assim

$$\begin{aligned} E(inteligência) &= 3/8E(inteligência = sim) + 5/8E(inteligência = nao) \\ &= 0 * 3/8 + 5/8 * 0.9709505 \\ &= 0.6068441 \end{aligned}$$

Realizando o mesmo processo com os outros dois atributos obtem-se

$$E(beleza) = 0.8112781$$
$$E(situac.financeira) = 0.3443609$$

Portanto, o atributo situação_financeira é o melhor atributo, enquanto que beleza é o pior atributo para começar a construir a árvore de decisão correspondente aos exemplos da Tabela 3 pg. 60. Assim, para gerar árvores de decisão mais simples, em cada nó da árvore deve-se escolher o atributo de menor entropia para continuar ramificando a árvore.

9.3 Algoritmo Geral da Família TDIDT

Deve ser observado que a árvore de decisão pode ser geral, no sentido de não apenas classificar objetos em duas classes — positiva e negativa — mas em múltiplas classes.

Existe uma família de sistemas de AM de propósito geral, que tem como característica a construção de árvores de decisão através de processos indutivos, utilizando uma metodologia Top Down, isto é, a árvore é construída começando da raiz e descendo até as folhas. Essa família é denominada TDIDT — Top Down Induction of Decision Tree — e representa o conhecimento adquirido (conceito) processando exemplos observados, através de árvores de decisão.

O algoritmo geral desses sistemas começa com uma árvore de decisão vazia que é refinada gradualmente até que classifique corretamente todos os exemplos da amostra. O procedimento geral é o seguinte:

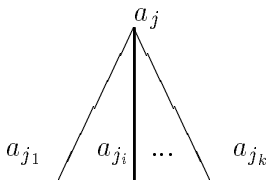
Dado um conjunto de exemplos de aprendizado E;
uma condição de parada $t(E)$; e
uma função de avaliação $aval(E, atributo)$

Se todas as instâncias em E satisfazem a condição de término $t(E)$

então retorne o valor da classe

caso contrário

1. para cada atributo a_i determine o valor da função $aval(E, a_i)$. Seja a_j o atributo que possui o melhor valor** de $aval(E, a_i)$.
2. se a_j possui valores a_{j_1}, a_{j_2}, ... a_{j_k} crie o seguinte nó da árvore:



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

O critério de parada t pode ser definido tanto para construir a árvore de decisão, que classifica exatamente todos os elementos do conjunto de aprendizado em domínios

**Atributo mais relevante para realizar a ramificação, segundo a função de avaliação.

completos, quanto para decidir pela não expansão da árvore quando a evidência for insuficiente nos exemplos fornecidos. Esse mecanismo é denominado pré-poda da árvore de decisão. A função de avaliação *aval* pode ser definida de várias formas diferentes.

As diversas condições de parada e funções de avaliação dão origem a vários algoritmos dessa família. Como já mencionado, o sistema ID3 [Quinlan 79] utiliza a "entropia" como função de avaliação dos atributos e seu critério de parada é que todos os exemplos no nó pertençam à mesma classe. Embora essa abordagem dê bons resultados nesse sistema, nada garante que esses sejam os melhores resultados. Outro critério ou função de avaliação poderia ser utilizado.

A continuação é apresentada uma implementação muito simples do algoritmo geral da família TDIDT.

9.3.1 Descrição da Implementação

A fim de ilustrar, será considerado o mesmo exemplo que o da seção 8.2.1. Nesse caso, os atributos dos objetos estão relacionados com o tamanho, forma e número de furos do objeto, como mostrado a seguir

Atributo	Valor
tamanho	{pequeno, grande}
forma	{longo, compacto, outros}
furos	{nenhum, 1, 2, 3, muitos}

Na implementação realizada, os exemplos são representados através de um conjunto de cláusulas Prolog da seguinte forma

```
exemplo(Classe, [Atributo1 = Valor1, Atributo2 = Valor2, ...]).
```

O procedimento para induzir a árvore de decisão é

```
arvore_inducao(Atributos, Exemplos, Arvore).
```

onde `Arvore` é a árvore induzida dos `Exemplos` usando os atributos na lista `Atributos`. Considerando que os atributos e exemplos estão representados, respectivamente, como

```
atributo(tamanho, [pequeno, grande]).
atributo(formato, [longo, compacto, outro]).
atributo(furos, [nenhum, 1, 2, 3, muitos]).

exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = longo, furos = nenhum]).
exemplo(chave, [tamanho = pequeno, formato = longo, furos = 1]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(chave, [tamanho = grande, formato = longo, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = compacto, furos = nenhum]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = longo, furos = 2]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = outro, furos = 2]).
exemplo(chave, [tamanho = pequeno, formato = outro, furos = 2]).
```

é possível coletar todos os exemplos e os atributos disponíveis em listas da seguinte forma

```
arvore_inducao(Arvore):-
        findall(exemplo(Classe, Obj), exemplo(Classe, Obj), Exemplos),
        findall(Atri, atributo(Atri, _), Atributos),
        arvore_inducao(Atributos, Exemplos, Arvore).
```

A árvore induzida possui uma forma diferente dependendo dos seguintes casos possíveis

1. `Arvore = null` se o conjunto de exemplos é vazio;
2. `Arvore = folha(Classe)` se todos os exemplos são de uma mesma classe `Classe`
3. `Arvore = arvore( Atributo, [Val1 : Sub_arvore1, Val2 : Sub_arvore2,...])` se os exemplos pertencerem a mais de uma classe, `Atributo` é a raiz da árvore, `Val1, Val2, ...` são seus valores, e `Sub_arvore1, Sub_arvore2, ...` são as subárvores correspondentes.

Esses três casos estão representados pelas seguintes três cláusulas

```
arvore_inducao(_, [], null):- !.

arvore_inducao(_, [exemplo(Classe, _)| Exemplos1], folha(Classe)):-
        mesma_classe(Classe, Exemplos1),
        !.

arvore_inducao(Atributos, Exemplos, arvore(Atributo, Sub_arvores)):-
        escolhe_atributo(Atributos, Exemplos, Atributo),
        apaga(Atributo, Atributos, RestaAtri),
        atributo(Atributo, Valores),
        arvores_inducao(Atributo, Valores, RestaAtri, Exemplos, Sub_arvores).
```

onde `arvores_inducao/5` induz `Sub_arvores` para um subconjunto de `Exemplos` de acordo com `Valores` de `Atributo`

```
% arvores_inducao(Atri, Vals, RestaAtri, Exemplos, Sub_arvores)

arvores_inducao(_, [], _, _, []).

arvores_inducao(Atri, [Val1|Vals], RestaAtri, Exs, [Val1:Arvore1|Arvores]):-
        atrival_subconj(Atri = Val1, Exs, ExemploSubconj),
        arvore_inducao(RestaAtri, ExemploSubconj, Arvore1),
        arvores_inducao(Atri, Vals, RestaAtri, Exs, Arvores).
```

O procedimento `atrival_subconj(Atributo = Valor, Exemplos, Subconj)`, definido a sguir, é verdadeiro se `Subconj` é um subconjunto de exemplos de `Exemplos` que satisfaz a condição `Atributo = Valor`

```
atrival_subconj(AtriValor, Exemplos, ExemploSubconj):-
        findall(exemplo(Classe, Obj), (membro(exemplo(Classe, Obj), Exemplos),
                            satisfaz(Obj, [AtriValor])), ExemploSubconj).
```

O procedimento `escolhe_atributo/3` seleciona o atributo que melhor discrimina entre as classes em função da entropia desses atributos minimizando a medida da entropia

escolhida, usando `findall/3` e `keysort/2` — predicados pré-definidos de Prolog. `findall/3` coloca em uma lista (3º argumento) todas as entropias com seus respectivos atributos e `keysort/2` ordena a lista em ordem crescente de entropia (2º atributo).

```
escolhe_atributo(Atris, Exemplos, MelhorAtri):-
        numero_elementos(Exemplos, 0, Num_Elem),
        findall(Ent_final-Atri,
            (membro(Atri, Atris), atributo(Atri, Valores),
            entropia_total(Exemplos, Num_Elem, Atri, Valores, 0, Ent_final)),
            ListaEntropia),
        keysort(ListaEntropia, ListaOrdenada),
        primeiro_elemento(ListaOrdenada, MelhorAtri).
```

O procedimento

```
entropia_total(_, _, _, [], Ent, Ent).

entropia_total(Exemplos, Num_Elem, Atri, [Valor1|Valor2], Ent1, Ent_final):-
        atrival_subconj(Atri = Valor1, Exemplos, Obs),
        calcula_entropia(Obs, Num, Entropia),
        Ent2 is Ent1 + (Num * Entropia / Num_Elem),
        entropia_total(Exemplos, Num_Elem, Atri, Valor2, Ent2, Ent_final).
```

calcula a entropia. `entropia_total/6` combina as entropias dos subconjuntos de exemplos após dividir a lista `Exemplos` de acordo com os valores de `Atributos`.

O procedimento `calcula_entropia/3` é o responsável pelo cálculo da entropia de cada subconjunto de exemplos

```
calcula_entropia([], 0, 0).

calcula_entropia(Obs, Num, Entropia):-
        numero_elementos(Obs, 0, Num),
        setof(D, diagnosticos(D, Obs), Ds),
        entropia(Ds, Obs, Num, 0, Entropia).
```

Para esse cálculo é utilizado o procedimento `entropia/5`, definido a seguir, que faz a somatória propriamente dita da entropia de cada classe

```
entropia([], Obs, Num, En, En):- !.

entropia([D|Ds], Obs, Num, En1, En):-
        conta_diagnostico(D, Obs, 0, ND),
        En2 is En1 - (ND/Num)*log(ND/Num)/log(2),
        entropia(Ds, Obs, Num, En2, En).
```

9.3.2 Exemplo de Execução

O programa, cuja listagem se encontra na seção 9.3.3, pode ser executado para construir a árvore de decisão da seguinte forma

```
?- arvore_inducao(Arvore).
```

cuja resposta, utilizando o conjunto de atributos e exemplos da seção anterior é

```
Arvore = arvore(furos,[nenhum : arvore(tamanho,[pequeno : folha(parafuso),
                                                grande : folha(caneta)]),
                  1 : arvore(formato,[longo : folha(chave),compacto : folha(porca),
                                      outro : null]),
                  2 : arvore(tamanho,[pequeno : folha(chave),grande : folha(tesoura)]),
                  3 : null,muitos : null])
```

Percorrendo essa árvore em profundidade até chegar nas folhas obtém-se as seguintes regras de decisão

1. *Se furos = nenhum*
   e tamanho = pequeno
   então é um parafuso

2. *Se furos = nenhum*
   e tamanho = grande
   então é uma caneta

3. *Se furos = 1*
   e formato = longo
   então é uma chave

4. *Se furos = 1*
   e formato = compacto
   então é uma porca

5. *Se furos = 2*
   e tamanho = pequeno
   então é uma chave

6. *Se furos = 1*
   e tamanho = grande
   então é uma tesoura

Deve ser observado que o conceito aprendido por esse programa e aquele descrito na seção 8.2 pg. 51 não são necessariamente idênticos, já que utilizam diferentes métodos de aprendizado.

9.3.3 Listagem do Programa

```
%----------------------- Lista de atributos -----------------------------
atributo(tamanho, [pequeno, grande]).
atributo(formato, [longo, compacto, outro]).
atributo(furos, [nenhum, 1, 2, 3, muitos]).

%------------------------ Lista de exemplos -----------------------------
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = longo, furos = nenhum]).
exemplo(chave, [tamanho = pequeno, formato = longo, furos = 1]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(chave, [tamanho = grande, formato = longo, furos = 1]).
exemplo(parafuso, [tamanho = pequeno, formato = compacto, furos = nenhum]).
exemplo(porca, [tamanho = pequeno, formato = compacto, furos = 1]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = longo, furos = 2]).
exemplo(caneta, [tamanho = grande, formato = longo, furos = nenhum]).
exemplo(tesoura, [tamanho = grande, formato = outro, furos = 2]).
exemplo(chave, [tamanho = pequeno, formato = outro, furos = 2]).

%----------------------- arvore_inducao(-) ------------------------------
% Chamada principal do programa, que inicializa a formacao da arvore de
% inducao e chama as demais subrotinas para a formacao do restante da
% arvore.

arvore_inducao(Arvore):-
        findall(exemplo(Classe, Obj), exemplo(Classe, Obj), Exemplos),
        findall(Atri, atributo(Atri, _), Atributos),
        arvore_inducao(Atributos, Exemplos, Arvore).

%----------------------- arvore_inducao(+,+,-) --------------------------
% Programa que monta uma arvore de decisao induzida pelos Exemplos usando
% os atributos da lista Atributos. A arvore e' retornada pelo terceiro
% parametro.

% Arvore = null caso o conjunto de exemplos esteja vazio

arvore_inducao(_, [], null):- !.

% Arvore = folha(Classe) caso todos os exemplos pertencerem a mesma classe

arvore_inducao(_, [exemplo(Classe, _)| Exemplos1], folha(Classe)):-
        mesma_classe(Classe, Exemplos1),
        !.

% Arvore  = arvore(Atributo, [Val1: subarvore1, Val2: subarvore2, ...]) caso
% os exemplos pertencam a mais de uma classe, Atributo e' a raiz da arvore,
% Val1, Val2, ... sao os seus valores e subarvore1, subarvore2, ... sao as
% arvores de decisao correspondentes

arvore_inducao(Atributos, Exemplos, arvore(Atributo, Sub_arvores)):-
        escolhe_atributo(Atributos, Exemplos, Atributo),
        apaga(Atributo, Atributos, RestaAtri),
        atributo(Atributo, Valores),
        arvores_inducao(Atributo, Valores, RestaAtri, Exemplos, Sub_arvores).
```

```
%---------------------- arvores_inducao(+,+,+,+,-) ----------------------
% O programa arvores_inducao induz Sub_arvores de decisao para subconjun-
% tos de Exemplos de acordo com os Valores de Atributo.

arvores_inducao(_, [], _, _, []).

arvores_inducao(Atri, [Val1|Vals], RestaAtri, Exs, [Val1:Arvore1|Arvores]):-
        atrival_subconj(Atri = Val1, Exs, ExemploSubconj),
        arvore_inducao(RestaAtri, ExemploSubconj, Arvore1),
        arvores_inducao(Atri, Vals, RestaAtri, Exs, Arvores).

%----------------------- atrival_subconj(+,+,-) -----------------------
% E' verdadeiro se Subconj. e' um subconjunto de exemplos em Exemplos que
% satisfaca a condicao Atributo = Valor.

atrival_subconj(AtriValor, Exemplos, ExemploSubconj):-
        findall(exemplo(Classe, Obj), (membro(exemplo(Classe, Obj), Exemplos),
                           satisfaz(Obj, [AtriValor])), ExemploSubconj).

%---------------------- escolhe_atributo(+,+,-) ----------------------
% O programa que segue nada mais faz que encontrar o candidato com menor
% entropia. Em cada interacao, existe um candidato a melhor classificador
% que sera inserido em uma lista de candidatos. Apos a obtencao de todos
% os candidatos, a lista sera ordenada em ordem crescente de acordo com o
% valor de sua entropia. No final, o melhor candidato sera o primeiro ele-
% mento da lista, ou seja, o de menor entropia.

escolhe_atributo(Atris, Exemplos, MelhorAtri):-
        numero_elementos(Exemplos, 0, Num_Elem),
        findall(Ent_final-Atri,
            (membro(Atri, Atris), atributo(Atri, Valores),
            entropia_total(Exemplos, Num_Elem, Atri, Valores, 0, Ent_final)),
            ListaEntropia),
        keysort(ListaEntropia, ListaOrdenada),
        primeiro_elemento(ListaOrdenada, MelhorAtri).

%---------------------- primeiro_elemento(+,-) ----------------------
% Obtem o primeiro elemento de uma lista.

primeiro_elemento([_ - Elemento|_], Elemento).

%--------------------- entropia_total(+,+,+,+,+,-) ---------------------
% O programa abaixo calcula a entropia total de um conjunto de ramos.
% Os ramos sao os correspondentes ao indicador Atri. Este indicador e'
% dado pela lista de seus valores.

entropia_total(_, _, _, [], Ent, Ent).

entropia_total(Exemplos, Num_Elem, Atri, [Valor1|Valor2], Ent1, Ent_final):-
        atrival_subconj(Atri = Valor1, Exemplos, Obs),
        calcula_entropia(Obs, Num, Entropia),
        Ent2 is Ent1 + (Num * Entropia / Num_Elem),
        entropia_total(Exemplos, Num_Elem, Atri, Valor2, Ent2, Ent_final).

%---------------------- calcula_entropia(+,+,-) ----------------------
```

```
% A funcao abaixo efetua o calculo da entropia de uma lista de obser-
% vacoes.

calcula_entropia([], 0, 0).

calcula_entropia(Obs, Num, Entropia):-
        numero_elementos(Obs, 0, Num),
        setof(D, diagnosticos(D, Obs), Ds),
        entropia(Ds, Obs, Num, 0, Entropia).


%------------------------ entropia(+,+,+,+,-) -------------------------
% O programa entropia faz a somatoria das entropias de cada um dos diag-
% nosticos. O quarto argumento e' o parametro acumulador utilizado para
% obter a referida somatoria. O resultado e' colocado no quinto parametro.

entropia([], Obs, Num, En, En):- !.

entropia([D|Ds], Obs, Num, En1, En):-
        conta_diagnostico(D, Obs, 0, ND),
        En2 is En1 - (ND/Num)*log(ND/Num)/log(2),
        entropia(Ds, Obs, Num, En2, En).

%---------------------- conta_diagnostico(+,+,+,-) ----------------------
% Dado um diagnostico D, o programa abaixo conta quantas vezes ele aparece
% em uma lista de observacoes.

conta_diagnostico(D, [], ND, ND):- !.

conta_diagnostico(D, [exemplo(DD, _)|RObs], ND1, ND):-
        incrementa(D, DD, ND1, ND2),
        conta_diagnostico(D, RObs, ND2, ND).

%------------------------ incrementa(+,+,+,-) --------------------------
% O parametro acumulador de conta_diagnostico(_,_,_,_) e' incrementado
% pelo programa abaixo. Observe que o parametro so' e' incrementado quan-
% do o diagnostico da observacao e' igual ao que esta sendo contado.

incrementa(D, D, N, N1):-
        N1 is N + 1,
        !.

incrementa(_, _, N, N).

%----------------------- numero_elementos(+,+,-) -----------------------
% O programa abaixo conta o numero de elementos em uma lista.

numero_elementos([], N, N):- !.

numero_elementos([_|R], N1, N):-
        N2 is N1 + 1,
        numero_elementos(R, N2, N).

%-------------------------- diagnosticos(-,+) --------------------------
% Dada uma lista de observacoes, o programa abaixo fornece um dos diagnos-
% ticos dela. Se este nao servir, fornece outro. E assim por diante. E'
```

```
% usado com setof(_,_,_) em calcula_entropia(_,_,_) para se conseguir a
% lista de todos os diagnosticos distintos que estao presentes em uma lista.

diagnosticos(D, [exemplo(D, _)|_]).

diagnosticos(D, [_|R]):-
        diagnosticos(D, R).

%-------------------------- mesma_classe(+,+) -------------------------
% O programa abaixo verifica se todos os exemplos de uma lista pertencem
% a uma mesma classe.

mesma_classe(_, []).

mesma_classe(Classe, [exemplo(ClasseX, _)|Cauda]):-
        Classe == ClasseX,
        mesma_classe(Classe, Cauda).

mesma_classe(Classe, _):-
        !,
        fail.

%------------------------------ membro(?,?) ------------------------------
% O programa membro(_,_) verifica se um dado elemento e' membro de uma
% lista, ou dada uma lista determina cada um de seus membros.

membro(X, [X|_]).

membro(X, [_|L]):-
        membro(X, L).

%---------------------------- satisfaz(+,+) ----------------------------
% O programa verifica se um determinado atributo e seu valor satisfazem
% uma determinada condicao.

satisfaz(Obj, Conj):-
        membro(Atri = Val, Conj),
        membro(Atri = ValX, Obj),
        ValX == Val.

%------------------------------ apaga(+,+,-) ----------------------------
% O programa elimina um atributo da lista de atributos caso ele ja tenha
% sido usado.

apaga(_,[],[]).

apaga(Atributo,[Atributo|Cauda1], Cauda2):-
        apaga(Atributo, Cauda1, Cauda2), !.

apaga(Atributo, [E|Cauda1],[E|Cauda2]):-
        apaga(Atributo, Cauda1 , Cauda2).
```

10 Considerações Finais

Em geral, os sistemas de Aprendizado de Máquina podem ser vistos como programas capazes de criar classificadores a partir de conjuntos de exemplos. O principal objetivo está em extrair conceitos expressos em alguma linguagem, por exemplo, regras de decisão, capazes de serem aplicadas a novos casos. Os classificadores são sistemas que utilizam tais regras para classificar casos nunca vistos.

Neste trabalho foi contextualizado Aprendizado de Máquina e foram descritos em detalhes implementações simples de alguns algoritmos de Aprendizado Simbólico de Máquina por exemplos. As implementações foram realizadas na linguagem de programação lógica Prolog, sintaxe de Edinburgh, mais especificamente Arity Prolog [Arity 90].

Uma das principais utilidades da linguagem Prolog é a prototipagem rápida. Em prototipagem, a ênfase está em implementar novas idéias rapidamente e a um custo pequeno, sem levar em consideração, necessariamente, a eficiência da implementação. Após desenvolver um primeiro protótipo, o sistema pode ser novamente implementado — em outra ou na mesma linguagem de programação — considerando agora a eficiência da implementação.

É importante ressaltar que este trabalho é uma primeira versão de uma Nota Didática que trata de AM. Assim, é esperada uma contribuição das pessoas que usem este trabalho no sentido de corrigí-lo e melhorá-lo até chegar a sua versão final.

Como já mencionado, um sistema de aprendizado supervisionado é um programa capaz de realizar decisões baseado na experiência contida em casos resolvidos com sucesso. As primeiras perguntas que surgem são

> *Como analisar as regras induzidas por um sistema de AM?*
>
> *Como determinar quando um sistema de AM é superior a outro? etc.*

As respostas a essas indagações, a serem tratadas em uma Nota Didática posterior, são de fundamental importância e são delineadas brevemente a seguir.

As regras de classificação induzidas por um sistema de aprendizado podem ser analisadas segundo dois critérios: a complexidade dessas regras e o erro de classificação sobre um conjunto independente de exemplos.

Muitas vezes, complexidade e taxa de erro na classificação de novos casos estão diretamente relacionadas. Regras de classificação demasiadamente complexas podem indicar que o conhecimento contido nos exemplos não foi suficientemente generalizado. Sistemas de aprendizado utilizam diferentes métodos para criar regras de classificação simples.

Os sistemas de aprendizado simbólico proposicionais, como os tratados neste trabalho, podem ser caracterizados pelo uso da linguagem atributo-valor para descrever os exemplos. Esse tipo de sistemas possuem teoria e métodos bem definidos e, frequentemente,

utilizam árvores ou regras de decisão para descrever os conceitos induzidos. Nesses sistemas é comum a aplicação de técnicas que buscam evitar que os conceitos induzidos sejam demasiadamente complexos. Uma das técnicas mais conhecidas para evitar esse problema chama-se poda. Utilizando poda procura-se eliminar nós — em árvores de decisão — ou condições — em regras de decisão — pouco relevantes ao conceito induzido.

Os sistemas de aprendizado proposicionais, apesar de amplamente difundidos, possuem severas restrições quanto aos conceitos que podem ser induzidos. A expressividade da linguagem atributo-valor permite que somente conceitos que podem ser expressos na linguagem lógica proposicional possam ser aprendidos. As limitações mais severas dos sistemas proposicionais, como a utilização de conhecimento de fundo e a possibilidade de expressar relações entre objetos, só podem ser superadas quando utilizadas linguagens mais expressiva, tal como a lógica de primeira ordem.

Os sistemas de aprendizado relacionais, tais como os sistemas de Programação Lógica Indutiva [Flach 94, Lavrač 94, Lavrač 95, Muggleton 94], utilizam a lógica de primeira ordem como linguagem de descrição de exemplos, hipóteses e conhecimento de fundo. Esta é uma área de pesquisa ativa em Aprendizado de Máquina, e grande parte de sua teoria e métodos ainda estão sendo formulados.

Um dos principais problemas que envolve o aprendizado relacional reside na alta expressividade da lógica de primeira ordem, o que implica em um espaço de hipóteses muito grande. Assim, muitos dos sistemas de aprendizado relacional buscam restringir tal expressividade a fim de tornar o aprendizado factível.

10.1 Erro de Classificação

Além da complexidade dos conceitos induzidos, os sistemas de aprendizado podem ser avaliados pelo erro de classificação em novos casos. Neste caso estudos experimentais são necessários, uma vez que não existe uma análise matemática que possa determinar se um algoritmo de aprendizado irá ter um bom desempenho em um conjunto de exemplos.

Entre as análises matemáticas que podem prover resultados de aplicação prática, encontra-se o modelo PAC – Probably Approximately Correct [Valiant 84] — provido pela teoria de aprendizado computacional [Kearns 94]. PAC é uma análise de pior caso. Independentemente da distribuição de probabilidade do conjunto de exemplos, PAC garante que os resultados de classificação serão corretos com uma pequena margem de erro. Apesar dessa análise prover resultados teóricos interessantes, mesmo para classificadores simples, os resultados podem indicar que um grande número de casos são necessários para garantir o desempenho estipulado.

Técnicas experimentais para estimar a taxa de erro verdadeira não somente provêem uma base para comparar objetivamente a performance de diversos algoritmos de aprendizado no mesmo conjunto de exemplos, mas também podem ser uma ferramenta poderosa para projetar um classificador. Métodos estatísticos são essenciais para uma

análise experimental não tendensiosa [Dietterich 97, Feelders 95, Flexer 96, Kibler 88, Michie 94, Salzberg 97]. Entre os métodos estatísticos mais conhecidos estão os métodos de resampling [Weiss 91], os quais são capazes de estimar a taxa de erro verdadeira de forma confiável mesmo para pequenas amostras de casos resolvidos. Os métodos de resampling são assim chamados pois consistem de repetidos experimentos de amostragem, treinamento e teste. Em [Batista 97] é descrito um ambiente computacional cujo objetivo é auxiliar na aplicação dos métodos de resampling a conjuntos de exemplos reais.

10.2 Aprendizado de Máquina na Web

Existem diversos sites na Web relacionados a Aprendizado de Máquina. Diversas implementações de algoritmos, bem como conjuntos de dados para analisar o comportamento desses algoritmos também podem ser encontrados.

A seguir são referenciados alguns sites que, na nossa opinião, destacam-se pela sua abrangência, apresentação e links para vários outros sites com informações relevantes.

Um primeiro site recomendado é o relacionado ao excelente livro sobre Inteligência Artificial de Russell e Norvig [Russell 95]. Neste site encontram-se, muito bem organizadas, informações sobre os temas tratados no livro, entre elas Aprendizado de Máquina.

- Prof. Stuart J. Russell
  http://www.cs.berkeley.edu/~russell/aima.html

Um outro pesquisador que mantém uma grande lista de links para outros sites na Internet, relacionados com Aprendizado de Máquina é o Prof. David W. Aha.

- David W. Aha
  http://www.aic.nrl.navy.mil/~aha/research/machine-learning.html

Existem também diversas páginas de grupos de pesquisa em Aprendizado de Máquina, tais como

- University of California at Irvine
  http://www.ics.uci.edu/AI/ML/Machine-Learning.html
- Oxford University Computing Laboratory
  http://www.comlab.ox.ac.uk/oucl/groups/machlearn/
- Knowledge Systems Laboratory of the National Research Council of Canada
  http://ai.iit.nrc.ca/home_page.html
  http://ai.iit.nrc.ca/bibliographies/
- Austrian Research Institute for Artificial Intelligence
  http://www.ai.univie.ac.at/oefai/ml-resources.html
- Carnegie Mellon University
  http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/learning/0.html
- German National Research Center for Information Technology
  http://www.gmd.de/ml-archive
- AI, CogSci and Robotics: Artificial Intelligence
  http://www-mice.cs.ucl.ac.uk/misc/ai/ai.html

- Machine learning at The University of Wisconsin
  http://www.cs.wisc.edu/~shavlik/uwml.html
- Machine Learning Library in C++
  http://www.sgi.com/Technology/mlc/
- Machine Learning Project
  http://www.cs.waikato.ac.nz/ml/
- The KAML Group, University of Ottawa
  http://www.csi.uottawa.ca/dept/kaml/KAML.html

Por fim, algumas páginas com outros recursos a respeito de Aprendizado de Máquina

- OFAI Library Information System Biblio. Este é um sistema interessante, pois permite que sejam feitas buscas de artigos, livros e relatórios técnicos da área de Inteligência Artificial por nome de autor e/ou título. O sistema apresenta links para alguns trabalhos que estão on-line
  http://www.ai.univie.ac.at/biblio.html
- Journal of Artificial Intelligence Research
  http://www.cs.washington.edu/research/jair/home.html
- Computer Science Technical Reports Archive Sites. Uma lista bastante completa de endereços eletrônicos de repositórios de relatórios técnicos de inúmeras universidades
  http://www.dai.ed.ac.uk/research/siteslist.html
- Machine learning course pages around the country
  http://www.cs.iastate.edu/~honavar/Courses/cs673/machine-learning-courses.html
- Yahoo Machine Learning
  http://www.yahoo.com/Science/Computer_Science/Artificial_Intelligence/Machine_Learning/
- Machine Learning Online
  http://mlis.www.wkap.nl/ml_links.htm

Entre as listas de Aprendizado de Máquina, destaca-se a lista moderada pelo Prof. M. Pazzani da University of California at Irvine. A freqüência da lista é de aproximadamente 2 semanas. Pedido de inscrição deve ser feito para ml-request@ics.uci.edu enquanto que contribuições devem ser enviadas para ml@ics.uci.edu

Referências

[Aha 91] Aha, D.W.; Kibler. D.; Albert, M. K. *Instance-based Learning Algorithms.* Machine Learning 6, 1, pp. 37–66, 1991.

[Arariboia 89] Grupo Arariboia. *Inteligência Artificial: um Curso Prático.* Editora LTC, Rio de Janeiro, 1989.

[Anderson 79] Anderson, J.R.; Kline, P.J. *A Learning System and its Psychological Implications.* Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokio, Japão, pp. 16–21, 1979.

[Anderson 83] Anderson, J.R. *Acquisition of Proof Skills in Geometry.* Machine Learning: An Artificial Intelligence Approach, R.S. Michalski & J.G. Carbonell & T.M. Mitchell (Ed.), Tioga Publishing Company, CA, pp. 191–220, 1983.

[Arity 90] Arity Corporation. *The Arity/Prolog Programming Language.* 1990.

[Batista 97] Batista, G.E.A.P.A. *Um Ambiente de Avaliação de Algoritmos de Aprendizado de Máquina Utilizando Exemplos.* Dissertações de Mestrado, ICMSC-USP, 1997. *(A ser defendida).*

[Bratko 89] Bratko, I. *Machine Learning.* In Human and Machine Problem Solving, K. Gilhosly, Academic Press, pp. 265–87, 1989.

[Bratko 90] Bratko, I. *Prolog Programming for Artificial Intelligence.* (2^a Ed.) Addison-Wesley, 1990.

[Breiman 84] Breiman, L.; Fredman, J.H.; Olshen, R.A; Stone, C.J. *Classification and Regression Trees.* Belmont: Wadsworth, 1984.

[Carbonell 83] Carbonell, J.G.; Langley P. *Learning by Analogy.* Machine Learning: An Artificial Intelligence Approach, R.S. Michalski & J.G. Carbonell & T.M. Mitchell (Ed.), Tioga Publishing Company, CA, pp. 137–162, 1983.

[Carbonell 87] Carbonell, J.G.; Langley P. *Machine Learning.* Encyclopedia of Artificial Intelligence, S.C. Shapiro (Ed.), John Wiley & Sons, U.S.A., pp. 464–88, 1987.

[Castineira 90] Castineira, M.I. *Aprendizado de Máquina por Exemplos Usando Árvores de Decisão.* Dissertação de Mestrado, ICMSC-USP, 1990.

[DeJong 86] DeJong,G.; Mooney,R.J. *Explanation-Based Learning: An Alternative View.* Machine Learning 2, Vol. 1, pp. 145–176, 1986.

[De Jong 88] De Jong, K. *Learning with Genetic Algorithms: An Overview.* Machine Learning, Nº 3, pp. 121–138, 1988.

[Dietterich 97] Dietterich, T.G. *Statistical Tests for Comparing Supervised Classification Learning Algorithms.* In Press, 1997.
ftp://ftp.cs.orst.edu/pub/tgd/papers/stats.ps.gz

[Feelders 95] Feelders, A.; Verkooijen, W. *Which Method Learns Most from the Data? Methodological Issues an the Analysis of Comparative Studies.* Prelimirary Proceedings of the 5th International Workshop on AI ans Statistics, pp. 219–225, 1995.
http://diamond.ddi.nl/*sim*ad/docs.aistats95.ps

[Flach 94] Flach, P. *Simply Logical. Intelligent Reasoning by Example.* Wiley, 1994.

[Flexer 96] Flexer, A. *Statistical Evaluation of Neural Networks Experiments: Minimum Requirements and Current Practice.* Proccedings of the 13th European Meeting on Cybernetics and Systems Research, 2, pp. 1005–1008, 1996.
ftp://ftp.ai.univie.ac.at/papers/oefai-tr-95-06.ps.Z

[Freitas 92] Freitas, A.A.; Kirner, C. *Introdução a Algoritmos Genéticos.* Anais do I Workshop sobre Redes Neurais, UFSCar, São Carlos, pp. 71–88, 1992.

[Haykin-94] Haykin, S. *Neural Networks A Comprehensive Foundation.* Macmillan College Publishing Company, Inc., 1994.

[Hogger 90] Hogger, C.J. *Essentials of Logic Programming.* Oxford University Press, 1990.

[Holland 86] Holland, J.H. *Escaping Brittleness: The Possibilities os General-purpose Learning Algorithms Applied to Parallel Rule-based Systems.* Machine Learning: An Artificial Intelligence Approach (Vol. 2) Michalski, R.S.; Carbonell, J.G.; Mitchell, T.M. (Eds.), Morgan Kaufmann, 1986.

[Hopfield 82] Hopfield, J.J. *Neural Networks and Physical Systems with Emergent Collective Computational Abilities.* Proceedings of the National Academy of Sciences of the U.S.A., 81, pp. 3088–3092, 1982.

[Kearns 94] Kearns, M.J.; Vazirani, U.V. *An Introduction to Computational Learning Theory.* The MIT press, 1994.

[Kibler 88] Kibler, D. Langley, P. *Machine Learning as a Experimental Science.* Machine Learning, 3(1), pp. 5–8, 1988.

[Kowalsky 79] Kowalsly, R. *Logic for Problem Solving.* Artificial Intelligence Series Nº 7. North-Holland, 1979.

[Lavrač 94] Lavrač, N.; Džeroski, S. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, London, 1994.

[Lavrač 95] Lavrač, N.; Raedt, L. *Inductive Logic Programming: A Survey of European Research.* AI Communications 8(1), pp. 3–19, 1995.

[Marcus 86] Marcus, C. *Prolog Programming: Aplications for Database System, Expert Systems and Natural Languages System.* Addison-Wesley, 1986.

[Martins 94] Martins, C.A.M. *Uso de Árvores de Decisão na Geração de Hipóteses Indutivas em Aprendizado de Máquina.* Dissertação de Mestrado, ICMSC-USP, 1994.

[McCulloch 43] McCulloch, W.S.; Pitts, W. *A Logical Calculus of the Ideas Immanet in Nervous Activity.* Bulletin of Mathematical Biophysics, Nº 5, pp. 115–133, 1943.

[Michalski 83] Michalski, R.S.; Carbonell, J.G.; Mitchell, T.M. (Eds.) *Machine Learning, An Artificial Intelligence Approach.* Tioga Publishing Company, Palo Alto, California, 1983.

[Michalski 86] Michalski, R.S.; Mozetic, I.; Hong, J.; Lavrac, N. *The Multi-purpose Incremental Learning System AQ15 and Its Testing Application to Three Medical Domains.* Proceedings of the Fifth Annual National Conference on Artificial Intelligence, pp. 1041–1045, 1986.

[Michalski 87] Michalski, R.S. *Concept Learning.* Encyclopedia of Artificial Intelligence, S.C. Shapiro (Ed.), John Willey & Sons, U.S.A., pp. 185–194, 1987.

[Michie 94] Michie, D.; Spiegelhalter D.J.; Taylor C.C. (Eds.). *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[Minsky 69] Minsky, M.L.; Papert, S.A. *Perceptrons.* MIT Press, Cambridge, 1969.

[Mitchell 82] Mitchell, T.M. *Generalization as Search.* Artificial Intelligence, Nº 18, pp. 203–226, 1982.

[Monard 93a] Monard, M.C.; Nicoletti, M.C. *Programas Prolog para Processamento de Listas e Aplicações.* Notas Didáticas do ICMSC-USP, ISSN 0103-2585, Nº 7, 71 pg, 1993.

[Monard 93b] Monard, M.C.; Nicoletti, M.C. *Técnicas Avançadas de Programação Prolog para Tratamento de Árvores.* Notas Didáticas do ICMSC-USP, ISSN 0103-2585, Nº 8, 51 pg, 1993.

[Morgan 73] Morgan, J.; Messenger, R. *THAID: A Sequential Search Program for the Analysis of Nominal Scale Dependent Variables.* Technical report, Institute for Social Research, University of Michigan, 1973.

[Muggleton-90] Muggleton, S.H.; Feng, C. *Efficient Induction of Logic Programs.* TIRM-90-044, Turing Institute Press, Glasgow, October, 1990.

[Muggleton 94] Muggleton, S.; Raedt L. *Inductive Logic Programming: Theory and Methods.* The Journal of Logic Programming 19, pp. 629–679, 1994. ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Papers/lpj.ps

[O'Keefe 90] O'Keefe, R.A. *The Craft of Prolog.* The MIT Press, 1990.

[Quinlan 79] Quinlan, J.R. *Discovering Rules by Induction from Large Collections of Examples.* Expert Systems in the Micro Electronic Age, D. Michie (Ed.), Edinburgh University Press, Edinburgh, pp. 168–201, 1979.

[Quinlan 86] Quinlan, J.R. *Induction of Decision Trees.* Machine Learning, № 1, pp. 81–106, 1986.

[Quinlan 86b] Quinlan, J.R. *The Effect of Noise on Concept Learning.* R.S. Michalski & J.G. Carbonell & T.M. Mitchell (Ed.), Machine Learning: An Artificial Intelligence Approach, San Mateo, CA: Morgan Kaufmann, Vol. 2, 1986.

[Quinlan 87a] Quinlan, J.R. *Generating Production Rules from Decision Trees.* Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy: Morgan Kaufmann, pp. 304–307, 1987.

[Quinlan 87b] Quinlan, J.R. *Simplifying Decision Trees.* International Journal of Man-Machine Studies, № 27, pp. 221–234, 1987.

[Quinlan 88] Quinlan, J.R. *C4.5 Programs for Machine Learning.* Morgan Kaufmann Publishers, CA, 1988.

[Raggett 92] Raggett, J.; Bains, W. *Artificial Intelligence from A to Z.* Chapman & Hall, 1992.

[Rich 93] Rich, E.; Knight, E. *Inteligência Artificial.*, (2^a Ed.) Makron Books do Brasil, 1993.

[Rosenblatt 58] Rosenblatt, F. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain.* Psycological Review, № 65, pp. 386–408, 1958.

[Rowe 88] Rowe, N.C. *Artificial Intelligence through Prolog.* Prentice Hall, 1988.

[Rumelhart 86] Rumelhart, D.E.; McClelland, J.L. *Parallel Distributed Processing: Explorations in the Microstruture of Cognition.* Vol. 1, MIT Press, 1986.

[Russell 95] Russell, S.J.; Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall International, 1995.

[Salzberg 97] Salzberg, S.L. *On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach.* Data Mining and Knowledge Discovery, 1, pp. 1–11, 1997.
http://www.cs.jhu.edu/~salzberg/critique.ps

[Shaw 90] Shaw, M.J.; Gentry, J.A. *Inductive Learning for Risk Classification.* University of Illinois at Urbana-Champaign, IEEE, pp. 47–53, February, 1990.

[Shoham 94] Shoham, Y. *Artificial Intelligence Techniques in Prolog.* Morgan Kaufman, 1994.

[Simon 83] Simon, H.A. *Search and Reasoning in Problem Solving.* Artificial Intelligence, n. 21, pp. 7–30, 1983.

[Stanfill 86] Stanfill, C.; Waltz, D. *Toward Memory-based Reasoning.* Communications of the ACM 29, 12, pp. 1213–1228, 1986.

[Sterling 86] Sterling, L.; Shapiro, E. *The Art of Prolog.* MIT Press Series in Logic Programming, E. Shapiro (Ed.), 1986.

[Utgoff 89] Utgoff, P.F. *Incremental Induction of Decision Trees.* Machine Learning, Nº 4, pp. 161–186, 1989.

[Valiant 84] Valiant, L.G. *A Theory of the Learning.* Communications of the ACM, Nº 27, pp. 1134–1142, 1984.

[Weiss 91] Weiss, S.M.; Kulikowski, C.A. *Computer Systems That Learn. Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufmann Publishers, CA, 1991.

[Winston 75] Winston, P.H. *Learning Structural Descriptions from Examples.* The Psychology of Computer Vision, P.H. Winston (Ed.), McGraw-Hill, New York, pp. 157–209, 1975.

[Winston 92] Winston, P.H. *Artificial Inteligence.* Addison-Wesley (3^{ra} Ed.), 1992.