UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE SISTEMAS DE INFORMAÇÃO

USO DE SELETOR DE ATRIBUTOS UTILIZANDO METODOLOGIA EM FILTRO PARA CLASSIFICAÇÃO DE BASES DE DADOS PROPOSICIONALIZADAS DO PROBLEMA DE PROGRAMAÇÃO EM LÓGICA INDUTIVA

Felipe Costa Rodrigues

Diamantina
2014

UNIVERSIDADE FEDERAL DOS VALES DO JEQUITINHONHA E MUCURI

DEPARTAMENTO DE COMPUTAÇÃO

CURSO DE SISTEMAS DE INFORMAÇÃO

USO DE SELETOR DE ATRIBUTOS UTILIZANDO METODOLOGIA EM FILTRO PARA CLASSIFICAÇÃO DE BASES DE DADOS PROPOSICIONALIZADAS DO PROBLEMA DE PROGRAMAÇÃO EM LÓGICA INDUTIVA

Felipe Costa Rodrigues

Orientador (a):

Cristiano Grijó Pitangui

Trabalho de Conclusão de Curso apresentado ao Curso de Sistemas de Informação, como parte dos requisitos exigidos para a conclusão do curso.

Diamantina
2014

Monografia de projeto final de graduação sob o título "Uso de Seletor de Atributos Utilizando Metodologia em Filtro para Classificação de Bases de Dados Proposicionalizadas do Problema de Programação em Lógica Indutiva", defendida por Felipe Costa Rodrigues e aprovada em 9 de julho de 2014, em Diamantina, Minas Gerais.

Banca Examinadora:

Prof. Dr. Cristiano Grijó Pitangui
Orientador

Prof. Dr. Alessandro Vivas Andrade

Profª. Drª. Luciana Pereira de Assis

AGRADECIMENTOS

A Deus por ter me dado saúde e força para superar as dificuldades. A esta universidade, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro um horizonte superior, eivado pela acendrada confiança no mérito e ética aqui presentes.

Ao meu orientador Cristiano, pelo suporte no pouco tempo que lhe coube, pelas suas correções e incentivos.

À minha mãe e irmã, pelo amor, incentivo e apoio incondicional.

E a todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

RESUMO

Rodrigues, F. C. Uso de seletor de atributos utilizando metodologia em filtro para classificação de bases de dados proposicionalizadas do problema de programação em lógica indutiva. **Monografia (Bacharelado em Sistemas de Informação). Universidade Federal dos Vales do Jequitinhonha e Mucuri. Diamantina. Minas Gerais.**

A Programação em Lógica Indutiva (ILP) tem os seus objetivos baseados no Aprendizado Indutivo e na Programação em Lógica. Ela é definida como a interseção entre Aprendizado de Máquina e Programação em Lógica, e, com isso, emprega técnicas de ambas as áreas. A aplicação da ILP pode ser vista em vários problemas importantes como: dimensionamento de malhas em métodos de elementos finitos, previsão de estrutura de proteínas, detecção de problemas de tráfego, processamento de linguagem natural, e outros.

Atualmente existem muitos sistemas ILP que aprendem através de um conhecimento preliminar e, com isso, são capazes de classificar exemplos.

Nesse trabalho apresentaremos um sistema ILP denominado PROP-AG. O sistema trabalha com proposicionalização, isto é, ele transforma uma base de dados representada lógica de primeira-ordem para representação em lógica proposicional. Posteriormente, o sistema usa a medida de ganho de informação para filtrar os melhores atributos da base de dados proposicionalizada, aplica um Algoritmo Genético para buscar por uma "boa" combinação de atributos, e, finalmente, usa uma Árvore de Decisão (o algoritmo C4.5) para construir o modelo de classificação. Resultados experimentais mostram que o PROP-GA encontra soluções competitivas aos sistemas considerados estado da arte.

Palavras-chave: Inteligência Artificial, aprendizado de máquina, Proposicionalização, Classificação, Árvore de Decisão.

ABSTRACT

Rodrigues, F. C. Use the attribute selector using methodology filter for classification of data bases proposicionalizadas problem in inductive logic programming. **Monoghaph (Bachelorship of Information Systems). Federal University of the Jequitinhonha and Mucuri Valleys. Diamantina. Minas Gerais.**

Inductive Logic Programming (ILP) has its goals based Learning and Inductive Logic Programming. It's defined as the intersection of Machine Learning and Logic Programming and therefore employs techniques from both areas. The application of ILP can be seen in several important problems such as scaling meshes in finite element methods, protein structure prediction, detection of traffic problems, natural language processing, and others.

Currently there are many ILP systems to learn through a preliminary knowledge and are able to classify examples.

In this work we present an ILP system called PROP-AG. The system works with propositionalization, that's, the transformation of a database represented in first-order logic to a new database represented in propositional logic.

Subsequently, the system uses the information gain measure to filter the best attributes of the base propositionalized data, apply a Genetic Algorithm to search for a "good" combination of attributes, and finally uses a decision tree (C4.5 algorithm) to construct the classification model. Experimental results show that the PROP-AG find competitive solutions to the systems considered state of the art.

Keywords: Artificial Intelligence, Learning Machine, Propositionalization, Classification, Decision Tree

SUMÁRIO

1 INTRODUÇÃO

1.1 Introdução

A Programação em Lógica Indutiva (ILP) é uma subárea do Aprendizado de Máquina (AM) (*Machine Learning*) que por sua vez, compõe a Inteligência artificial. A ILP investiga a construção indutiva de teorias de cláusulas de Horn, de primeira-ordem, a partir de exemplos e de um conhecimento preliminar (MUGGLETON, 2006). A ILP tem os seus objetivos baseados no aprendizado indutivo e na programação lógica. Ela é definida como a interseção entre Aprendizado de Máquina e Programação em Lógica (MUGGLETON, 1991), por isso, a ILP emprega tanto técnicas do Aprendizado de Máquina quanto da Programação em Lógica. A Programação em Lógica Indutiva está centrada no aprendizado indutivo que expressa uma forma de raciocínio que parte do específico para o geral.

A aplicação da ILP pode ser vista em vários problemas importantes como: dimensionamento de malhas em métodos de elementos finitos, previsão de estrutura de proteínas, detecção de problemas de tráfego, processamento de linguagem natural, e outros.

Para este trabalho aplicaremos a técnica de proposicionalização de uma base de dados em Lógica de Primeira-Ordem (LPO). Essa é uma técnica na qual uma base de dados que se encontra em LPO é transformada em uma base de dados em Lógica Proposicional (FRANÇA; ZAVERUCHA, 2013).

Após a proposicionalização das bases de dados encontrou-se um problema: para algumas bases de dados o número de atributos era grande demais (mais de 50 mil atributos) para que pudesse ser encontrada uma boa solução em tempo razoável. Para resolver esse problema, foi proposta a implementação de um método que reduziria a quantidade de atributos da base, filtrando os melhores. Foram aplicadas as técnicas de Entropia e o Ganho de Informação para se selecionar os melhores atributos. Definidos os atributos, um Algoritmo Genético (AG) é aplicado para buscar uma combinação destes possíveis atributos.

1.2 Objetivos

1.2.1 Objetivo Geral

Este trabalho trata do problema de Programação em Lógica Indutiva. Primeiramente, usa-se o método de proposicionalização para transformar uma base de dados em Lógica de Primeira-Ordem para Lógica Proposicional. Posteriormente, usa-se um algoritmo de árvore de decisão para solucionar o problema enquanto um Algoritmo Genético busca pela combinação dos melhores atributos da base de dados proposicionalizada.

1.2.2 Objetivos específicos:

1- Implementar um método de proposicionalização juntamente com um método de classificação.

2- Analisar estatisticamente os resultados obtidos.

3- Comparar a análise estatística do método proposto com outros sistemas ILP.

1.3 Organização do Trabalho

- **O Capítulo 2** apresenta os principais conceitos sobre aprendizado de máquina e seus ramos.
- **O Capítulo 3** apresenta uma revisão sobre ILP; aqui é mostrado o funcionamento de alguns sistemas ILP com finalidade de compará-los com o sistema proposto.
- **O Capítulo 4** apresenta o sistema proposto, chamado de PROP-AG. Seus principais componentes e funcionalidades são apresentados e detalhadamente discutidos.

- **O Capítulo 5** apresenta os resultados experimentais e discussões da aplicação do sistema proposto.
- **O Capítulo 6** apresenta as conclusões obtidas a partir dos experimentos computacionais, bem como propostas de trabalhos futuros.

2 APRENDIZADO DE MÁQUINA

2.1 Aprendizado e o Aprendizado de Máquina

O aprendizado é a capacidade de se adaptar, de modificar e melhorar seu comportamento e suas respostas, sendo, portanto, uma das propriedades mais importantes dos seres ditos inteligentes.

Aprendizado de Máquina (AM) é uma subárea de pesquisa muito importante em Inteligência Artificial, pois a capacidade de aprender é essencial para um comportamento inteligente. O AM estuda métodos computacionais para adquirir novos conhecimentos, novas habilidades e novos meios de organizar o conhecimento já existente (MITCHELL, 1997). O estudo de técnicas de aprendizado baseado em computador também pode fornecer um melhor entendimento de nosso próprio processo de raciocínio.

Segundo Batista (BATISTA, 2003), uma das críticas mais comuns a 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. Ada Augusta, uma das primeiras filósofas em computação escreveu: "A máquina analítica não tem qualquer pretensão de originar nada. Ela pode fazer qualquer coisa desde que nós saibamos como mandá-la executar".

Vários críticos de IA interpretaram esse comentário como uma indicação de que os computadores não são capazes de aprender. Entretanto, nada impede que programemos um computador para interpretar informações recebidas, de tal forma que melhore gradualmente seu desempenho.

Existem várias abordagens de aprendizado que podem ser utilizadas por um sistema computacional como, por exemplo, o aprendizado por hábito, por instrução, por dedução, por analogia e por indução. O aprendizado indutivo é um dos mais úteis, pois permite obter novos conhecimentos a partir de exemplos, ou casos, particulares previamente observados. No entanto, é também um dos mais desafiadores, pois o

conhecimento gerado ultrapassa os limites das premissas, e não existem garantias de que esse conhecimento seja verdadeiro.

2.2 Aprendizado Indutivo

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, ou seja, as hipóteses levam a conclusões cujos conteúdos excedem os das premissas. É esse traço característico da indução que torna os argumentos indutivos indispensáveis para a fundamentação de uma significativa porção dos nossos conhecimentos (BATISTA, 2003).

Ao contrário do que acontece com um argumento dedutivo e válido, um argumento indutivo e correto pode, perfeitamente, admitir uma conclusão falsa, ainda que suas premissas sejam verdadeiras. 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 indução matemática. Em um argumento matemático indutivo correto, partindo de premissas verdadeiras obtêm-se, invariavelmente, conclusões verdadeiras.

Segundo Batista (BATISTA, 2003), 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-se que seja relevante, pode nos capacitar a determinar, com maior precisão se a conclusão é verdadeira. A evidência adicional pode afetar o grau de sustentação da conclusão.

A inferência indutiva é um dos principais meios de criar novos conhecimentos e prever eventos futuros. O processo de indução é indispensável na obtenção de novos conhecimentos pelo ser humano. Foi por meio 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.

2.3 Aprendizado de Máquina Indutivo por Exemplos

O aprendizado indutivo é efetuado a partir do raciocínio sobre exemplos fornecidos por um processo externo ao aprendiz. Em AM, o aprendiz é um sistema computacional frequentemente denotado por sistema de aprendizado, algoritmo de aprendizado, ou simplesmente indutor. Um sistema de aprendizado é um sistema computacional que toma decisões baseado em experiências acumuladas contidas em casos resolvidos com sucesso (BATISTA apud WEISS & KULIKOWSKI, 2003).

O aprendizado indutivo por exemplos pode ser dividido em aprendizado supervisionado e não supervisionado.

No aprendizado supervisionado, o objetivo é induzir conceitos a partir de exemplos que estão pré-classificados, ou seja, exemplos rotulados com uma classe conhecida. Desse modo, pode-se dizer que o aprendizado humano é baseado em experiências passadas. Já um sistema computacional aprende de dados que representam alguma "experiência passada" de um domínio de aplicação. Nesse aprendizado, é fornecido ao sistema de aprendizado um conjunto de exemplos $E = \{E1, E2, \ldots, En\}$, sendo que cada exemplo $Ei \in E$ é uma tupla

$$Ei = (xi, yi) \tag{2.1}$$

Onde x_i é um vetor de valores que representam as característica, ou atributos do exemplo E_i, e y_i é o valor da classe desse exemplo.

O objetivo do aprendizado supervisionado é induzir um mapeamento geral dos vetores *x* para valores *y*. Portanto, o sistema de aprendizado deve construir um modelo, $y = f(x)$, de uma função desconhecida, *f*, também chamada de função conceito, que permite predizer valores *y* para exemplos previamente não vistos.

Entretanto, o número de exemplos utilizados para a criação do modelo não é, na maioria dos casos, suficiente para caracterizar completamente essa função *f*. Na

realidade, sistemas de aprendizado são capazes de induzir uma função *h* que aproxima *f*, ou seja, $h(x) \approx f(x)$. Nesse caso, *h* é chamada de hipótese sobre a função conceito *f*.

No aprendizado não supervisionado é fornecido ao sistema de aprendizado um conjunto de exemplos E, no qual cada exemplo consiste somente de vetores *x*, não incluído a informação sobre a classe *y*. O objetivo do aprendizado não supervisionado é construir um modelo que procura por regularidades nos exemplos, formando agrupamentos ou *clusteres* de exemplos com características similares (RUSSEL e NOVING, 2003).

No primeiro caso, onde o objetivo é encontrar uma função *h* que se aproxima de uma função *f*, a hipótese *h* é denominada classificador, e a tarefa de aprendizado é denotado classificação. No segundo caso no qual o objetivo do aprendizado é tentar separar os exemplos por similaridade, a hipótese *h* é denominada regressor, e a tarefa de aprendizado é denotada regressão. A Figura 2.1 apresenta a hierarquia do aprendizado. Este trabalho usa técnicas de classificação para obter hipóteses.

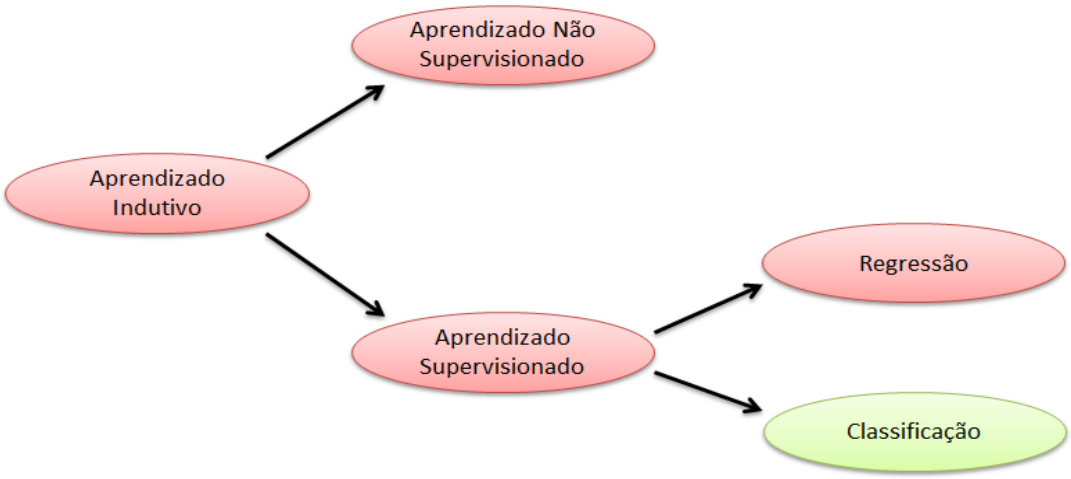

Figura 2.1 Hierarquia do Aprendizado

2.4 Classificação

A classificação é uma tarefa de mineração de dados, na qual os algoritmos buscam por padrões que classifiquem os dados, ou seja, por classificadores. No contexto de classificadores, um exemplo ou instância, é um conjunto ordenado de atributos, descrevendo um objeto de interesse. Cada atributo, também chamado de variável ou característica, possui valores de um subconjunto pré-definido de valores que são dependentes do problema, e que descrevem algum aspecto do exemplo. Este conjunto pré-definido de valores é conhecido como domínio de variável (HALMENSCHLAGER, 2002).

Um conjunto de dados usado na mineração é formado por um conjunto de instâncias, que possui *n* instâncias e *m* atributos. O conjunto de treinamento é um conjunto de instâncias rotuladas e o conjunto de teste é um conjunto de instâncias não rotuladas. O conjunto de teste também pode ser um conjunto de instâncias rotuladas, que fazem parte do conjunto de instâncias, usado para estimar a qualidade do classificador.

Ao algoritmo de indução é fornecido um conjunto de treinamento. A tarefa do algoritmo é gerar um bom classificador a partir deste conjunto de instâncias classificadas. O classificador pode então ser usado para classificar instâncias não rotuladas, fazendo uma predição correta do rótulo de cada uma dessas novas instâncias. Assim, considera-se a classificação como uma tarefa de previsão, em que um conjunto de atributos previsores (características) é usado para prever um atributo meta (rótulo). A tarefa consiste em descobrir o relacionamento entre os atributos previsores e o atributo meta, usando um conjunto de instâncias cuja classe é previamente conhecida. Após o aprendizado, o objetivo é utilizar o relacionamento descoberto para prever a classe de novos dados, a partir, apenas, de seus atributos previsores (HALMENSCHLAGER, 2002).

O classificador também pode ser avaliado em relação à sua acurácia, compreensibilidade, tempo de aprendizado, requisitos de armazenamento, simplicidade e alguma outra propriedade que defina o quão bom ele é para esta determinada tarefa. Como usaremos o conceito de acurácia ao longo do texto é importante que ele seja

definido. A acurácia é a proporção de instâncias de uma base de dados classificada corretamente pelo classificador.
A acurácia é dada pela seguinte fórmula:

$$Acurácia = \frac{NúmeroDeClassificaçõesCorretas}{NúmeroTotalDeCasosDeTeste} \quad (2.1)$$

2.5 Árvores de Decisão

Foram propostas várias técnicas capazes de aprender a partir de um conjunto de exemplos. Um requisito básico para todas elas é que o conceito a ser aprendido deve estar relacionado com casos observados, isto é, exemplos. E cada exemplo deve estar rotulado com a classe à qual pertence.

Os sistemas de aprendizado simbólico buscam aprender construindo representações simbólicas de um conceito por meio da análise de exemplos e contraexemplos desse conceito. As representações simbólicas estão tipicamente na forma de alguma expressão lógica: árvores de decisão, regras de decisão ou redes semânticas.

As árvores de decisão, ou árvore de classificação, são representações simples do conhecimento e têm sido aplicadas em sistemas de aprendizado. Elas são amplamente utilizadas em algoritmos de classificação, como um meio eficiente para construir classificadores que predizem classes baseadas nos valores de atributos. Assim, podem ser utilizadas em várias aplicações como diagnósticos médicos, análise de risco em créditos, entre outros exemplos.

Uma árvore de decisão usa uma representação baseada em árvores, que é um tipo de estrutura de dados não linear, que possui um número finito de elementos ou nós. Cada árvore possui um único nó especial, chamado nó raiz, colocado no topo da representação gráfica, que é o nó pai de suas subárvores, cujos nós, por sua vez, são os nós filhos do nó raiz. Todos os nós que possuem o mesmo pai são considerados nós irmãos. Os nós que não possuem filhos são considerados nós terminais e são chamados de folhas. Já o nível de um nó é a distância do mesmo até a raiz, ou seja, o número de ligações percorridas até chegar a raiz. A profundidade de uma árvore de decisão é definida pela maior distância entre uma folha e a raiz, existindo árvores com

profundidade uniforme em todas as folhas e outras não. Dependendo do número de filhos de cada nó, a árvore pode ser considerada binária, quando possuir sempre dois filhos em cada nó; ternária, quando possuir três ligações em cada nó; ou mista, quando o número de filhos for variável (RUSSEL e NOVING, 2003).

O método de indução de árvores de decisão a partir de dados empíricos, conhecido como particionamento recursivo, foi estudado por pesquisadores na área de IA e estatística. Os sistemas ID3 (QUINLAN, 1986) e C4.5 (QUINLAN, 1993) para indução de árvores de decisão tiveram uma importante contribuição sobre a pesquisa em IA.

Uma árvore de decisão tem a função de particionar recursivamente um conjunto de treinamento, até que cada subconjunto obtido deste particionamento contenha casos de uma única classe. Para atingir essa meta, a técnica de árvores de decisão examina e compara a construção de uma árvore de decisão, são dados organizados de maneira compacta, que são utilizados para classificar novos casos.

O algoritmo ID3 foi um dos primeiros algoritmos de árvore de decisão, tendo sua elaboração baseada em sistemas de inferência e em conceitos de sistemas de aprendizagem. Ele constrói árvores de decisão a partir de um dado conjunto de exemplos, sendo a árvore resultante usada para classificar amostras futuras. O ID3 separa um conjunto de treinamento em subconjuntos, de forma que estes contenham exemplos de uma única classe. A divisão é efetuada através de um único atributo, que é selecionado a partir de uma propriedade estatística, denominada Ganho de Informação, que mede o grau de informação daquele atributo (QUINLAN, 1986).

O algoritmo C4.5 é um aprimoramento do algoritmo ID3, isto, devido ao fato de trabalhar com valores indisponíveis, com valores contínuos, podar árvores de decisão e derivar regras. Antes de discorrer mais sobre árvores de decisão, especialmente sobre a C4.5 que é usada nesse trabalho, abordaremos conceitos indispensáveis em sua construção, tais como, o Ganho de Informação e a entropia.

2.5.1 Entropia e Ganho de Informação

A entropia de um sistema é uma medida do seu grau de desorganização. Na termodinâmica é a grandeza que permite avaliar a degradação da energia de um sistema:

a entropia de um sistema caracteriza o seu grau de desordem. De forma geral, usa-se essa como medida de impureza ou desordem dos dados *D*. Com o uso da fórmula da entropia será possível verificar se um atributo é relevante ou não. À medida que a base vai se purificando, o valor de entropia vai diminuindo. No contexto das árvores de decisão a entropia é usada para estimar a aleatoriedade da variável a prever (classe) (MITCHELL, 1997).

Dizemos que uma base é pura caso todos os exemplos sejam classificados da mesma forma, no nosso caso, todos os exemplos devem ser classificados como *True* ou *False*. A técnica de entropia foi desenvolvido por Ross Quinlan (QUINLAN, 1993). Esta técnica mede a quantidade de informação necessária para codificar a classe do nó. Para simplificar a explicação, suponha que haja um problema de classificação binária, o atributo alvo pode assumir no máximo dois valores, por exemplo: positivo *p+*, ou negativo *p-*. Seja *D* um conjunto de exemplos de algum conceito a ser aprendido. Então, a entropia de *D* em relação ao atributo alvo é dada pela Equação 2.2.

$$entropia(D) = -\sum_{i=1}^{|C|} - pi \log_2 pi \qquad (2.2)$$

onde *D* é o conjunto de exemplos e *pi* é a probabilidade da classe *i* ocorrer. Nota-se que, a entropia torna-se 0 (zero) caso todos os exemplos pertençam à mesma classe, e a entropia máxima é igual a 1 quando cada classe possuir número igual de exemplos.

Exemplo 2.1: Aqui ilustraremos os valores de entropias de bases de dados com certas proporções de exemplos positivos e negativos.

1. A base de dados *D* tem 50% de exemplos positivos *(Pr(positivo) = 0,5)* e 50% de exemplos *negativos (Pr(negativos) = 0,5).*
   $entropia(D) = -0{,}5 \times \log_2 0{,}5 - 0{,}5 \times \log_2 0{,}5 = 1$

2. A base de dados *D* tem 20% de exemplos positivos *(Pr(positivo) = 0,2)* e 80% de exemplos *negativos (Pr(negativos) = 0,8).*
   $entropia(D) = -0{,}2 \times \log_2 0{,}2 - 0{,}8 \times \log_2 0{,}8 = 0{,}722$

3. A base de dados *D* tem 100% de exemplos positivos *(Pr(positivo) =1,0)* e 0% de exemplos *negativos (Pr(negativos) = 0,0).*

$$entropia(D) = -1{,}0 \times \log_2 1{,}0 - 0{,}0 \times \log_2 0{,}0 = 0$$

A partir do conceito de entropia, define-se o Ganho de Informação como a redução esperada da entropia causada quando se observa atentamente a informação sobre um atributo. Formalmente, sejam: *A* um atributo presente nos exemplos do conjunto *D*; *dom(A)* o conjunto de valores que *A* pode assumir, e D_v o subconjunto dos elementos de *D* em que *A* possui valor *v*, com *v* pertencente ao *dom(A)*. Então, o Ganho de Informação do atributo *A* em relação ao conjunto *D* é dado pela equação 2.3

$$Ganho(D, A) = Entropia(D) - \sum_{v=1}^{n} \frac{|Dv|}{|D|} \times entropia(D_j) \quad (2.3)$$

Exemplo 2.2: A Tabela 2.1 representa os clientes de uma firma financeira. Uma firma financeira recebe geralmente inúmeros pedidos de empréstimos. Cada almejante ao empréstimo possui uma série de características. O problema constitui na decisão de conceder o empréstimo, ou seja, discriminar os clientes entre pagadores e não pagadores. O Ganho de Informação nos mostra qual é o atributo mais valoroso quando precisamos tomar essa decisão.

Tabela 2.1: Empréstimos

	Idade	**Emprego**	**Casa Própria**	**Crédito**	**Classe**
1	Jovem	Falso	Falso	Ruim	Falso
2	Jovem	Falso	Falso	Bom	Falso
3	Jovem	Verdadeiro	Falso	Bom	Verdadeiro
4	Jovem	Verdadeiro	Verdadeiro	Ruim	Verdadeiro
5	Jovem	Falso	Falso	Ruim	Falso
6	Meia-idade	Falso	Falso	Ruim	Falso
7	Meia-idade	Falso	Falso	Bom	Falso
8	Meia-idade	Verdadeiro	Verdadeiro	Bom	Verdadeiro
9	Meia-idade	Falso	Verdadeiro	Excelente	Verdadeiro
10	Meia-idade	Falso	Verdadeiro	Excelente	Verdadeiro
11	Velho	Falso	Verdadeiro	Excelente	Verdadeiro
12	Velho	Falso	Verdadeiro	Bom	Verdadeiro
13	Velho	Verdadeiro	Falso	Bom	Verdadeiro
14	Velho	Verdadeiro	Falso	Excelente	Verdadeiro
15	Velho	Falso	Falso	Ruim	Falso

Consideramos a base de dados *D* representada pela Tabela 2.1. Podemos calcular sua entropia da seguinte forma:

$$entropia(D) = -\left(\frac{6}{15}\right)\log\left(\frac{6}{15}\right) - \left(\frac{9}{15}\right)\log\left(\frac{9}{15}\right) = 0{,}971$$

Munidos da entropia da base *D* podemos encontrar a entropia esperada de algum dos atributos pertencentes à base de dados *D*. Por exemplo, a entropia esperada do atributo casa própria, calculado a seguir:

$$entropia(Casa\_Própria) = -\left(\frac{6}{15}\right) \times entropia(D_1) - \left(\frac{9}{15}\right) \times entropia(D_2) = 0{,}551$$

Onde D_1 e D_2 (entropia esperada) são observações feitas a partir do atributo *Casa_Própria*. Podemos calcular D_1 observando o que acontece quando um exemplo é positivo para o atributo *Casa_Própria*, ou seja, como ele é classificado (pagador ou não pagador) a partir do atributo *Casa_Própria* com valor verdadeiro. O mesmo ocorre para D_2 com valor falso. Finalmente podemos calcular o Ganho de Informação para o atributo *Casa_Própria* a partir da equação 2.4.

$$Ganho(D, Casa\_Própria) = 0{,}971 - 0{,}551 = 0{,}420$$

2.5.2 O Algoritmo C4.5

O C4.5 gera seus modelos a partir de um conjunto de dados, previamente observado, o qual traz informações empíricas sobre o conceito que o modelo deve aprender. Utilizando uma estratégia de dividir para conquistar, o C4.5 particiona recursivamente os dados em um subconjunto progressivamente mais organizados. (MENEZES, 2011).

O aprendizado inicia com a pergunta: “Qual atributo deverá ser o nó raiz?”. Responde-se a essa pergunta através da avaliação de cada um dos atributos por uma heurística, então o melhor atributo é selecionado para ser a raiz da árvore. Um descendente é criado para cada possível valor deste atributo e os exemplos são, então, repassados aos nós filhos de acordo com o valor que possuem em tal atributo. Esse processo é repetido em cada descendente até que algum critério de parada seja satisfeito. O Ganho de Informação é usado como heurística para selecionar os nós da árvore em construção. Assim, o Ganho de Informação nos mostra qual atributo devemos colocar

no nó raiz e nos demais nós.

Exemplo 2.3: Usando as informações da Tabela 2.1, representaremos uma possível escolha para o nó raiz da árvore a partir do ganho de informação discutido anteriormente.

Calculando o ganho dos quatro atributos presentes na Tabela 2.1, temos os seguintes resultados, *Casa_Própria*(0,420), *Crédito*(0,363), *Empregado*(0,324) e *Idade*(0,083). Podemos perceber que o atributo que tem o maior Ganho de Informação é o atributo *Casa_Própria*. Assim ele seria escolhido para ser o nó raiz da árvore de decisão.

Apresentado por Ross Quinlan (QUINLAN, 1993), o C4.5 visa a geração de árvores de decisão com tratamento de atributos contínuos e discretos, construindo uma árvore com um número de partições variável e com as folhas sendo indicadas pelos valores do atributo categórico.

Para evitar a geração de todas as árvores possíveis, o algoritmo C4.5 se baseia no atributo mais informativo, escolhido entre todos os atributos ainda não considerados no caminho desde a raiz. O algoritmo determina o atributo mais informativo como sendo aquele que possuir o maior Ganho de Informação, resultante da diferença do valor da informação do atributo categórico e do valor da informação do atributo em questão.

Para cada atributo é calculado o seu Ganho de Informação. O atributo que tiver o maior Ganho de Informação será considerado pelo algoritmo como o próximo nó da árvore. Assim, a partição começa pelo nó raiz e continua pelos nós filhos da mesma maneira, até que todos os exemplos dessa partição possuam a mesma classe, rotulando-se este nó como folha e recebendo sua respectiva classe. A Figura 2.2 representa o núcleo do C4.5. O pseudo-algoritmo a seguir propõe mostrar de forma simples o funcionamento de tal técnica.

Algoritmo C4.5

Entrada:

(*Exs*): conjunto de exemplos de treinamento.

(*AtribObjetivo*): é o atributo cujo valor a árvore deverá predizer.

(*Atribs*): é a lista dos atributos, que podem ser nós internos da árvore.

Saída:

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

Começo-C4.5:

01 – Crie Raiz, que será o nó raiz da árvore:

02 – Se todos os exemplos de *Exs* forem positivos, então

03 – Faça *Raiz.classe* = "+"

04 - Retorne Raiz, que é o nó raiz da árvore de decisão.

05 - Se todos os exemplos de *Exs* forem negativos, então

06 - Faça *Raiz.classe* = "-"

07 - Retorne Raiz.

08 - Se *Atribs* for uma lista vazia, então

09 - Faça *Raiz.classe*= valor de *AtribObjeto* mais frequente entre os exemplos de *Exs*.

10 - Retorne Raiz.

11 - Senão

12 - Faça A ser o atributo de *Atribs* que melhor classifica os exemplos de *Exs*.

13 - Faça *Raiz.atributo* = *A*.

14 - Para cada possível valor $v_i$do Atributo *A*

15 - Adicione um novo ramo ao nó *Raiz*, correspondente ao teste $A = v_i$.

16 - Faça Exs_{vi} ser o subconjunto de *Exs* formado pelos exemplos que têm o atributo A com valor v_i.

17 - Se o conjunto $Exs_{vi}$for vazio, então

18 - Crie uma nova folha F.

19 - Faça *F.classe*= valor de *AtribObjetivo* mais frequente entre os exemplos de *Exs*.

20 - Adicione *F* ao ramo.

21 - Senão

22 - Crie uma subárvore*SubArv*.

23 - Faça *SubArv* = C4.5(Exs_{vi}, *AtribObjetivo, Atribs –{A})*.

24 – Retorne Raiz.
Fim-Algoritmo-Cobertura.

Figura 2.2: Algoritmo C4.5 (MENEZES, 2011)

3 PROGRAMAÇÃO EM LÓGICA INDUTIVA

3.1 Introdução

A Programação em Lógica Indutiva é uma subárea do Aprendizado de Máquina que por sua vez é parte da Inteligência artificial cujo objetivo é o desenvolvimento de técnicas computacionais sobre o aprendizado bem como a construção de sistemas capazes de adquirir conhecimento de forma automática. Um sistema de aprendizado é um programa de computador que toma decisões baseado em experiências acumuladas através da solução bem sucedida de problemas anteriores. A ILP investiga a construção indutiva de teorias de cláusulas de Horn, de primeira-ordem, a partir de exemplos e de um conhecimento preliminar (MUGGLETON, 2006). Do aprendizado indutivo, herda seu objetivo: construção de ferramentas e técnicas para induzir hipóteses a partir de observações (exemplos) e sintetizar novo conhecimento a partir da experiência. Já da programação em lógica (LLOYD, 1987), a ILP herda o formalismo representacional, como a representação de teorias, hipóteses, exemplos e conhecimento preliminar.

Basicamente, um sistema ILP tem o objetivo principal de induzir teorias a partir de exemplos e de um conhecimento preliminar expresso por cláusulas de Horn. Ela pode ser aplicada a vários problemas importantes como: dimensionamento de malhas em métodos de elementos finitos, previsão de estrutura de proteínas, detecção de problemas de tráfego, processamento de linguagem natural, entre outros.

A importância dessa área pode ser mensurada, até certo ponto, pelo grande número de sistemas desenvolvidos para resolver esse problema, dentre os quais se destacam Progol (MUGGLETON, 1995) (MUGGLETON, 1996), FOIL (QUINLAN, 1990), TILDE (BLOCKEEL, DE RAEDT, 1998), Aleph (SRINIVASAN, 2003), e o EDA-ILP (PITANGUI, 2013).

Segundo Mitchell (MITCHELL,1997), o aprendizado pode ser visto como um problema de busca no espaço de todas as possíveis hipóteses. Dessa forma, pode-se dizer que o objetivo de um sistema ILP é encontrar uma teoria, expressa por cláusulas de Horn, que implica logicamente todos os exemplos positivos e não implique logicamente nenhum exemplo negativo, utilizando, para isso, um conhecimento preliminar dado a priori e aceito como correto. A ILP combina o Aprendizado Indutivo

de Teorias (AIT) e Programação em Lógica (PL), por isso faz-se necessário definir esses conceitos antes de definir formalmente PLI.

A definição de Aprendizado Indutivo de Conceitos pode ser formulada como o problema de se encontrar uma hipótese que cubra, prove, todos os exemplos positivos e não cubra nenhum dos exemplos negativos (MITCHELL, 1997). De forma geral, derivar conclusões gerais a partir de observações específicas é chamado de indução. Dessa forma, no AIC, os conceitos são de fato induzidos, uma vez que as hipóteses, conclusões, são induzidas a partir dos exemplos, observações específicas. Esse processo de indução pode ser visto como um processo de busca em que se almeja encontrar a melhor hipótese, de acordo com certa função de avaliação, no espaço de todas as possíveis hipóteses expressas na linguagem de representação (MITCHELL, 1997). A conjectura fundamental do aprendizado indutivo é que qualquer hipótese descoberta que aproxima bem um determinado conceito (função alvo) usando um conjunto suficientemente grande de exemplos de treinamento, também aproximará bem a função alvo sobre os outros exemplos não observados, generalização (MITCHELL, 1997).

Uma linguagem de representação deve ser escolhida para representar os conceitos, as hipóteses, os exemplos, e o conhecimento preliminar. Essa decisão de escolha da linguagem de representação é muito importante, uma vez que ela limita o tipo de conceito que pode ser aprendido (MITCHELL, 1997). Com uma linguagem de representação detentora de baixa expressividade, pode-se não ser possível representar algum problema, por ele ser muito complexo para a linguagem adotada. Em contrapartida, uma linguagem com grande expressividade pode ser capaz de representar um conjunto maior de domínios de problemas. Entretanto, esta solução pode fornecer grande liberdade no sentido de ser possível construir hipóteses de muitas formas diferentes, o que poderia levar a uma impossibilidade de se encontrar o conceito correto.

3.2 Linguagens de Representação

3.2.1 Lógica Proposicional

Quando se deseja usar um computador para resolver um problema, o primeiro passo a realizar é traduzi-lo para termos que o computador entenda. Dessa forma, deve-se escolher uma linguagem de representação apropriada para o problema.

As linguagens de representação variam de fragmentos da Lógica Proposicional à lógica de segunda ordem, sendo a primeira detentora de baixo poder de expressividade e essa última muito complexa e, por isso raramente utilizada. Deve-se saber que existem várias formas de representação de hipóteses, entre elas a representação proposicional e a representação em Lógica de Primeira-Ordem (PITANGUI, 2013).

A Lógica Proposicional é usada quando um problema pode ser representado por um número fixo de atributos, sendo que cada um representa uma característica específica do problema. Para exemplificar esse aprendizado de hipóteses cita-se a busca por uma hipótese que defina quando jogar tênis (MITCHELL, 1997) baseando-se nas condições climáticas. A Tabela 3.1 representa essa hipótese.

Tabela 3.1: Situações Favoráveis para se Jogar Tênis (MITCHELL, 1997)

	Atributos			**Classe**	
Dia	Tempo	Temperatura	Umidade	Vento	Joga Tênis?
1	Sol	Quente	Alta	Fraco	Não
2	Sol	Quente	Alta	Forte	Não
3	Nublado	Quente	Alta	Fraco	Sim
4	Chuva	Moderado	Alta	Fraco	Sim

De acordo com a Tabela 3.1, é possível representar facilmente as situações em que poderá haver jogos. No canto esquerdo estão representados os dias e na linha superior estão representados os atributos ou características. Assim, uma possível hipótese aprendida (também expressa de forma proposicional) poderia ser: Se (tempo = nublado ou chuvoso) então jogue tênis.

3.2.2 Lógica de Primeira-Ordem

A LPO pode ser vista como uma extensão da Lógica Proposicional sendo capaz de representar relações entre objetos, concluir particularizações de uma propriedade geral dos indivíduos de um universo de discurso, assim como derivar generalizações a partir de fatos que valem para um indivíduo arbitrário do universo de discurso (PITANGUI, 2013).

Dessa forma, a Lógica Proposicional é simples para representar ambientes complexos de forma concisa, entretanto, não tem a capacidade de expressão para descrever um ambiente com muitos objetos.

A principal diferença entre a Lógica de Primeira-Ordem e a Lógica Proposicional é o compromisso ontológico, caracterizado como sendo o que cada linguagem pressupõe sobre a natureza da realidade. Desse modo a Lógica Proposicional pressupõe que no mundo existem fatos que podem ser válidos ou não, enquanto a Lógica de Primeira-Ordem pressupõe que o mundo consiste em objetos com certas relações entre eles que podem ser válidas ou não (RUSSEL e NOVING, 2003).

Como foi dito, a Lógica de Primeira-Ordem tem uma representatividade maior que a Lógica Proposicional, assim, pode-se representar relações a partir dela usando operadores lógicos, e através de um programa lógico descobrir mais sobre essas relações. Por possuir maior expressividade quando comparada a Lógica Proposicional, a Lógica de Primeira-Ordem usa símbolos mais complexos quando comparados aos símbolos utilizados na LP.

Nesse sentido, a motivação do uso da Lógica de Primeira-Ordem vem do fato de que para alguns problemas, a Lógica Proposicional não pode representar adequadamente as estruturas de dados neles presentes (PITANGUI, 2013).

A linguagem da Lógica Proposicional não é adequada para representar relações entre objetos. Por exemplo, se fosse utilizada a linguagem proposicional para representar "João é pai de Maria e José é pai de João" seriam usadas duas letras sentenciais diferentes para expressar relações semelhantes (por exemplo, P para simbolizar "João é pai de Maria" e Q para simbolizar "José é pai de João") entretanto a representação dessas duas frases não estariam de fato captando a mesma relação de parentesco entre João e Maria e entre José e João.

A seguir será descrito os elementos que compõe o alfabeto de LPO e as suas principais definições.

1. **Símbolos Lógicos:**
    - **a.** Pontuação: consiste de parêntese aberto, parêntese fechado e vírgula.
    - **b.** Conectivos: $\neg$ (negação), $\wedge$ (conjunção), $\vee$ (disjunção), $\leftarrow$ (implicação), e $\leftrightarrow$ (bi-condicional).
    - **c.** Quantificadores: $\forall$ (universal), e $\exists$ (existencial).
    - **d.** Variáveis: se iniciam com letras maiúsculas, tais como *A*, *X*, *Var*.
    - **e.** Símbolo de igualdade (opcional): =
2. **Símbolos não-lógicos:**
    - **a.** Funções: se iniciam por letras minúsculas.
    - **b.** Constantes: são símbolos funcionais de aridade (número de argumentos) zero.
    - **c.** Predicados: expressam relações entre seus argumentos.

Definição 3.1 (Conjunto de termos de primeira-ordem): O conjunto dos termos de primeira-ordem é definido recursivamente pelas seguintes regras:

1. Qualquer constante é um termo.

2. Qualquer variável é um termo.

3. Toda expressão $f\ (t_1, \ldots, t_n)$ de $n \geq 1$ argumentos (onde cada argumento t_i é um termo e *f* é um símbolo de função de aridade n) é um termo.

4. Nada mais é um termo.

Exemplo 3.1: $f(t_1,\ldots, t_n)$ é um termo em que $t_1,\ldots, t_n$ são termos.

Definição 3.2 (Fórmula Atômica ou Átomo): Se $t_1, \ldots, t_n$ são termos e *P* é um símbolo predicativo *n*-ário (com *n* argumentos), então $P(t_1,\ldots, t_n)$ é chamado de fórmula atômica ou átomo.

Exemplo 3.2: *pai(josé, joão)* é um átomo, em que o predicado pai possui aridade dois e expressa a relação de paternidade entre *josé* e *joão*(duas constantes).

Um átomo é um literal positivo, e a negação de um átomo é um literal negativo. Se A e B são fórmulas, então ($\neg$A), (A $\wedge$ B), (A $\vee$ B), (A $\leftarrow$ B), e (A $\leftrightarrow$ B) são

fórmulas. Se B é uma fórmula e X é uma variável, então $\forall X$ (B) e $\exists X$ (B) também são fórmulas.

Definição 3.3 (Cláusula): Uma cláusula é uma disjunção de literais precedida por um prefixo de quantificadores universais, um para cada variável que aparece na disjunção, na seguinte forma:

$$\forall X_1 \forall X_2 ... \ \forall X_s \ (L_1 \vee L_2 ... \vee L_m)$$

onde cada L_i é um literal, e $X_1, X_2 ... \ X_s$ são todas as variáveis ocorrendo em $(L_1 \vee L_2 ... \vee L_m)$. O conjunto $\{A_1, A_2, ..., A_h, \neg B_1, \neg B_2, ..., \neg B_b\}$ onde A_i e B_i são átomos, indica a cláusula:

$$(A_1 \vee ... \vee A_h \vee \neg B_1 \vee ... \vee \neg B_b)$$

que é equivalentemente representada por: $A_1 ... \ A_h \leftarrow B_1, ..., B_b$ onde $A_1 ... \ A_h$ é a cabeça e $B_1, ..., B_b$ é o corpo da cláusula. Normalmente, em ILP é usado ':-' em vez de '$\leftarrow$'.

Definição 3.4 (Teoria): Uma teoria é um conjunto de cláusulas, representando a conjunção destas cláusulas.

Definição 3.5 (Cláusula de Horn): Uma cláusula de Horn é uma cláusula que possui no máximo um literal positivo, ou seja, possui, no máximo, um literal na cabeça.

Exemplo 3.3: A cláusula $A \leftarrow L_1, L_2, L_3$ é uma cláusula de Horn.

Definição 3.6 (Cláusula Definida): Uma cláusula Definida é uma cláusula de Horn com exatamente um literal na cabeça.

Exemplo 3.4: A cláusula $A \leftarrow L_1, L_2, L_3$ é uma cláusula Definida.

Definição 3.7 (Cláusula Objetivo): Uma cláusula Objetivo é uma cláusula de Horn sem o literal na cabeça.

Exemplo 3.5: A cláusula $\leftarrow L_1, L_2, L_3$ é uma cláusula Objetivo.

Definição 3.8 (Fato): Um Fato é uma cláusula Definida que possui corpo vazio.

Exemplo 3.6: A cláusula A ← é um Fato. Fatos são normalmente representados sem o símbolo de implicação.

Definição 3.9 (Programa Lógico Definido): Programa Lógico Definido é um conjunto de cláusulas definidas.

Exemplo 3.7: A figura 3.1 ilustra um Programa Lógico Definido. As cláusulas de 1 até 5 são fatos. As cláusulas de 6 a 10 ilustram algumas relações familiares que podem (ou não) ocorrer entre os fatos expressos no programa lógico. Assim, por exemplo, a cláusula de número 8 informa que um indivíduo *X* qualquer será irmão de um indivíduo *Y* qualquer, caso *X* seja irmão de *Y* (relação definida pela cláusula de número 10), e *X* seja do gênero *masculino*. As outras relações podem ser interpretadas de forma equivalente.

```
01 - pais(jorge, julio).

02 - pais(jorge, lucia).

03 - genero(lucia, feminino).

04 - genero(jorge, masculino).

05 - genero(julio, masculino).

06 - pai(X,Y) :- pais(X,Y), genero(X, masculino).

07 - mãe(X,Y) :- pais(X,Y), genero(X, feminino).

08 - irmão(X,Y) :- irmãos(X,Y), genero(X, masculino).
```

Figura 3.1: Exemplo de Programa Lógico (PITANGUI, 2013)

Definição 3.10 (Substituição θ): Uma substituição $\theta = \{X_1/t_1,..., X_n/t_n\}$ é um mapeamento finito de variáveis em termos que atribui a cada variável X_i um termo t_i, para $1 \leq i \leq n$.

Exemplo 3.8: θ = {*X/a, Y/b*} é um exemplo de substituição θ em que as ocorrências da variável *X* serão simultaneamente substituídas pela constante *a*, e as ocorrências da variável *Y* serão simultaneamente substituídas pela constante *b*.

Definição 3.11 (Termo, átomo ou cláusula básica(o)): Um termo, átomo ou cláusula é dito básica(o) caso não exista nenhuma variável ocorrendo nele(a).

Exemplo 3.9: O literal *pai (josé, joão)* é dito básico, uma vez que nele não ocorre nenhuma variável.

Definição 3.12 (Unificador):(Unificador): Assuma que L_1 e L_2 são literais. Pode-se dizer que a substituição θ é um unificador para L_1 e L_2, se e somente se, $L_1\theta = L_2\theta$. Pode-se dizer também que L_1 e L_2 são unificáveis via θ.

De forma geral, podem existir vários unificadores, e, dentre eles, o unificador mais geral pode ser encontrado.

Definição 3.13 (Generalidade de Substituição θ): Considere duas substituições θ_1 e θ_2. Diz-se que θ_1 é mais geral que θ_2, $\theta_1 \preceq \theta_2$, se existir uma substituição θ_3 tal que $\theta_1\theta_3 = \theta_2$.

Definição 3.14 (Unificador Mais Geral): Pode-se dizer que θ é o Unificador Mais Geral (UMG) para L_1 e L_2, se θ é mais geral que todos os outros unificadores de L_1 e L_2.

Exemplo 3.10: Assuma que os literais L_1 e L_2 são, respectivamente, *p(X,b)* e *p(a,b)*. Dessa forma, θ = {*X/a*} é um unificador para L_1 e L_2 e, além disso, θ é o UMG para L_1 e L_2.

Definição 3.15 (θ-subsume ($\preceq$)): Uma cláusula *C* θ-subsume a cláusula *D*, ou seja, $C \preceq D$, se existe uma substituição θ tal que $C\theta \subseteq D$, isto é, todos os literais de *C* com a substituição θ formam um subconjunto dos literais de *D*.

Exemplo 3.11: A clausula *C*: *f(A,B)* ← *p(B,G), q(G,A)* θ-subsume a cláusula *D*: *f(a,b)*← *p(b,g), q(g,a), t(a,d),* através da substituição θ = {*A/a, B/b, G/g*}.

Apresentados alguns dos principais conceitos que regem a LPO, pode-se definir o paradigma de Programação em Lógica da seguinte maneira:

Definição 3.16 (Programação em Lógica): Programação em Lógica é o uso de uma notação lógica formal para comunicar processos computacionais a um computador (SEBESTA, 1996).

A LPO é a notação formal utilizada nas atuais linguagens de programação em lógica, como Prolog, por exemplo. As linguagens de programação em lógica necessitam de um processo de inferência para computar os resultados desejados (SEBESTA, 1996). O método de prova chamado de resolução (ROBINSON, 1965) é utilizado no processo de inferência. A definição seguinte deve ser interpretada dentro do escopo de programação em lógica.

Definição 3.17 (Consulta a um Programa Lógico): Uma Consulta a um Programa Lógico é uma cláusula na forma $\leftarrow L_1, L_2, ..., L_n$, em que L_i, para $1 \leq i \leq n$, são literais.

Quando se consulta um programa lógico, um procedimento de resolução chamado de SLD (*Selection rule driven Linear resolution for Definite clauses*) é aplicado com objetivo de verificar se a consulta é satisfeita pelo programa lógico. Ela aplica uma série de passos de derivações para verificar se a consulta é satisfeita. Um passo de derivação consiste das seguintes operações:

1. Selecione um literal L_i $1 \leq i \leq n$, da consulta.
2. Selecione uma cláusula C no programa lógica tal que sua cabeça possa ser unificada com L_i.
3. Selecione o UMG para a consulta e a cabeça da cláusula C.
4. Substitua L_i na consulta pelo corpo da cláusula C, e aplique o UMG à consulta resultante.

Nos passos 1 e 2, a ordem em que os literais da consulta devem ser resolvidos (regra de seleção) e a ordem em que as cláusulas do programa lógico são usadas na derivação precisam ser especificados para fazer com que a resolução seja determinística. Caso a derivação acabe com a cláusula vazia (cláusula sem nenhum literal), denotada por □, então a derivação foi um sucesso e a resposta à consulta é um "sim". Caso a derivação não encontre a cláusula vazia, a derivação não obteve sucesso e retorna um "não" a consulta.

De forma geral, se uma consulta *C* pode ser derivada de um programa lógico *PL* em zero ou mais passos de derivação, denota-se, este fato por *PL* |- *C*. Uma importante propriedade da resolução é que apenas consequências lógicas podem ser derivadas. Esta propriedade é chamada corretude *(soundness)* da resolução. De forma geral, uma fórmula *F* implica logicamente uma fórmula *G* (*F* |= *G*) se e somente se todo modelo de *F* for também modelo de *G*. Outra importante propriedade da resolução é chamada de completude (*completeness*), que afirma que, para um dado programa lógico PL e *A* um fato básico, então, *PL* |= *A* se e somente se (*PL* ∪ {←*A*}) |- □. Pode-se provar que a resolução SLD possui as propriedades de corretude e de completude quando as cláusulas trabalhadas se resumem às cláusulas de Horn.

Uma vez que se tenha definido a regra de seleção, a totalidade das derivações SLD para uma dada consulta e um dado programa lógico pode ser representada em uma árvore SLD. Cada ramo da árvore é uma derivação SLD que ocorre via regra de seleção. Os nós da árvore são consultas com um literal selecionado, sendo que cada nó da árvore tem exatamente um filho para cada cláusula que unifica com o literal selecionado da consulta contida no nó. Em Prolog, a árvore SLD é explorada pela busca em profundidade (RUSSEL, NORVIG, 2003), e dessa forma, são construídas todas as possíveis derivações para a consulta até se encontrar uma cláusula vazia, ou até todas as possíveis derivações terem sido tentadas. No último caso, como já afirmado, a consulta falha. A título de ilustração do processo descrito, atente para o exemplo 3.12 a seguir.

Exemplo 3.12: considere uma simplificação do programa lógico da figura 3.1 que é apresentado na figura 3.2

```
01 - pais(jorge, julio).

02 - pais(jorge, lucia).

03 - genero(lucia,feminino).

04 - genero(jorge, masculino).

05 - genero(julio, masculino).

06 - pai(X,Y) :- pais(X,Y), genero(X, masculino).
```

Figura 3.2: Exemplo Simples de Programa Lógico Definido (PITANGUI, 2013)

Suponha que a seguinte consulta foi realizada: ← *pai(X, lucia)*, ou seja, deseja-se saber quem é o pai de *lucia*. No primeiro passo, com θ_1 = {*X/X* e *Y/lucia*}, o único literal da consulta inicial se unifica com a cabeça da cláusula de número 6, fornecendo, após aplicação do passo 4 da derivação, a consulta ← *pais(X, lucia), genero(X, masculino)*. O primeiro literal desta nova consulta é unificado pela substituição θ_2 = *{X/jorge}* com a cabeça da cláusula de número 2, fornecendo, após aplicação do passo 4 da derivação, a consulta ← *genero(jorge, masculino)*. O único literal desta nova consulta é unificado pela substituição θ_3 = {∅} com a cabeça da cláusula de número 4 resultando na cláusula vazia □, ou seja, a derivação obteve sucesso e o sistema retorna um "sim" como resultado deixando instanciada a variável *X* com a constante *jorge*, ou seja, *X = jorge*. Este é o único caminho possível na árvore SLD devido à simplicidade do programa lógico. Contudo, se existissem, por exemplo, várias cláusulas com o literal *pai*(*X*, *Y*) na cabeça, certamente ter-se-iam vários outros ramos na árvore SLD para esta consulta específica.

3.3 Programação em Lógica Indutiva

3.3.1 Introdução

Definição 3.18 (Programação em Lógica Indutiva): A PLI pode ser vista como o estudo de métodos de aprendizado de hipóteses que são expressas por meio de predicados lógicos de primeira ordem. Segundo (PITANGUI, 2013), dados: (a) um conhecimento preliminar invariante denotado por *BK*, (b) um conjunto de exemplos positivos $E+$ e negativos $E-$, deve-se encontrar uma teoria *H* que, junto com o conhecimento preliminar *BK*, implique logicamente todos os exemplos (completude), ou seja, $BK \cup H \models E+$, e nenhum dos exemplos negativos (consistência), $\forall e^- \in E- : BK \cup H \not\models e^-$, e que esteja de acordo com o critério de qualidade estabelecido, tal como minimalidade do tamanho da teoria.

Na prática, os requisitos de completeza e consistência são relaxados, assim deve-se encontrar uma teoria que implique a maior quantidade possível de exemplos

positivos ao mesmo tempo em que implica logicamente a menor quantidade possível de exemplos negativos.

A partir disso pode-se dizer que o objetivo de ILP é o de deduzir uma teoria (conjunto de cláusulas) quando lhe são dados uma base de dados com fatos e outras relações lógicas (BK) (PITANGUI, 2013).

3.4 Sistemas em ILP

3.4.1 Conceitos Preliminares

Sistemas ILP, assim como outros sistemas de aprendizado indutivo, podem ser divididos ao longo de várias classes, como segue:

Sistemas *Top-Down*: começam com hipótese H tal que o conjunto $H \cup B$ é muito genérico e, então, o especialista. Um sistema *Top-Down* pode adaptar-se localmente aos exemplos através de um passo de generalização. Tal generalização pode ser necessária para corrigir um passo anterior de maior especialização, o que poderia tornar a hipótese final bastante fraca. Após a correção, o sistema continua sua busca *Top-Down.*

Sistemas *Bottom-Up*: começam com uma hipótese H, tal que $H \cup B$ é muito específico e, então, o generaliza. Analogamente ao sistema *Top-Down,* o sistema *Bottom-Up* pode algumas vezes realizar um passo de especialização. Entretanto, um sistema pode geralmente ser classificado de maneira natural como *Top-Down* ou *Bottom-Up,* dependendo da direção geral de sua busca.

3.4.1.1 Algoritmo de Cobertura

Grande parte dos sistemas ILP utiliza um algoritmo de cobertura em seu ciclo de execução (MITCHELL, 1997), como por exemplo, FOIL (QUINLAN, 1990) o Progol (MUGGLELTON, 1995) (MUGGLETON, 1996), e o Aleph (SRINIVASAN, 2003).

Um algoritmo de cobertura constrói uma nova regra até que todos os exemplos positivos da classe sejam cobertos pelo conjunto de regras, ou até que algum outro

critério de qualidade pré-definido seja satisfeito. Uma vez que uma regra é adicionada ao modelo, todos os exemplos positivos cobertos por aquela regra são removidos do conjunto de exemplos positivos.

O papel de um exemplo positivo (para uma dada classe) é forçar a atenção do algoritmo de aprendizado para uma área particular do espaço de exemplos. Já o papel dos exemplos negativos é prevenir uma super-generalização e assegurar que outras áreas do espaço de exemplos não sejam cobertas pela regra. Exemplos positivos devem ser cobertos por pelo menos uma regra, enquanto que exemplos negativos não deveriam ser cobertos por nenhuma regra.

A ideia geral de todos os algoritmos de cobertura pode ser resumida nos seguintes passos:

1. Aprenda uma cláusula que prove certa quantidade de exemplos positivos.
2. Remova todos os exemplos positivos provados pela cláusula.
3. Repita passos 1 e 2 até que todos os exemplos positivos estejam provados ou até que outro critério de parada seja satisfeito.

3.4.1.2 *Modos, Determinations* e a *Bottom Clause*

Enquanto o algoritmo de cobertura dita, de certa forma, como a teoria será construída, os *modos*, os *determinations* e a *bottom clause* (BC), ditam de forma geral, quais são as possíveis cláusulas a serem consideradas durante a busca. Pode-se dizer que estes restringem o espaço de busca a certas cláusulas que satisfaçam certos tipos de restrições.

1. Modos*:*

As declarações de *modos* descrevem relações (predicados) entre objetos de dados e tipos desses dados. Adicionalmente, essas declarações informam se a relação estipulada pode ser utilizada na cabeça (declaração do tipo *modeh*) ou no corpo (declarações do tipo *modeb*) das cláusulas a serem geradas. Na declaração de *modos*, também são descritos o tipo dos argumentos e o número máximo de instanciações de cada predicado. Essas declarações têm o formato *mode (Número_Chamadas, Modo)*(PITANGUI, 2013).

O parâmetro *Número_Chamadas*, ou, *recall_number*, determina o limite máximo do número de instanciações alternativas de um predicado, sendo que uma instanciação é a substituição dos argumentos do predicado por variáveis ou constantes. Segundo (PITANGUI, 2013), o *recall_number* pode ser qualquer número positivo $n \geq 1$, ou um *, indicando que não existe limite de instanciações. Para se atribuir um número positivo específico para o *recall_number*, existe a necessidade de se conhecer o número de instanciações possíveis para um determinado predicado. Por exemplo, para uma declaração do predicado ancestral (Ancestral, Pessoa), poderia ser fornecido um *recall_number* igual a *, pois, a princípio, não se sabe quantos ancestrais uma pessoa pode ter. O *Modo* indica o formato do predicado que será utilizado. Esta declaração é realizada da seguinte forma: *predicado(Tipo_do_Argumento_1, ... , Tipo_do_Argumento_n).* Considere o exemplo a seguir (PITANGUI, 2013).

Exemplo 3.13: Considere o aprendizado da relação *sogra(Sogra, Genro)*, com BK descrito pelas relações *progenitor(Mãe, Filha)* e *esposa(Esposa, Marido)*. As declarações de modo poderiam ser (já na sintaxe correta de como devem ser realizadas):

:- modeh(1, sogra(+mulher, +homem)).
:- modeb(, progenitor(+mulher, -mulher)).*
:- modeb(1, esposa(+mulher, +homem)).

onde os símbolos + e - serão explicados a seguir. A primeira declaração, *modeh(1, sogra(+mulher, +homem))*, indica o predicado que irá compor a cabeça das cláusulas, neste caso o predicado *sogra/2* (/2 significa que o predicado possui aridade 2, ou seja, dois argumentos). O valor de *recall_number* é igual a 1 para este predicado, pois para uma dado *homem* tem-se uma única *esposa* e, portanto, uma única *sogra*. Continuando com a interpretação desta declaração, tem-se que o primeiro argumento do predicado *sogra(Sogra, Genro)*, i.e., *Sogra*, deve ser do tipo *mulher*, e *Genro* deve ser do tipo *homem*. A segunda e a terceira declaração indicam, respectivamente, que as cláusulas geradas devem ter no corpo o predicado *progenitor(Mae, Filha)* e *esposa(Esposa, Marido)* nos quais, *Mãe*, *Filha* e *Esposa* são do tipo *mulher* e *Marido* é do tipo *homem*. O valor do *recall_number* para o predicado *progenitor(Mae, Filha)* é igual a *, uma vez que não se sabe quantas filhas uma mãe pode ter. Para o predicado *esposa(Esposa,*

Marido), este parâmetro tem o valor 1, uma vez que para uma dada *esposa* tem-se um único *marido* e vice-versa.

Os tipos dos argumentos podem ser +, - ou #. O símbolo '+' indica que o argumento do predicado é uma variável de entrada. O símbolo '-' indica uma variável de saída enquanto '#' indica que este argumento é uma constante. A utilização destes tipos de variáveis deve seguir as seguintes regras:

- Variáveis de Entrada (+): qualquer variável de entrada do tipo T em um literal do corpo B_i deve aparecer como uma variável de saída do tipo T em um literal do corpo antes de B_i, ou como uma variável de entrada do tipo T na cabeça.

- Variáveis de Saída (-): qualquer variável de saída do tipo T em um literal na cabeça deve aparecer como uma variável de saída do tipo T em um literal do corpo B_i. Qualquer literal que contenha uma variável de saída deve ter, no mínimo, uma variável de entrada. Isto pode ser verificado na declaração de modos especificados acima.

- Constantes (#):qualquer argumento denotado por # só pode ser substituídos por constantes.

Os tipos devem ser especificados para cada argumento dos predicados utilizados na construção de uma cláusula. Para o Progol e o Aleph, por exemplo, os tipos são apenas nomes e a declaração de tipos nada mais é que um conjunto de fatos. Por exemplo, a descrição dos objetos dos tipos homem e mulher poderiam ser:

mulher(maria),mulher(ana), homem(paulo), homem(pedro).

2. *Determinations:*

Segundo Pitangui (PITANGUI, 2013), grande parte dos sistemas ILP, utilizam a relação *determination/2* para declarar os predicados que podem ser usados no corpo de uma cláusula. Esta declaração possui o seguinte formato:

determination(Predicado_Alvo/Aridade, Predicado_Corpo/Aridade)

em que *Predicado_Alvo* é o predicado que irá aparecer na cabeça da cláusula induzida, e *Predicado_Corpo* é o predicado que irá aparecer no corpo da cláusula induzida. Note

que ambos os predicados devem ser sucedidos por sua aridade. Por exemplo, para o aprendizado do predicado alvo *sogra(Sogra, Genro)*, uma possível declaração seria:

:- determination(sogra/2, progenitor/2)

:- determination(sogra/2, esposa/2)

Estas declarações afirmam que a cláusula que possui o predicado *sogra/2* na cabeça, pode possuir em seu corpo os predicados *progenitor/2* e/ou *esposa/2*.

3. *A Bottom Clause:*

Alguns sistemas ILP utilizam a ordenação do espaço de hipóteses para executar a sua busca. De forma geral, alguns sistemas ILP limitam inferiormente e superiormente o espaço de busca de uma única cláusula, formando, desta forma, um látice. Este látice é geralmente limitado superiormente pelo elemento mais geral, ou seja, a cláusula vazia, enquanto que o limite inferior é dado pela cláusula mais específica (*bottom clause*), isto é, o elemento menos geral do látice. Assumindo que $L_i(M)$ denota a linguagem dos modos, e $|-_h$ □ denota a derivação da cláusula vazia em no máximo h resoluções, a *bottom clause* pode ser definida como segue (MUGGLETON, 1995).

Pragmaticamente, a *bottom clause* é construída utilizando um exemplo positivo, o BK, a declaração de modos, e as *determinations*. Esta cláusula tem a propriedade de provar, em alguns passos de resolução, o exemplo positivo utilizado em sua construção. De forma geral, pode-se dizer que sistemas que utilizam a *bottom clause* como limite inferior do látice e a cláusula vazia como limite superior desta estrutura, limitam seu espaço de busca a cláusulas que possuem em seu corpo um subconjunto dos literais presentes no corpo da *bottom clause*.

3.4.2 Alguns Sistemas ILP

3.4.2.1 FOIL (*First Order Inductive Learner*)

FOIL é uma extensão do sistema de aprendizado de árvore de decisão ID3 (QUINLAN, 1986). Seu processo de aprendizado utiliza uma abordagem de cobertura que supõe que as cláusulas da hipótese são aprendidas uma a uma. Cada nova cláusula

C construída pelo sistema deve ser de tal forma que, junto com a hipótese atual (que corresponde a todas as cláusulas já obtidas anteriormente) e conhecimento prévio, implica alguns exemplos positivos que não são implicados sem ela. Além disso, a cláusula *C* e o conhecimento prévio não podem implicar exemplos negativos. Após a construção de uma cláusula, o sistema a adiciona à hipótese atual e remove do conjunto de exemplos àqueles positivos que foram cobertos por ela. Este processo continua até que não existam mais exemplos positivos no conjunto de exemplos (QUINLAN, 1990).

O sistema FOIL se enquadra na categoria de sistemas *Top-down*. Cada cláusula da hipótese é construída começando da cabeça, que é então especializada por um operador de refinamento que adiciona novos literais ao corpo da cláusula. A busca através de um grafo de refinamento, ou seja, a seleção de literais que serão adicionados, é guiada por uma heurística de Ganho de Informação. A figura 3.3 apresenta o funcionamento básico do sistema FOIL.

Funcionamento Geral do Sistema FOIL

```
Entrada:
(E+, E-): conjunto de exemplos de treinamento formado por
exemplos positivos E+ e por exemplos negativos E-;
     BK: Conhecimento preliminar;
Variáveis Locais:
cláusula, teoria (conjunto de cláusulas);
Saída:
Teoria modificada;
Começo-FOIL:

01- teoria := ∅;
02- cláusula := □;
03- Enquanto (cláusula provar exemplos negativos) faça:
04-    Encontre um "bom" literal para ser adicionado à cláusula;
06- fim-enquanto;
05- teoria := teoria ∪ cláusula;
06- E+ := E+ - exemplos positivos provados por cláusula;
07- Se (|E+| ≠ 0) então
08-    retorne ao passo 2;
09- fim-se;
10- retorne teoria;
Fim-FOIL.
```

Figura 3.3: Funcionamento do Algoritmo FOIL (Fonte: PITANGUI, 2013)

3.4.2.2 PROGOL

O sistema Progol (MUGGLETON, 1995), (MUGGLETON, 1996) usa o algoritmo de cobertura como forma de construir suas teorias juntamente com a construção da *bottom clause* que é usada para limitar inferiormente o látice que representa o espaço de busca de cláusulas isoladas. Dessa forma, o sistema Progol se restringe a buscar por cláusulas que são mais gerais que a *bottom clause* gerada, ou seja, este sistema busca por clausulas cujo corpo é constituído por um subconjunto dos literais presentes no corpo da *bottom clause*. A Figura 2.3 ilustra o funcionamento do algoritmo.

Funcionamento Geral do Sistema Progol

Entrada:
(E+, E-): conjunto de exemplos de treinamento formado por
exemplos positivos E+ e por exemplos negativos E-;
BK: Conhecimento preliminar;
Variáveis Locais:
cláusula, *bottom clause* (⊥), teoria (conjunto de cláusulas);
Saída:
Teoria modificada;

```
01- teoria := ∅;
02- Enquanto (|E+| ≠ 0) faça:
03-     Selecione e ∈E+;
04-     Construa ⊥ a partir de e;
05-     Encontre uma "boa" cláusula entre ⊥e usando o A*;
06-     teoria := teoria ∪ cláusula;
07-     E+ := E+ - exemplos positivos provados por cláusula;
08- fim-enquanto;
08- retorne teoria;
Fim-Progol.
```

Figura 3.4: Funcionamento do Algoritmo PROGOL (PITANGUI, 2013)

3.4.2.3 ALEPH (*A Learning Engine for Proposing Hypotheses)*

De forma geral, pode-se dizer (GOADRICH, 2009) que o Aleph (SRINIVASAN, 2003) é uma implementação do Progol totalmente construída utilizando a linguagem Prolog. Dessa forma, as declarações de modos, *determinations*, geração da *bottom clause* e até mesmo a maneira sequencial de construção de teorias, que é realizada utilizando um algoritmo de cobertura e a *bottom clause*, são os mesmos do sistema Progol. O que torna o Aleph importante e até mesmo (de certa forma) mais popular que o Progol, é a sua potencial flexibilidade, uma vez que o Aleph possui dezenas de parâmetros que se adaptam a uma grande gama de problemas (PITANGUI, 2013).

3.5 A Proposicionalização

A proposicionalização da base é um método onde uma base de dados em Lógica de Primeira-Ordem é transformada em uma base de dados em Lógica Proposicional (FRANÇA; ZAVERUCHA, 2013). Primeiramente existe uma base de dados em LPO que deve ser saturada. Esse processo satura (gera uma *bottom clause*) exemplo por exemplo fazendo com que sejam gerados os literais pertencentes ao exemplo em questão. Esses novos literais gerados serão usados mais tarde como os atributos da nova base de dados. A proposicionalização ocorre da seguinte forma: a nova base de dados em Lógica Proposicional será formada por todos os literais pertencentes aos exemplos da base em LPO, ou seja, cada atributo de cada exemplo da nova base será representado por um literal da base antiga, e o número total de atributos na base proposicionalizada será igual ao número total de literais não repetidos. Desta forma, uma instância ou exemplo será formada por um vetor X com N atributos (número total de literais não repetidos) e uma classe y. Assim, cada atributo de uma instância da base em Lógica Proposicional receberá valor verdadeiro ou falso de acordo com a existência ou não daquele literal na representação em LPO, ou seja, caso um indivíduo possua o literal Y na base de dados representada pela Lógica de Primeira-Ordem, na representação em Lógica Proposicional, a instância ou exemplo receberá o valor verdadeiro para o atributo Y' correspondente ao literal Y, caso contrário, receberá o valor falso. Dessa forma é possível ilustrar o que acontece a partir do Exemplo 3.4:

Exemplo 3.4:

Dadas as seguintes cláusulas:

Cláusulas da classe *True* (verdadeiro):

1 - *a(X, Y) :- b(X, Y), c(Z, Y)*

Cláusulas da classe *False* (falso):

2 - *b(Z, Y) :- b(X, Z), c(Z, Y)*

3 - *c(Y, X) :- a(X, Y), d(X, Z)*

4 - *d(X, Z) :- b(X, Z), c(Z, Y)*

5 - *c(Y, X) :- b(X, Z), a(X, Y)*

Aplicando a proposicionalização:

Primeiro conta-se os literais não repetidos

1 *a(X, Y)*; 2 *b(X, Y)*; 3 *c(Z, Y)*; 4 *b(Z, Y)*; 5 *b(X, Z)*; 6 *c(Y, X)*; 7 *d(X, Z)*.

A partir dos 7 (sete) literais não repetidos encontrados, montaremos a nova base:

Tabela 3.2: Cláusulas Proposicionalizadas

	a(X, Y)	***b(X, Y)***	***c(Z, Y)***	***b(Z, Y)***	***b(X, Z)***	***c(Y, X)***	***d(X, Z)***	**Classe**
1	Verdadeiro	Verdadeiro	Verdadeiro	Falso	Falso	Falso	Falso	Verdadeiro
2	Falso	Falso	Verdadeiro	Verdadeiro	Verdadeiro	Falso	Falso	Falso
3	Verdadeiro	Falso	Falso	Falso	Falso	Verdadeiro	Verdadeiro	Falso
4	Falso	Falso	Verdadeiro	Falso	Verdadeiro	Falso	Verdadeiro	Falso
5	Verdadeiro	Falso	Falso	Falso	Verdadeiro	Verdadeiro	Falso	Falso

No final obtém-se uma base de dados em LP, onde todos os literais são representados por atributos que recebem um valor verdadeiro ou falso. No problema em questão, é adicionado um atributo a mais para cada indivíduo que representa a classe à

qual o exemplo pertence. Assim, no caso a cláusula pertença à classe *false* o indivíduo também pertencerá à classe *false* (possuirá *false* como valor do atributo classe). Um problema desse método é que o número de literais não repetidos de todas as cláusulas será transformado no número de atributos para cada indivíduo da base em LP, logo o número de atributos para cada indivíduo será grande, e nem todos os atributos serão interessantes para a classificação. A partir dessa constatação, foi possível observar que seria interessante usar algum método para selecionar os melhores atributos da base antes da aplicação do método de classificação. Para solucionar esse problema foi aplicadas as a medida de Ganho de Informação.

4 O SISTEMA PROP-AG

4.1 Construção Geral do Sistema

O sistema PROP-AG, desenvolvido nesse trabalho é um algoritmo com o propósito de encontrar uma hipótese que classifique uma base de dados. Nele uma base de dados é proposicionalizada, os melhores atributos pertencentes a nova base são filtrados, a partir desses melhores atributos cria-se um otimizador (AG). Depois de evoluir a população até um ponto satisfatório, escolhe-se o melhor indivíduo (modelo de classificação), que é o resultado obtido pelo sistema. O sistema foi escrito em C++, usando também a partir de chamadas, o classificador C4.5 (J48) do Weka que é uma coleção de algoritmos para aprendizado de máquina implementados em JAVA, .

4.2 O Algoritmo Genético no sistema PROP-AG

Explicaremos rapidamente como o algoritmo genético opera no sistema.

1- Representação dos indivíduos: um indivíduo é o conjunto dos *N* melhores atributos. Esse indivíduo é representado por um vetor binário onde cada bit representa a utilização ou não do atributo para construir o modelo de classificação. Os indivíduos são iniciados aleatoriamente.

2- Operador de Recombinação: dois indivíduos são recombinados utilizando a recombinação de um ponto.

3- Operador de Mutação: inverte o gene de um indivíduo (de 0 para 1, ou de 1 para 0) com certa probabilidade.

4- Operador de Seleção: a seleção adotada é a seleção por torneio. A seleção por torneio funciona a partir dos seguintes passos (CATARINA, 2005).

i. Escolher inicialmente com a mesma probabilidade *n* indivíduos;
ii. Selecionar indivíduos com maior aptidão dentre os *n* escolhidos;
iii. Repetir os passos 1 e 2 até preencher a população desejava.

Adotamos o *n* igual a 2 em nosso sistema.

4.3 Algoritmo Geral

Com o intuito de facilitar o entendimento, o PROP-AG pode ser dividido em alguns passos lógicos mostrados no algoritmo da Figura 4.1. Os parâmetros relatados no pseudo-algoritmo são os parâmetros básicos para o funcionamento do algoritmo. O PROP-AG utiliza mais parâmetros como entrada além dos mostrados na Figura 4.1. Eles são descritos nas sessões 4.5 e 4.6.

Algoritmo PROP-AG

Entrada:

(D): base de dados em Lógica de Primeira-Ordem contendo os exemplos negativos e positivos;

(P): Porcentual de atributos que serão usados para a classificação;

(numGe): quantidade de gerações do AG;

Variáveis Locais:

(D'): base de dados Proposicionalizada;

(P'): número de atributos que serão usados para a classificação;

(pop): população do algoritmo genético;

(t): tamanho da população do AG;

Saída:

(resultadoWeka: melhor árvore de decisão C4.5 construída;

Começo-PROP-AG:

01 – Proposicionaliza a base de dados *D* e armazena *D'*;

02 – Calcula o Ganho de Informação para todos os atributos da base de dados D';

03 – Escolhe os *P'* melhores atributos;

04 – Cria a população inicial do AG (*pop*) com *t* indivíduos inicializados aleatoriamente;

05 - Calcular_aptidão_população(pop);

06 – Para(($cont_1$:= 0) até *numGe*) *faça:*

07 – Reproduzir_Indivíduos(*pop*);

08 - Mutação(pop);

08 - Calcular_aptidão_população_C4.5(pop); Nesse ponto é criado uma árvore para cada indivíduo, e a aptidão é dada pela aptidão da própria árvore.

09 - Seleção_Torneio(pop);

10 - fim-para;

11 – Seleciona melhor indivíduo da população;

13 – Constrói modelo de decisão a partir do C4.5 usado;

Fim-PROP-AG.

Figura 4.1:Pseudo Código PROP-AG

4.4 Explicação Sobre o PROP-AG

O algoritmo PROP-AG, como mostrado anteriormente funciona da seguinte forma: primeiramente, escolhemos uma base de dados para a qual será gerado o modelo de classificação. Depois da base carregada, acontece proposicionalização, no qual a base de dados passa a ser representada em Lógica Proposicional. Após a proposicionalização, calcula-se o Ganho de Informação de cada um dos atributos da base, os *N* (percentual de atributos) atributos mais valorosos serão usados na construção do modelo. A partir desse ponto o verifica-se se o número de gerações para o otimizador é maior que 0 (zero). Caso essa condição seja satisfeita o otimizador criará uma população inicial. Essa população evoluirá e ao final será escolhido o indivíduo (modelo de classificação) com a maior aptidão. Caso o número de gerações seja igual à 0 (zero) o otimizador não será ativado, sendo assim, será criado um classificador com os melhores atributos selecionados que é a resposta. A Figura 5.2 representa o fluxo do algoritmo. A partir da Figura 5.2 é possível observar que o algoritmo pode ser executado com ou sem o otimizador AG. A versão do sistema executada sem o otimizador AG é chamada de PROP. O WEKA é um produto da Universidade de Waikato (Nova Zelândia) e foi implementado pela primeira vez em sua forma moderna em 1997. Ele usa a GNU *General Public License* (GPL). O software foi escrito na linguagem JAVA e contém uma GUI para interagir com arquivos de dados e produzir resultados visuais (pense em tabelas e curvas). Ele também tem uma API geral, assim é possível incorporar o WEKA com outros sistemas.

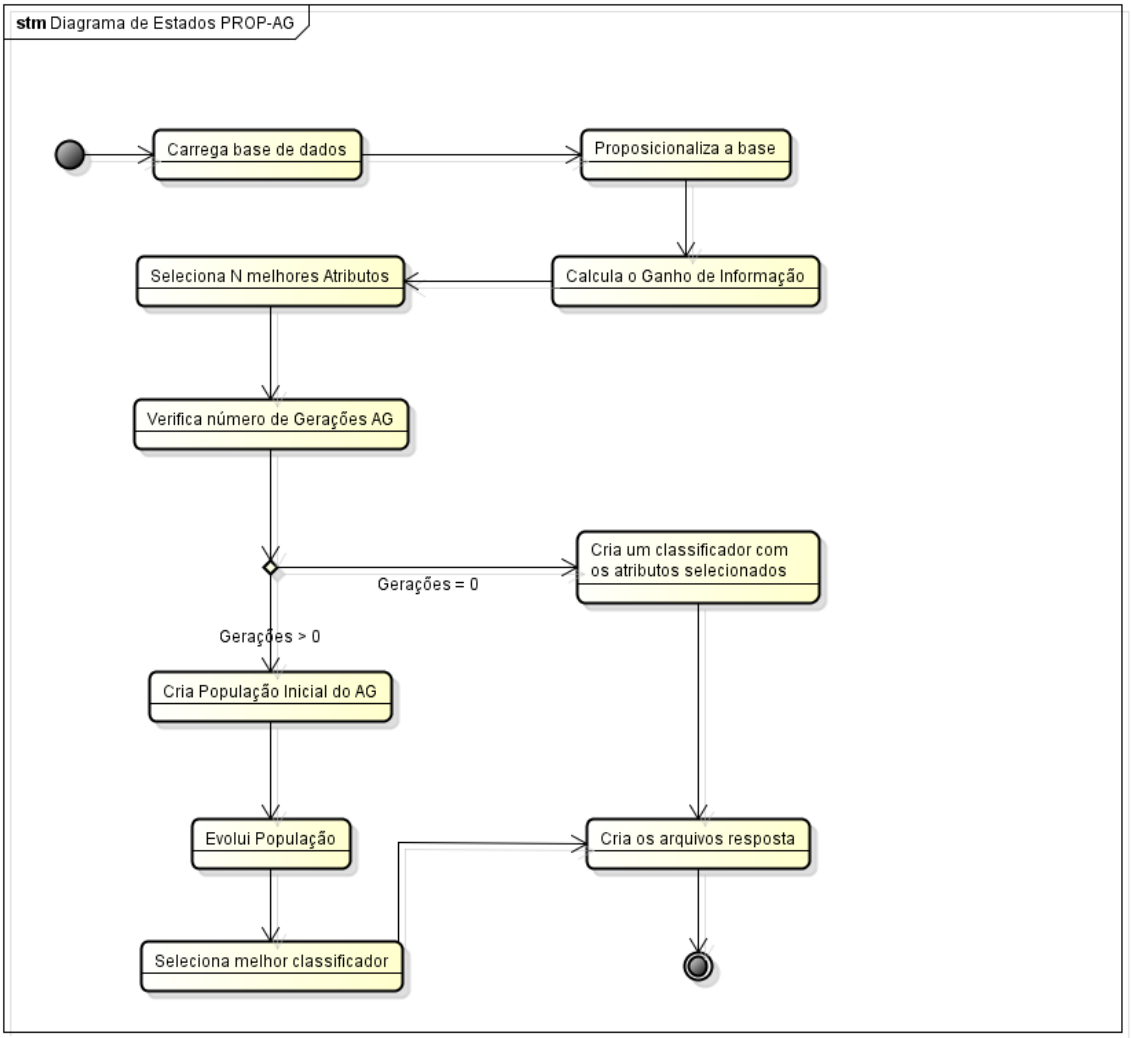

Figura 4.2: Diagrama de Estados do Algoritmo PROP-AG

4.5 Parâmetros Necessários para a Execução do PROP

O Algoritmo PROP-AG pode ser executado a partir de uma chamada ao sistema dentro da pasta onde foi compilado da seguinte forma: ./PROP-AG [lista_de_parâmetros], onde a lista de parâmetros é composta pelos parâmetros do PROP mais os parâmetros do otimizador. O chamado então fica da seguinte forma: ./PROP-AG ***nome_da_base percent_atributos posicao_dif diferenciador pasta_alg experimento gerações***

1. ***nome_da_base***: base de dados que será proposicionalizada. O algoritmo espera receber o nome da pasta onde estão os arquivos da base de dados. Ex.: Alz_amine que corresponde a base de dados Amine.

2. ***percent_atributos***: é o percentual de atributos da base que será utilizado para a criação do modelo de classificação, ou seja, os *X* melhores atributos que serão usados.
3. ***posicao_dif***: uma base de dados geralmente contém dez sub-bases, os arquivos das bases usadas são nomeados de *train.f, test.f, train.n, test.n* e *train.b*. Para identificar cada uma delas usa-se o número da base na frente do nome ou após o nome. Ex. 1train.f ou train1.f. O atributo *posicao_dif* tem a função de informar ao PROP se essa identificação vem antes ou após o nome (*train, test*).
4. ***diferenciador***: o diferenciador indica qual sub-base será usada pelo PROP.
5. ***pasta_alg*:** indica o diretório pai no qual o algoritmo se encontra para que ele possa encontrar as bases de dados e criar os arquivos de resposta.
6. ***experimento:*** nome da subpasta onde serão salvas as saídas do algoritmo, ou seja, os arquivos de resposta.
7. ***geracoes***: número de gerações do AG. Esse parâmetro é detalhada na sessão 5.6.2.

4.6 Influência dos Parâmetros

Discutem-se aqui os principais parâmetros usados nesse algoritmo e sua finalidade. Os parâmetros do algoritmo proposto foram divididos em duas partes. A primeira parte relata os parâmetros que a árvore de decisão C4.5 pode usar no Weka. A segunda parte relata os parâmetros do AG.

4.6.1 Os Parâmetros do Weka

Escolheu-se utilizar a configuração de parâmetros *default* para a execução do algoritmo C4.5. Abaixo seguem os principais parâmetros que podem ser configurados.

1. **–U:** use uma árvore não podada.
2. **–C <0,25>:** definir limite de confiança para a poda. O valor padrão é 0,25.
3. **–M <2>:** definir o número mínimo de instâncias por folha. O valor padrão é 2.
4. **–B:** usar apenas divisões binárias.
5. **–R:** usar redução de erro na confiança.

6. **–L:** não limpar depois que a árvore foi construída.
7. **–A:** Laplace para a suavização das probabilidades previstas.

4.6.2 Os parâmetros do otimizador AG

1. ***geracoes***: caso esse parâmetro seja diferente de zero, ele ativa o otimizador AG. É usado também para indicar o número de gerações do AG. Caso seja igual a zero ele ignora o otimizador AG e executa apenas o PROP.
2. Probabilidade de mutação (***probMut***): controla a probabilidade de um indivíduo da população sofrer mutação.
3. Probabilidade de criação de novos filhos***(probGerarFilhos)***: é a probabilidade do cruzamento de dois indivíduos produzir novos indivíduos.
4. Tamanho da população***(gaTamanhoPopulacao)***: é o tamanho máximo da população do AG.

5 RESULTADOS E DISCUSSÕES

Com o objetivo de verificar as potencialidades dos sistemas propostos (O PROP-AG e sua variação PROP que não usa o Algoritmo Genético), foram comparados ao sistema Aleph, ao Sistema EDA-ILP (PITANGUI, 2013), ao Sistema REDA-ILP (PITANGUI, 2013) e ao sistema HEDA-ILP (PITANGUI, 2013). O Aleph foi escolhido para os testes devido a sua grande flexibilidade (MUGGLETON, 2007) e popularidade, como também devido ao fato de que ele é capaz de obter bons resultados em uma grande gama de problemas. Os sistemas EDA-ILP, REDA-ILP, HEDA-ILP são sistemas apresentados em Pitangui (PITANGUI, 2013), que conseguem obter resultados competitivos com sistemas considerados estado da arte.

Com objetivo de avaliar os sistemas, foram utilizadas 4 bases de dados do mundo real, a saber: alzheimers-amine (Amine), alzheimers-toxic (Toxic), alzheimers-acetyl (Acetyl), alzheimers-memory (Memory) (KING et al., 1992). Esses problemas tratam da predição da atividade bioquímica das variantes da Tacrina (uma droga para o tratamento do Alzheimer) (KING et al., 1992). Estas bases de dados contêm conhecimento preliminar sobre as propriedades físicas e químicas de vários compostos. O objetivo é comparar várias propriedades bioquímicas, tais como: baixa toxicidade (**Toxic**), alta inibição da acetilcolinesterase (**Acetyl**), boa reversão da perda de memória induzida pela escopolamina (**Memory**), e a inibição da recaptação da amina (**Amine**).

5.1 Metodologia Experimental

Todos os sistemas foram avaliados em relação à acurácia obtida (usando t-testes (corrigidos) de Student (NADEAU; BENGIO, 2003) com $p < 0,05$) e em relação ao tempo computacional gasto (medido em segundos). Todos os experimentos foram realizados usando-se validação cruzada em 10 *folds*, sendo que os resultados apresentados para cada sistema (acurácia e tempo de execução) são a média dos resultados destes *folds.* Para os sistemas EDA-ILP, REDA-ILP, HEDA-ILP e PROP-AG, devido aos seus comportamentos estocásticos, o resultado em cada um dos 10 *folds* foi obtido calculando-se a média de 10 execuções em cada *fold.*

Utilizou-se a validação cruzada interna com 10 *folds* para determinar os valores dos parâmetros para todos os sistemas, exceto para o Aleph.

Para o Aleph, sua configuração foi retirada dos trabalhos (HUYNH, MOONEY 2008), (MUGGLETON et al. 2010-a), e (MUGGLETON et al. 2010-b), uma vez que tais trabalhos sugerem bons valores de parâmetros para o Aleph para as bases de dados consideradas. Optou-se por adotar os valores de parâmetros sugeridos nesses trabalhos em vez de usar a validação cruzada para determiná-los, pois é muito difícil adotar tal procedimento devido à grande quantidade de parâmetros dos sistemas. Assim, adotaram-se os valores de parâmetros sugeridos já que o Aleph obtém bons resultados quando parametrizado como indicado.

Para o PROP-AG, os parâmetros do otimizador adotam os seguintes valores: probabilidade de mutação de 8%; probabilidade da criação de novos filhos de 80%; tamanho máximo da população de 30 indivíduos e 100 (cem) gerações. Já para o C4.5 foram usados os parâmetros padrão (*default*) do sistema.

Para os sistemas EDA-ILP, REDA-ILP e HEDA-ILP, suas configurações foram retiradas do trabalho (PITANGUI, 2013), uma vez que esse trabalho sugere bons valores de parâmetros para as bases de dados consideradas.

A Tabela 5.1 apresenta os resultados obtidos para os problemas do mundo real, sendo que (Acc) representa a acurácia obtida e (T) é o tempo (em segundos) usado para executar os sistemas.

Tabela 5.1: Resultados Experimentais para as bases de dados Acetyl, Amine e Memory, Toxic.

	ALEPH		**HEDA-ILP**		**REDA-ILP**		**EDA-ILP**		**PROP**		**PROP-AG**	
	Acc	**T(s)**	**Acc**	**T(s)**	**Acc**	**T(s)**	**Acc**	**T(s)**	**Acc**	**T(s)**	**Acc**	**T(s)**
Acetyl	63,6	116,2	69,8	40,2	66,4	21,5	63,4	47,8	69,5	**6,6**	**70,3**	9284,8
Amine	69,7	56,6	**79,5**	31,9	72,2	10,1	71,3	41,3	**79,5**	**4,6**	79,4	3955,8
Memory	63,8	31,2	**69,9**	33,4	66,1	11,6	64,8	23,2	62,4	**3,1**	62,5	2665,5
Toxic	78,3	27,4	**81,5**	36,7	74,0	14,8	74,7	37,4	78,7	**3,9**	79,0	4166,0

A tabela mostra que os melhores resultados quanto a acurácia estão concentrados em torno do algoritmo HEDA-ILP, e os melhores resultados quanto ao tempo de execução estão concentrado em torno do algoritmo PROP. A seguir discutiremos os

resultados para cada uma das bases testadas. Visando uma melhor apresentação dos resultados obtidos será feita uma comparação através de gráficos. Como a versão PROP do algoritmo PROP-AG mostra-se estatisticamente equivalente quanto à acurácia, será usado somente o PROP, assim a visualização torna-se mais simples, já que o tempo de execução do PROP-AG é extremamente maior que os demais. A acurácias dos sistemas é somente comparada com a acurácia do PROP, pois como dito antes, PROP e PROP-AG são estatisticamente equivalentes.

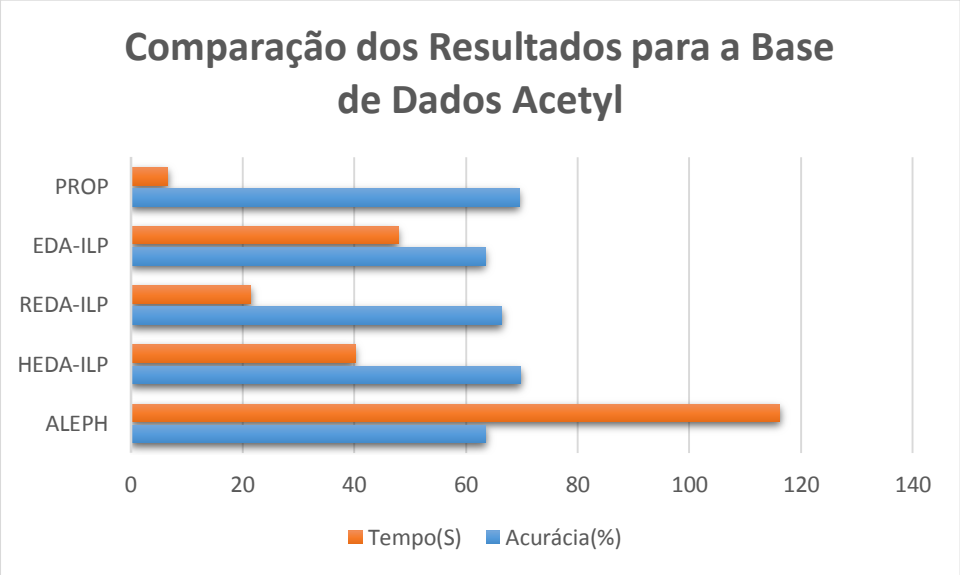

Figura 5.1: Comparação dos Resultados para a Base de Dados Acetyl.

Analisando a Figura 5.1, pode-se perceber que o tempo de execução do PROP é inferior a todos os outros algoritmos comparados. Infelizmente o mesmo não acontece para o PROP-AG que obtém tempo de execução muito superior a todos os outros algoritmos. Para a base de dados Acetyl, PROP obtém acurácia estatisticamente superior aos algoritmos EDA-ILP, REDA-ILP e ao Aleph, e obtém acurácia estatisticamente equivalente ao algoritmo HEDA-ILP.

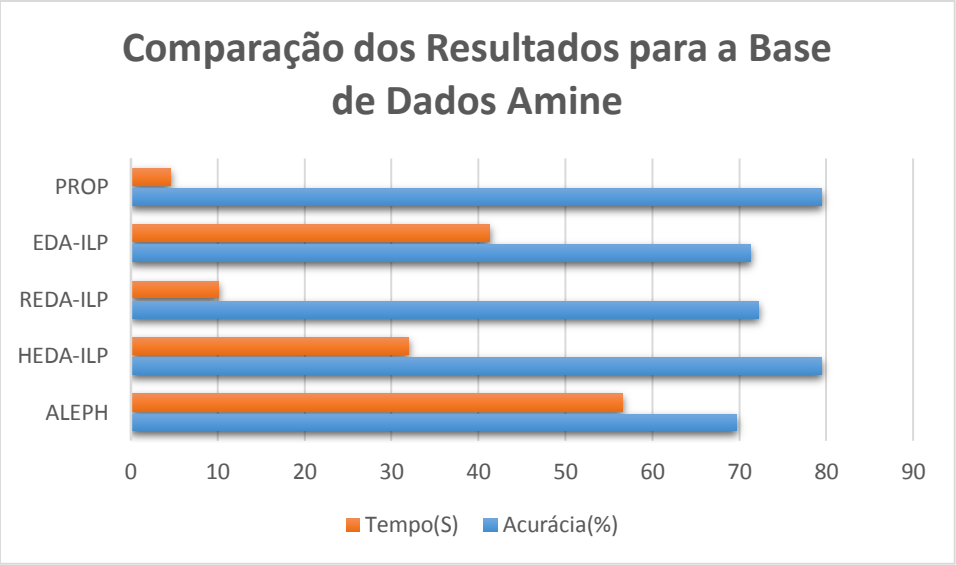

Figura 5.2: Comparação dos Resultados para a Base de Dados Amine.

Para a base de dados Amine, analisando a Figura 5.2, pode-se perceber que o tempo de execução do PROP é inferior a todos os outros algoritmos comparados, e o tempo para o PROP-AG é extremamente superior a todos os outros algoritmos. Para a base de dados Amine, o algoritmo PROP obtém acurácia estatisticamente superior aos algoritmos EDA-ILP, REDA-ILP e ao Aleph, e obtém acurácia estatisticamente equivalente ao algoritmo HEDA-ILP.

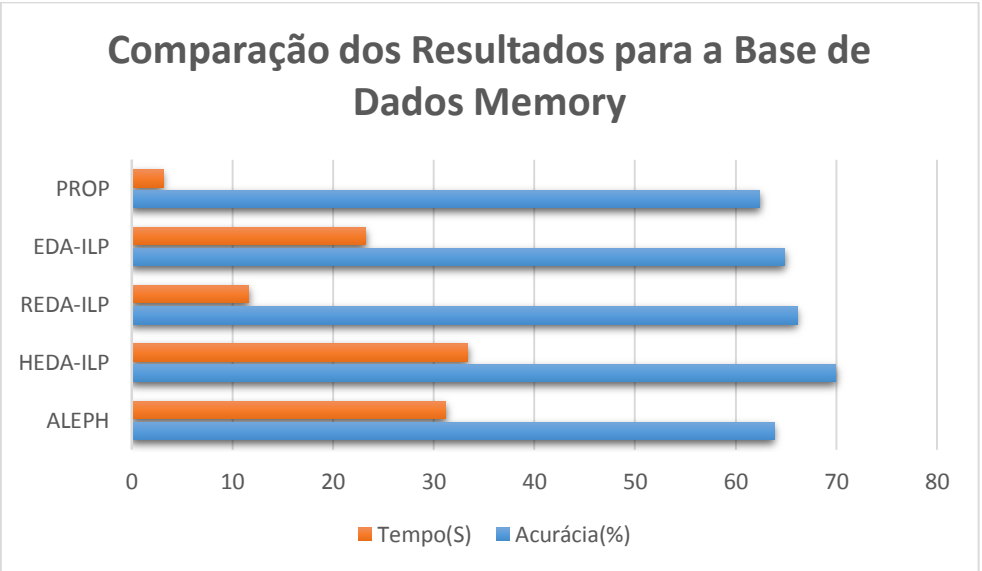

Figura 5.3: Comparação dos Resultados para a Base de Dados Memory.

Para a base de dados Memory, cujos resultados foram representados na Figura 5.3, também pode-se perceber que o tempo de execução do PROP é inferior a todos os outros algoritmos comparados, e o tempo para o PROP-AG é extremamente superior a todos os outros algoritmos. Para essa base ainda, o algoritmo PROP obtém acurácia estatisticamente inferior ao algoritmos EDA-ILP, HEDA-ILP e ao REDA-ILP, e obtém acurácia estatisticamente equivalente ao algoritmo Aleph.

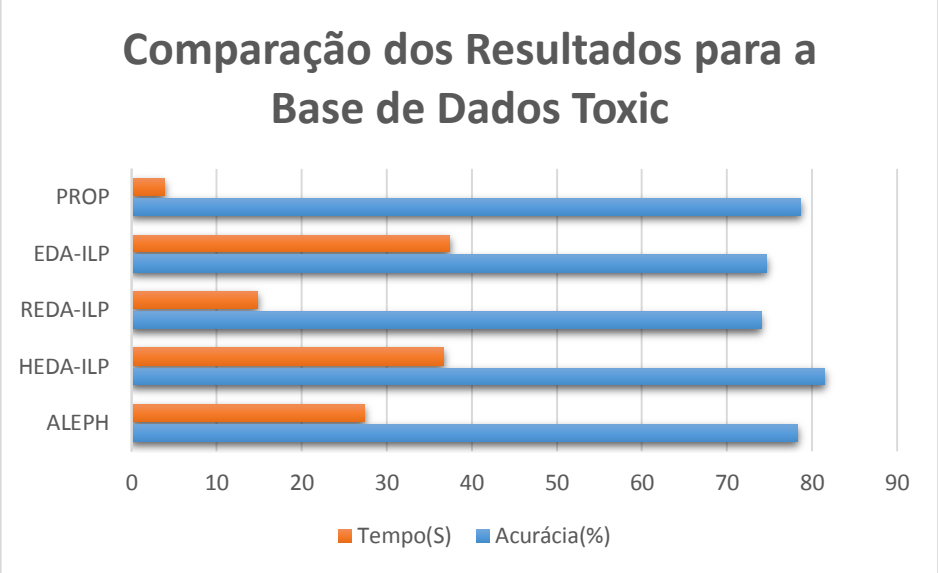

Figura 5.4: Comparação dos Resultados para a Base de Dados Toxic.

Já para a base de dados Toxic, apresentada na Figura 5.4, observou-se que o tempo de execução do PROP é inferior a todos os outros algoritmos comparados, e o tempo para o PROP-AG é extremamente superior a todos os outros algoritmos. Para essa base ainda, o algoritmo PROP obtém acurácia estatisticamente superior ao algoritmos EDA-ILP e ao REDA-ILP, obtém acurácia estatisticamente equivalente ao algoritmo Aleph e obtém acurácia estatisticamente inferior ao HEDA-ILP.

6 CONCLUSÃO

Primeiramente o otimizador, AG, não auxiliou o sistema a encontrar melhores resultados. Isso ocorreu pelo fato de o AG encontrar uma solução razoável, e acabar ficando preso a ela, sem explorar o restante do domínio de soluções possíveis. Para contornar esse problema o AG precisaria de mais tempo de execução, o que não seria viável nesse momento pois, o otimizador utiliza como função de aptidão um algoritmo (C4.5) que que é usada repetidas vezes, e seu tempo de execução é considerável.

Assim, os melhores resultados foram alcançados pelo sistema PROP, o qual os resultados são estatisticamente similares aos PROP-AG. O sistema PROP, com relação a acurácia, mostra-se tão bom quanto a sistemas considerados estado da arte, obtendo resultados estatisticamente similares. Algumas vezes mostra melhor acurácia do que alguns sistemas considerados estado da arte, obtendo resultados de até 8% que os outros sistemas avaliados.

Da mesma maneira, em relação ao tempo de execução, o PROP foi melhor que todos os sistemas avaliados. Onde consegue obter tempo de execução, por exemplo, 10 vezes menor para algumas bases em relação ao Aleph.

Trabalhos Futuros

A seguir são apresentados alguns trabalhos futuros a serem realizados.

- **Implementação da medida de Gini:** pode-se usar a medida de Gini para criar um segundo filtro de atributos que possa ser usado paralelamente com o Ganho de Informação, ou separadamente para analisar os novos resultados. (BREIMAN et al., 1984)
- **Implementação do C4.5 localmente**: pode-se implementar o algoritmo C4.5 localmente para diminuir o tempo gasto com chamadas ao sistema (logo que o algoritmo é um algoritmo externo pertencente ao Weka), isso seria interessante pois a função de aptidão utilizada pelo AG é o próprio C4.5. Assim poderíamos aumentar o número de gerações do otimizador, o que é custa muito cara nesse momento (em tempo de execução), para que ele possa explorar uma parte maior do domínio de resultados.

- **Implementação do CART:** pode-se implementar o algoritmo CART como outra forma de construir árvores de decisão, e comparar a nova estratégia com a usada atualmente. O CART também poderia funcionar como uma função de aptidão para o sistema atual. CART (BREIMAN et al., 1984).

REFERÊNCIAS BIBLIOGRÁFICAS

BATISTA, G. E. A. P. A., Pré-processamento de Dados em Aprendizado de Máquina Supervisionado, 2003.

BLOCKEEL, H., DE RAEDT, L., Top-down induction of first order logical decision trees. Artificial Intelligence, 101(1-2):285-297, 1998.

BREIMAN, L., FRIEDMAN, J. H., OLSHEN, R. A., & STONE, C. J. Classificationand RegressionTrees. Wadsworth, 1984.

CATARINA, A.S., Algoritmos evolutivos aplicados ao processo de análise de dadosgeográficos. São José dos Campos: [s,n], 2005.

FRANCA, M.; ZAVERUCHA, G.; Garcez, A. S.. Fast relational learning using bottom clause propositionalization with artificial neural networks. Machine Learning, v. 00, p. 1-26, 2013.

GOADRICH, M., Adaptively Finding and Combining First-Order Rules for Large, Skewed Data Sets. PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, 2009.

HALMENSCHLAGER, C. Um Algoritmo para Indução de Árvores e regras de Decisão, 2002.

HUYHN T., MOONEY R., Discriminative Structure and Parameter Learning for Markov Logic Networks. In Proceedings of the 25th International Conference on Machine Learning (ICML-2008), Helsinki, Finland, 2008.

LLOYD, J., 1987, Foundations of Logic Programming, 2 ed., Springer Verlag.

KING, R.D., MUGGLETON, S., STERNBERG, M., Drug Design by Machine Learning: The Use of Inductive Logic Programming to Model the Structure-Activity Relationships of Trimethoprim Analogues Binding to Dihydrofolate Reductase. Proceedings of the National Academy of Sciences 89(23), 11,322–11,326, 1992.

MENEZES, G., C., Htilde-rt: Um algoritmo de Aprendizado de Árvores de regressão de Lógica de Primeira-Ordem para Fluxos de Dados Relacionais, 2011.

MITCHELL, T., Machine learning. New York: McGraw-Hill, 1997.

MUGGLETON, S., “Inductive Logic Programming”, New Generation Computing, v. 13, n. 4, pp. 245-286, 1991.

MUGGLETON, S., “Inverse Entailment and Progol”, New Generation Computing, v. 13,n. 4, pp. 245-286, 1995.

MUGGLETON, S., Learning from positive data. In Muggleton, S., editor, Proceedings of the 6th International Workshop on Inductive Logic Programming, pages 225–244. Stockholm University, Royal Institute of Technology, 1996.

MUGGLETON, S., TAMADDONI-NEZHAD A., QG/GA: A stochastic search approach for Progol. In Proceedings of the 16th International Conference on Inductive Logic Programming, LNAI 4455, pages 37-39. Springer-Verlag, 2006.

MUGGLETON, S., TAMADDONI-NEZHAD A., QG/GA: A stochastic search for Progol. Machine Learning, 70(2-3):123-133, 2007.

MUGGLETON, S., SANTOS, J., TAMADDONI-NEZHAD, A., Top Log: ILP using a logic program declarative bias. In Proceedings of the International Conference on Logic Programming 2008, LNCS 5366, pages 687-692. Springer-Verlag, 2010-a.

MUGGLETON, S., SANTOS, J.C.A., TAMADDONI-NEZHAD, A., ProGolem: A System Based on Relative Minimal Generalisation. In: Proceedings of the 1th International Conference on Inductive Logic Programming (ILP-09), pp. 131–148, 2010-b.

NADEAU C., BENGIO Y., "Inference for the Generalization Error", Machine Learning 52(3) pp. 239-281, 2003.

PITANGUI, C., Programação em Lógica Indutiva Através de Algoritmo Estimador de Distribuição e Cláusula mais Específica, 2013.

QUINLAN, J. R., Induction of decision trees. Machine Learning, 1(1):81-106, 1986.

QUINLAN, J., Learning logical definition from relations. *Machine Learning*5:239.266, 1990.

QUINLAN, J. R., C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

ROBINSON, A., A Machine-Oriented Logic Based on the Resolution Principle, Journal of the ACM, vol.12, pp. 23-41, 1965.

RUSSELL, S., NORVIG, P., Artificial Intelligence: A Modern Approach, Prentice-Hall, Englewood Cliffs, NJ, 2003.

SEBESTA, R., Concepts of programming languages (3rd ed.), Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, 1996.

SRINIVASAN. A., 2003, Disponível em: <http://www.comlab.ox.ac.uk/activities/machinelearning/Aleph/> Acesso em: 05/06/2013.

WEKA, Disponível em:

<http://www.cs.waikato.ac.nz/ml/weka/documentation.html>Acesso em: 07/06/2013.

AUTORIZAÇÃO

Autorizo a reprodução e/ou divulgação total ou parcial do presente trabalho, por qualquer meio convencional ou eletrônico, desde que citada a fonte.

_____

Felipe Costa Rodrigues

fcostarodrigues@gmail.com

Universidade Federal dos Vales do Jequitinhonha e Mucuri

Campus JK - Diamantina/MG

Rodovia MGT 367 - Km 583, nº 5000; Alto da Jacuba CEP 39100-000