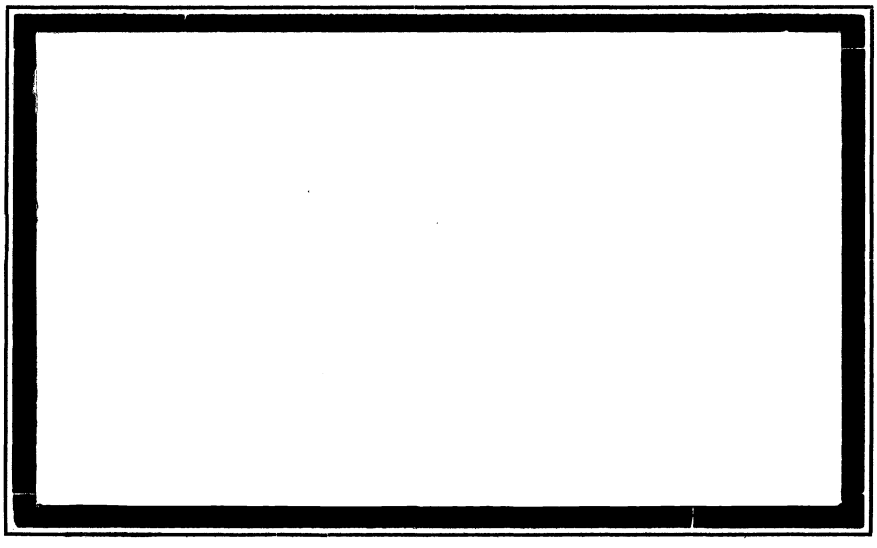
UNIVERSIDADE DE SÃO PAULO



RELATÓRIOS TÉCNICOS

Instituto de Ciências Matemáticas de São Carlos

Instituto de Ciências Matemáticas de São Carlos

ISSN-0109-2569

Árvores de decisão binárias:

Descrição e implementação de algoritmos para a construção e poda no aprendizado indutivo de máquina

CLAUDIA APARECIDA MARTINS

MARIA CAROLINA MONARD

Nº 17

RELATÓRIOS TÉCNICOS DO ICMSC

SÃO CARLOS
fev. / 1994

SYSNO 859336
DATA / /
ICMC - SBAB

Árvores de Decisão Binárias: Descrição e Implementação de Algoritmos para a Construção e Poda no Aprendizado Indutivo de Máquina[1]

Claudia Aparecida Martins
Universidade de São Paulo
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail:cam@icmsc.usp.br

Maria Carolina Monard
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística
Caixa Postal 668, 13560-970 - São Carlos, SP
e-mail: mcmonard@icmsc.usp.br

Sumário

Aprendizado Indutivo de Máquina por Exemplos é uma das estratégias utilizada em Aprendizado de Máquina. Esta estratégia induz a descrição de conceitos através de processos de generalização e especialização realizados sobre exemplos observados. A família TDIDT adota esta estratégia. Os algoritmos desta família se caracterizam por construir árvores de decisão para tarefas de classificação.

Neste trabalho é apresentado e discutido a implementação de um sistema que constrói árvores de decisão a partir de um conjunto de exemplos, que possuem atributos com valores binários, utilizando a metodologia da família TDIDT para o aprendizado indutivo não incremental. É apresentado, também, a implementação de um método de pós-poda para a árvore gerada, denominado Método de Redução do Erro bem como alguns procedimentos que são úteis para analisar a complexidade da árvore gerada.

Janeiro de 1994

[1]Trabalho realizado com o auxílio do CNPq e FAPESP de acordo com o Proc. Nº 92/1586-0 e Proc. Nº 92/2151-8.

Conteúdo

1 Introdução **1**

2 Família TDIDT **2**

2.1 Algoritmo Geral dos Sistemas da Família TDIDT 2

2.2 Algoritmo para Atributos com Valores Binários 4

3 Considerações Gerais sobre a Implementação da Árvore de Decisão para Atributos com Valores Binários **5**

3.1 Descrição da Implementação 6

3.1.1 Procedimentos: Definição e Discussão 7

3.1.2 Exemplos de Execução 14

3.1.3 Listagem do Programa 16

4 Poda em Árvores de Decisão **21**

4.1 Método de Poda de Redução do Erro 22

4.2 Descrição da Implementação 24

4.2.1 Procedimentos: Definição e Discussão 24

4.2.2 Exemplos de Execução 27

4.2.3 Listagem do Programa 28

5 Procedimentos Utilitários **32**

6 Conclusões **33**

Referências **35**

1 Introdução

Uma das linhas de pesquisa em Inteligência Artificial que está sendo amplamente estudada é Aprendizado de Máquina. Aprendizado de Máquina — AM — é uma área de pesquisa que estuda métodos computacionais apropriados para a aquisição de novos conhecimentos, novas habilidades e novas formas de organização do conhecimento já existente.

Existe uma família de sistemas de propósito geral, pertencentes à área de Aprendizado de Máquina por exemplos, que tem como característica a construção de árvores de decisão, através de processos indutivos, utilizando uma metodologia Top Down. Esta família é denominada TDIDT — Top Down Induction of Decision Tree —, e representam o conhecimento adquirido (conceitos) processando exemplos observados, através de árvores de decisão.

No ICMSC-USP está sendo desenvolvido um projeto de AM para geração de regras de conhecimento. Dentro desse projeto foi implementado, na linguagem de programação Prolog, um procedimento geral não incremental da família TDIDT tal que, dado um conjunto de exemplos, com qualquer número de atributos, bem como qualquer número de valores discretos de atributos, constrói a árvore de decisão correspondente [Castineira 91]. Foram também implementados dois algoritmos para realizar a pós-poda da árvore de decisão gerada [Nicoletti 92a].

Ainda dentro desse projeto, foram realizadas diversas pesquisas de aprendizado indutivo construtivo de máquina [Nicoletti 92b,Nicoletti 93a,Nicoletti 93b]. Indução Construtiva pode ser definida como uma forma de indução que gera novos atributos não presentes no conjunto de treinamento inicial, isto é, é um processo que muda o espaço de descrições — algumas vezes os atributos iniciais usados para descrever os exemplos não são diretamente relevantes para descrever um conceito mas, quando combinados, convenientemente, geram novos atributos (*features*), podendo tornar-se altamente representativos para expressar um conceito. O processo de construção de *features* pode ser direcionado e controlado pelo usuário ou automaticamente conduzido pelo sistema.

Os trabalhos e implementações já desenvolvidos no ICMSC referentes à construção da árvore de decisão, pós-poda da árvore, aprendizado indutivo construtivo e outros, utilizam uma mesma estrutura de dados para representar a árvore de decisão. A estrutura utilizada para representar a árvore é bem geral e permite representar qualquer árvore de decisão, mas não é simples. Entretanto, no caso de construção automática de *features*, onde as informações que se tem sobre os atributos é que eles estão presente ou ausente, por exemplo, já se sabe que essa árvore é uma árvore binária. Neste caso, utilizando uma estrutura de dados específica para este problema, as implementações correspondentes à construção da árvore, construção de *features* utilizando diferentes algoritmos e posteriormente a poda da árvore, podem ser bastante simplificadas, permitindo assim uma maior facilidade para pesquisar e implementar, no futuro, outros algoritmos e operadores construtivos.

O objetivo deste trabalho é descrever em detalhes a implementação Prolog de um algoritmo para a construção de uma árvore de decisão utilizando a estrutura de árvore

binária. Isto é, a implementação cobre os casos de exemplos com apenas dois valores de atributos. A implementação de um mecanismo de pós-poda e procedimentos necessários para a análise estatística da árvore gerada, são também descritos.

O trabalho está organizado da seguinte forma: a seção 2 descreve o algoritmo geral da família TDIDT, assim como o algoritmo específico para atributos com valores binários. A seção 3 descreve a implementação da construção da árvore de decisão através de exemplos para esse algoritmo específico. Os procedimentos mais importantes são descritos em detalhe, bem como exemplos de execução.

A seção 4 discute poda em árvores de decisão e mostra a implementação do método de pós-poda, denominado Método de Poda de Redução do Erro, para árvores binárias de decisão. Os procedimentos mais importantes são descritos em detalhe, bem como exemplos de execução. A seção 5 mostra a implementação de alguns procedimentos utilitários importantes para analisar a complexidade da árvore de decisão gerada e, finalmente, as conclusões são apresentadas na seção 6.

2 Família TDIDT

Top Down Induction of Decision Tree - TDIDT - é uma família de sistemas de Aprendizado de Máquina por exemplos, que são caracterizados por representarem o conhecimento adquirido através de árvores de decisão. A característica desta família é utilizar uma metodologia Top Down, isto é, a árvore é construída começando da raiz e descendo até as folhas.

Na construção da árvore de decisão é levado em consideração que para ramificar cada nó da árvore é necessário escolher um determinado atributo; desta forma, os exemplos correspondentes a esse nó serão agrupados em ramos de acordo com os valores do atributo selecionado. O intuito é gerar a menor árvore de decisão que classifica corretamente todos os exemplos. Para atingir este objetivo é necessário escolher, em cada passo, o atributo mais relevante, isto é, o atributo que melhor particiona os exemplos segundo o valor da classe. A escolha do atributo mais promissor é realizada utilizando uma função denominada *função de avaliação*. Existem diversas funções de avaliação; geralmente elas estão baseadas na teoria da informação ou em medições estatísticas [Mingers 89a].

O processo de escolha de um atributo — utilizando uma determinada função de avaliação — e ramificação dos exemplos segundo os valores desse atributo é repetido para cada nó da árvore enquanto não se verifique a *condição de parada* no nó. Geralmente a condição de parada é a de que todos os exemplos do nó — ou a maioria deles — pertençam à mesma classe. Se esta condição for satisfeita o nó torna-se uma folha rotulada com o valor da classe.

2.1 Algoritmo Geral dos Sistemas da Família TDIDT

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

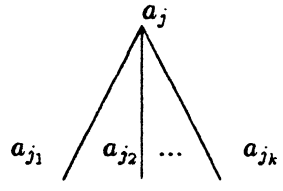
Dados um conjunto de exemplos de aprendizado E;
uma condição de parada $t(E)$;
uma função de avaliação $aval(E, atributo)$

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

então retorne o valor da classe

caso contrário

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



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

O critério de parada t pode ser definido para

- construir a árvore de decisão que classifica exatamente todos os elementos do conjunto de aprendizado em domínios completos — sistema ID3 [Quinlan 79];
- decidir pela não expansão da árvore quando a evidência for insuficiente nos exemplos fornecidos; este mecanismo é denominado pré-poda da árvore de decisão — usado pelos sistemas C4 [Quinlan 86] e Assistant-86 [Cestnik 87].

A função de avaliação *aval* pode ter diversas definições. As diversas condições de parada e funções de avaliação dão origem aos vários algoritmos desta família. Por exemplo, o sistema ID3 utiliza a entropia como função de avaliação dos atributos, e seu critério

[2] Atributo mais relevante para realizar a ramificação, segundo a função de avaliação.

de parada é que todos os exemplos no nó pertençam à mesma classe. Embora esta abordagem dê bons resultados neste sistema, nada garante que esses sejam os melhores resultados. Outro critério ou função de avaliação poderiam ser utilizados.

2.2 Algoritmo para Atributos com Valores Binários

Neste trabalho são considerados atributos com apenas dois valores possíveis, isto é, um atributo está presente (valor do atributo = 1) ou ausente (valor do atributo = 0) no conjunto de treinamento. Neste caso, o algoritmo geral da família TDIDT pode ser simplificado no que se refere à ramificação da árvore, da seguinte forma:

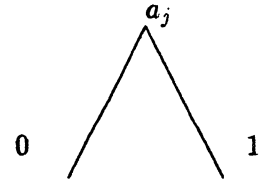
Dados um conjunto de exemplos de aprendizado E;
uma condição de parada $t(E)$;
uma função de avaliação $aval(E, atributo)$

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

Então retorne o valor da classe

Caso contrário

1. para cada atributo a_i determine o valor da função $aval(E, a_i)$.
   Seja a_j o atributo que possui o melhor valor de $aval(E, a_i)$.
2. a_j assume somente os valores 0 e 1, crie o seguinte nó da árvore:



3. particione os exemplos do conjunto E nos subconjuntos E_0 (correspondente aos valores de $a_j = 0$) e E_1 (correspondente aos valores de $a_j = 1$).
4. aplique o algoritmo recursivamente para os subconjuntos E_0 e E_1.

A função de avaliação — $aval(E, a_i)$ — implementada neste trabalho é a entropia. Para calcular a entropia de um determinado atributo binário, é necessário calcular a entropia de suas duas ramificações. Se um ramo contém observações de uma única classe, pode-se dizer que ele está ordenado e possui entropia zero; sua desorganização é nula. Caso contrário, uma observação pode ser classificada em 2 classes diferentes: c_0 (classe = 0) e c_1 (classe = 1). Considerando que a probabilidade de um objeto pertencer à classe c_i é $p(i)$, então a entropia de classificação do ramo é dada por

$$En(A = v_j) = -\sum_{i=0}^{1} p(i)\log_2 p(i) \quad (1)$$

onde $A = v_j$ significa que o atributo A tem o valor v_j com $v_j \in \{0,1\}$. Isto é, $En(A = v_j)$ é a entropia do ramo correspondente ao valor v_j do atributo A. Para calcular a entropia total de um atributo deve-se considerar a entropia de cada um dos ramos correspondentes a este atributo. Seja:

M o número total de observações,

En_0 **e** En_1 as entropias dos dois ramos 0 e 1 respectivamente, associados ao atributo escolhido, e

N_0 **e** N_1 o número de observações de cada ramo.

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

$$En(A) = \sum_{i=0}^{1} E_i \frac{N_i}{M} \quad (2)$$

Assim, para gerar árvores de decisão mais simples, em cada nó da árvore deve-se escolher o atributo de menor entropia para continuar ramificando a árvore.

3 Considerações Gerais sobre a Implementação da Árvore de Decisão para Atributos com Valores Binários

Esta implementação foi realizada na linguagem de programação Prolog [Arity 90]. A estrutura de dados utilizada é listas e árvores binárias. A árvore binária é justificada pelo fato dos atributos assumirem apenas dois valores possíveis, isto é, o atributo poderá estar presente ou ausente no conjunto de treinamento. A estrutura geral da árvore de decisão construída é:

$$arv(SubarvEsq, MelhorAtributo, SubarvDir)$$

onde $SubarvEsq$ e $SubarvDir$ são subárvores que classificam os exemplos para os quais o valor de $MelhorAtributo$ é 0 e 1 respectivamente.

A implementação realizada leva em consideração várias situações que podem acontecer durante o processo de construção da árvore, descritas a seguir:

1. Todos os exemplos pertencem à mesma classe. Neste caso a subárvore correspondente consiste de um simples nó rotulado com o valor da classe.

2. Durante o processo de construção da árvore, não é possível classificar exatamente exemplos que tenham atributos com o mesmo valor e que pertencem a classes diferentes. Neste caso, após a manipulação de todos os exemplos do conjunto de aprendizado, a estratégia adotada nesta implementação é a de rotular essa folha com o valor da classe mais provável.

3. Um nó folha é rotulado com o átomo `null` quando para um determinado valor de atributo não existirem exemplos no conjunto de treinamento que sejam classificados pelo ramo correspondente.

3.1 Descrição da Implementação

Em Prolog, um *procedimento* consiste do conjunto de cláusulas referente à uma mesma relação ou predicado. Predicados são notados pelo seu nome e pela sua aridade. Por exemplo, a notação `abc/4` refere-se ao predicado de nome `abc` que tem 4 argumentos, ou seja, aridade 4. Esta convenção é utilizada neste texto. Além disso, quando for conveniente, os argumentos dos procedimentos serão descritos da seguinte forma[3]:

[+] $< arg_n >$: o n-ésimo argumento é de entrada

[−] $< arg_n >$: o n-ésimo argumento é de saída

[?] $< arg_n >$: o n-ésimo argumento é de entrada/saída

Antes de ativar o procedimento que é realmente responsável pela construção da árvore, é necessário ter na memória o conjunto de exemplos e o nome dos atributos presentes nesses exemplos. O conjunto de exemplos deve estar representado por uma lista de listas, onde cada uma dessas listas corresponde a um exemplo. Os elementos dessas listas são estruturas da forma `<nome do atributo>:<valor>` ou `classe:<valor>`.

Os atributos presentes nesses exemplos devem estar representados por uma lista de átomos onde cada átomo é o nome de um atributo. A implementação realizada considera que os atributos podem assumir valores no conjunto {0,1} e a classe é reconhecida pelo nome específico `classe`. Se este não for o caso, o conjunto de exemplos deve ser pré-processado. Além disso, todos os atributos devem comparecer em cada um dos exemplos.

A fim de exemplificar, considere que o seguinte conjunto de exemplos vai ser utilizado para construir a árvore de decisão.

[3]Observar que a instanciação dos argumentos, descrita neste trabalho, refere-se ao caso da árvore estar sendo construída.

Atributos			Classe
x1	x2	x3	
sim	sim	sim	sim
não	não	não	sim
sim	não	não	não
sim	não	sim	sim

Ele deve ser transformado para uma estrutura de lista de listas de exemplos da seguinte forma

```
Exemplos = [[x1 : 1,x2 : 1,classe : 1,x3 : 1],
            [x1 : 0,x2 : 0,classe : 1,x3 : 0],
            [x1 : 1,x2 : 0,classe : 0,x3 : 0],
            [x1 : 1,x2 : 0,classe : 1,x3 : 1]].
```

e a lista de atributos deve ser representada como

```
Atributos = [x1,x2,x3].
```

É bom ressaltar que a ordem dos elementos em ambas as listas é irrelevante para a implementação realizada. Isto é, o atributo `x1`, por exemplo, pode comparecer em diferentes posições em cada um dos exemplos considerados. Entretanto, se `classe:<valor>` comparecer sempre como o primeiro elemento nas listas de exemplos, o tempo de execução do algoritmo implementado diminui.

3.1.1 Procedimentos: Definição e Discussão

O procedimento no nível mais alto — `arvore_bin/2` — está definido da seguinte forma:

Procedimento 3.1.1.1 arvore_bin/2

```
arvore_bin(Arvore,Flag) :-
    atributos(Atributos),
    exemplos(Exemplos),
    forme_arv_bin(Exemplos,Atributos,Arvore,Flag).
```

onde os dois primeiros predicados: `atributos/1` e `exemplos/1` referem-se aos procedimentos que fazem o pré-processamento dos dados de entrada. Eles devem ser definidos pelo usuário já que estão ligados à forma e o meio onde esses exemplos estão armazenados. Ambos os procedimentos devem ser bem sucedidos com `Atributos` instanciado com uma lista que contém todos os atributos dos exemplos considerados, e `Exemplos` instanciado em uma lista de lista de exemplos, como explicado anteriormente.

Considerando o exemplo anterior, ambas as variáveis serão instanciadas da seguinte forma:

```
Exemplos = [[x1 : 1,x2 : 1,classe : 1,x3 : 1],
            [x1 : 0,x2 : 0,classe : 1,x3 : 0],
            [x1 : 1,x2 : 0,classe : 0,x3 : 0],
            [x1 : 1,x2 : 0,classe : 1,x3 : 1]].

Atributos = [x1,x2,x3].
```

A saída de **arvore_bin/2** consiste na árvore gerada e um *flag* indicando se a árvore é exata ou não. Por exemplo, para este conjunto de exemplos e atributos tem-se o seguinte resultado para a interrogação `?- arvore_bin(Arvore,Flag).`

```
Arvore = arv(1,x1,arv(0,x3,1))
Flag = exata
```

O terceiro procedimento — **forme_arv_bin/4** — está definido da seguinte forma:

Procedimento 3.1.1.2 `forme_arv_bin/4`

```
forme_arv_bin(E,A,T,F) :-
    forme_arvore_bin(E,A,T,F),
    ifthen( var(F), F=exata).
```

Ele é responsável por ativar o procedimento que realmente deflagra a construção da árvore de decisão — **forme_arvore_bin/4** —, bem como instanciar o *flag* F no caso da árvore ser exata, como mostra, esquematicamente, a Figura 1.

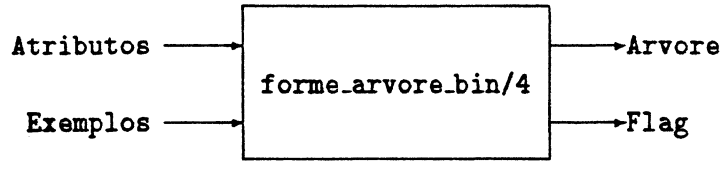


Figura 1: Procedimento Responsável pela Construção da Árvore de Decisão

Procedimento 3.1.1.3 `forme_arvore_bin/4`

Os argumentos de **forme_arvore_bin/4** são:

[+] $< arg_1 >$: lista de exemplos

[+] $< arg_2 >$: lista de atributos

[−] $< arg_3 >$: árvore de decisão construída a partir dos exemplos

[−] $< arg_4 >$: flag que indica se a árvore construída é exata

e consta dos seguintes passos:

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

Durante o processo de construção da árvore, a árvore de decisão é representada pela seguinte estrutura geral:

$$arv([[\ldots],[\ldots],\ldots],MelhorAtributo,[[\ldots],[\ldots],\ldots])$$

onde o primeiro e o terceiro argumentos são lista de listas de exemplos sem o atributo *MelhorAtributo*, particionados de acordo com o valor do atributo. Isto é, o primeiro argumento consiste dos exemplos onde o valor de *MelhorAtributo* é 0 e o terceiro argumento os exemplos onde o valor de *MelhorAtributo* é 1 — Figura 2.

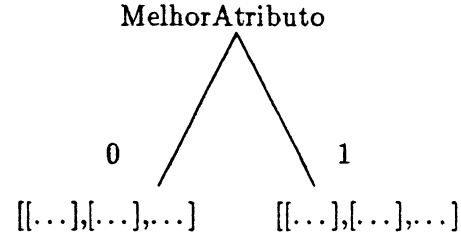


Figura 2: Estrutura da Árvore de Decisão Durante o Processo de Construção

A estrutura final da árvore é representada por nós de decisão que são os atributos, particionados de acordo com seu valor, e nós folhas que representam a classe ou o rótulo `null`. Segue a definição deste procedimento.

```
forme_arvore_bin([],_,null,F).

forme_arvore_bin(Exs,_,Classe,F) :-
    mesma_classe(Exs,Classe),
    !.

forme_arvore_bin(Exs,[],Classe,nao_exata) :-
    classe_majoria(Exs,Classe),
    !.

forme_arvore_bin(Exs,Ats,arv(ArvEsq,MelhorAt,ArvDir),F) :-
    melhor_at_bin(Ats,Exs,MelhorAt,NovosAts,[SubEsq,SubDir]),
    forme_arvore_bin(SubEsq,NovosAts,ArvEsq,F),
    forme_arvore_bin(SubDir,NovosAts,ArvDir,F).
```

As três primeiras cláusulas de `forme_arvore_bin/4` constituem o critério de parada. A primeira cláusula trata a situação onde a lista de exemplos está vazia. Neste caso ela rotula o nó folha com o átomo `null`. A segunda cláusula trata a situação em que todos os exemplos do atual conjunto de exemplos pertencem à mesma classe, fato que determina o término da ramificação em questão, e rotula o nó com o valor de tal classe.

A terceira cláusula de `forme_arvore_bin/4` trata a situação onde não se pode mais ramificar a árvore, uma vez que todos os atributos já foram usados — a lista de atributos está vazia — e os exemplos do ramo em questão não pertencem à mesma classe. A árvore obtida é classificada como não exata — o quarto argumento de `forme_arvore_bin/4` é instanciado com o átomo `nao_exata`. Quando isso acontece, determina-se qual é a classe com o maior número de exemplos — `classe_majoritaria/2` — e o nó folha correspondente é rotulado com o valor dessa classe.

A quarta cláusula trata o caso geral onde, primeiramente, o procedimento `melhor_at_bin/5` encontra o melhor atributo, particiona o conjunto de exemplos e aplica o procedimento recursivamente em ambas as subárvores.

Procedimento 3.1.1.4 `melhor_at_bin/5`

O procedimento `melhor_at_bin/5` é utilizado para determinar o melhor atributo de uma lista de atributos, de acordo com um determinado critério — nesta implementação, o critério escolhido foi a entropia[4]. Dado um conjunto de exemplos, além de encontrar o melhor atributo, o procedimento retira-o do conjunto de atributos e particiona o conjunto de exemplos de acordo com os valores — 0 ou 1 — que o melhor atributo pode assumir. Seu último argumento é instanciado com uma lista de dois elementos da seguinte forma:

$$[[E_{0_1}, E_{0_2}, \ldots, E_{0_i}], [E_{1_1}, E_{1_2}, \ldots, E_{1_j}]]$$

onde:

- $[E_{0_1}, E_{0_2}, \ldots, E_{0_i}]$ é a lista de exemplos para os quais o valor do melhor atributo é 0;
- $[E_{1_1}, E_{1_2}, \ldots, E_{1_j}]$ é a lista de exemplos para os quais o valor do melhor atributo é 1.

Os argumentos de `melhor_at_bin/5` são:

[+] $< arg_1 >$: lista de atributos

[+] $< arg_2 >$: lista de exemplos

[−] $< arg_3 >$: o melhor atributo

[4] Devido à modularidade da implementação, se outro critério for escolhido, somente o procedimento `calcule_entropia_bin/2` deve ser redefinido.

[−] < arg_4 >: lista do conjunto de atributos sem o melhor atributo

[−] < arg_5 >: lista de exemplos particionados segundo os valores do melhor atributo

```
melhor_at_bin(Ats,Exs,MelhorAt,NovosAts,[SE,SD]):-
    calcule_bin(Ats,Exs,Lista),
    calcule_entropia_bin(Lista,[MelhorAt,_]),
    retira(MelhorAt,Ats,NovosAts),
    divide_bin(Exs,MelhorAt,SE,SD).
```

A função básica de `melhor_at_bin/5`, composta de uma única cláusula, é invocar quatro predicados. O primeiro predicado `calcule_bin/3` tem como objetivo criar uma lista onde são lembradas informações tais como a quantidade de vezes que o atributo A_n cujo valor é i ($i = 0$ ou 1) e o valor da classe é j ($j = 0$ ou 1) comparece no conjunto de treinamento. Isto será útil para realizar o cálculo da entropia (ou eventualmente, para implementar outra função de avaliação).

Esta informação é utilizada pelo segundo predicado `calcule_entropia_bin/2` para encontrar o melhor atributo — `MelhorAt`. O terceiro predicado `retira/3` é responsável por retirar esse melhor atributo da lista de atributos e, finalmente, o quarto predicado `divide_bin/4` particiona os exemplos em função do valor de `MelhorAt`.

Procedimento 3.1.1.5 `calcule_bin/3`

Os argumentos de `calcule_bin/3` são:

[+] < arg_1 >: lista de atributos

[+] < arg_2 >: lista de exemplos

[−] < arg_3 >: lista de atributos com os valores calculados

```
calcule_bin(Ats,Exs,Lista):-
    lista_incicial_bin(Ats,ListaInicial),
    realiza_calculo_bin(Exs,ListaInicial,Lista),
```

A lista criada pelo procedimento `calcule_bin/3` tem a seguinte forma:

$$[[At_1, A_1 : B_1, C_1 : D_1],[At_2, A_2 : B_2, C_2 : D_2],\ldots,]$$

onde

At_n: é o nome do atributo n

A_n: é o número de vezes que o atributo n comparece no conjunto de exemplos com valor de atributo 0 e o valor da classe 0

B_n: é o número de vezes que o atributo n comparece no conjunto de exemplos com valor de atributo 0 e o valor da classe 1

C_n: é o número de vezes que o atributo n comparece no conjunto de exemplos com valor de atributo 1 e o valor da classe 0

D_n: é o número de vezes que o atributo n comparece no conjunto de exemplos com valor de atributo 1 e o valor da classe 1

Por exemplo, a interrogação `?- calcule_bin(Atributos,Exemplos,L).` ativada com

```
Atributos = [x1,x2,x3].

Exemplos = [[x1 : 1,x2 : 1,classe : 1,x3 : 1],
            [x1 : 0,x2 : 0,classe : 1,x3 : 0],
            [x1 : 1,x2 : 0,classe : 0,x3 : 0],
            [x1 : 1,x2 : 0,classe : 1,x3 : 1]].
```

é bem sucedida com

```
L = [[x1,0:1,1:2],[x2,1:2,0:1],[x3,1:1,0:2]]
```

Após o procedimento `calcule_bin/3`, ser bem sucedido, a lista criada com essas informações é utilizada pelo procedimento `calcule_entropia_bin/2` para realizar o cálculo da entropia e retornar o melhor atributo — aquele que tiver a menor entropia.

Procedimento 3.1.1.6 `calcule_entropia_bin/2`

Para calcular a entropia de um atributo é necessário agrupar todos os exemplos segundo os valores deste atributo, e para cada grupo — ramo — deve-se somar o produto da frequência de cada classe pelo logaritmo desta frequência — equação (1), pg. 4. Estes números, obtidos para cada valor do atributo, ou para cada ramo, devem ser multiplicados pelo número de exemplos do correspondente ramo e somados para compor o valor final da entropia — equação (2), pg. 5. Deve ser observado que o algoritmo deve encontrar o atributo com menor valor de entropia, mas não o verdadeiro valor da entropia. Assim, a divisão pelo número total de exemplos é desnecessária.

Os argumentos de `calcule_entropia_bin/2` são:

[+] $< arg_1 >$: lista criada pelo procedimento `calcule_bin/2`, pg. 11

[−] $< arg_2 >$: lista contendo o menor valor da entropia e seu respectivo atributo

```
calcule_entropia_bin(Lista,W) :-
    calcule_entropia_bin(Lista,L,W).

calcule_entropia_bin([],L,L]).

calcule_entropia_bin([At,A:B,C:D]|R],L,W) :-
    entropia([A,B],E),
    entropia([C,D],I),
    Ent is ((A+C)*E) + ((B+D)*I),
    guarda_entropia(At,Ent,L,L1),
    calcule_entropia_bin(R,L1,W).
```

O procedimento `calcule_entropia_bin/2` ativa `calcule_entropia_bin/3`. Este procedimento é composto de duas cláusulas. A primeira cláusula é o critério de parada. A segunda realiza recursivamente o cálculo da equação (1). Os três primeiros predicados da segunda cláusula são responsáveis pelo cálculo da entropia do primeiro atributo, na lista corrente. Logo após, este último valor de entropia calculado, é comparado com o maior valor da entropia já encontrado, realizando a substituição se for o caso — `guarda_entropia/4`. Finalmente, `calcule_entropia_bin/3` é chamado recursivamente para repetir este processo sobre os atributos restantes.

Por exemplo, a interrogação `?- calcule_entropia_bin(Lista,W).` ativada com

```
Lista = [[x1,0:1,1:2],[x2,1:2,0:1],[x3,1:1,0:2]]
```

é bem sucedida com

```
W = [x1,1.80617997]
```

Voltando ao procedimento **melhor_at_bin/5**, pg. 10, o último predicado — **divide_bin/4** — é ativado após ter sido encontrado o melhor atributo da lista de exemplos, a fim de particionar os exemplos em uma subárvore esquerda e uma subárvore direita.

Procedimento 3.1.1.7 divide_bin/4

Os argumentos de **divide_bin/4** são:

[+] $< arg_1 >$: lista de exemplos

[+] $< arg_2 >$: o melhor atributo desta lista de exemplos

[−] $< arg_3 >$: subárvore esquerda

[−] $< arg_4 >$: subárvore direita

```
divide_bin(Exs,MelhorAt,SE,SD) :-
    divide_bin(Exs,MelhorAt,[],SE,[],SD).

divide_bin([],_,SE,SE,SD,SD).

divide_bin([Ex|Exs],MelhorAt,SEac,SE,SDac,SD) :-
    retira(MelhorAt:Valor,Ex,ExNovo),
    ifthenelse(Valor = 1,
           divide_bin(Exs,MelhorAt,SEac,SE,[ExNovo|SDac],SD),
           divide_bin(Exs,MelhorAt,[ExNovo|SEac],SE,SDac,SD)).
```

O procedimento `divide_bin/4` ativa o procedimento `divide_bin/6` que contém dois argumentos a mais. Eles são acumuladores necessários para efeito de processamento. O procedimento `divide_bin/6` particiona os exemplos em uma subárvore esquerda e uma subárvore direita de acordo com o valor do melhor atributo ser 0 ou 1 respectivamente. O procedimento `retira/3` é o responsável de retirar o melhor atributo do conjunto de exemplos. Esse novo conjunto de exemplos é colocado na subárvore esquerda ou na subárvore direita caso o valor de melhor atributo seja 0 ou 1 respectivamente, e o procedimento `divide_bin/6` é chamado, recursivamente, para processar o restante dos exemplos em ambas as subárvores.

A listagem completa do programa encontra-se na seção 3.1.3, pg. 16.

3.1.2 Exemplos de Execução

Nesta seção são apresentados vários exemplos de execução do algoritmo implementado `arvore_bin/2` para diversos conjuntos de exemplos. Em cada caso é mostrada a instanciação das variáveis `Atributos` e `Exemplos` consideradas, e o resultado da interrogação correspondente.

Exemplo 3.1.2.1

```
Atributos = [x1,x2,x3,x4]

Exemplos =
  [[x1:1, x2:1, classe:0, x3:0, x4:0],
   [x1:0, x2:1, classe:1, x3:0, x4:0],
   [x1:1, x2:1, classe:0, x3:0, x4:0],
   [x1:1, x2:1, classe:1, x3:1, x4:1],
   [x1:1, x2:0, x3:1, classe:1, x4:1]]

?- arvore_bin(Arvore,Flag).

Arvore = arv(1,x1,arv(0,x3,1)
Flag = exata ->;
no
```

A árvore construída pelo algoritmo é uma árvore exata onde os atributos x2 e x4 são atributos irrelevantes — não comparecem na árvore.

Exemplo 3.1.2.2

```
Atributos = [x1,x2,x3,x4]

Exemplos =
  [[x1:1, x2:1, classe:0, x3:1, x4:0],
   [x1:0, x2:0, classe:1, x3:1, x4:1],
   [x1:0, x2:0, classe:0, x3:0, x4:0],
   [x1:1, x2:1, classe:1, x3:1, x4:1],
   [x1:0, x2:0, x3:1, classe:0, x4:1]]

?- arvore_bin(Arvore,Flag).

Arvore = arv(0,x4,arv(null,x3,arv(arv(1,x2,null),x1,1)))
Flag = nao_exata ->;
no
```

A árvore gerada não é exata, o que indica que em pelo menos um ramo a classe foi aproximada pela classe majoritária, e não há atributos irrelevantes. Além disso, alguns nós folhas foram rotulados com o átomo null, mostrando que não há exemplos no conjunto de treinamento que sejam classificados pelo ramo correspondente.

Exemplo 3.1.2.3

```
Atributos = [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10]

Exemplos =
  [[x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:0,x9:0,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:1,x6:1,x7:1,x8:0,x9:0,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:0,x9:1,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:1,x6:0,x7:0,x8:0,x9:0,x10:0],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:1,x6:0,x7:1,x8:0,x9:0,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:0,x6:1,x7:0,x8:1,x9:1,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:0,x6:0,x7:0,x8:1,x9:1,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:0,x6:0,x7:1,x8:1,x9:0,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:1,x6:0,x7:0,x8:0,x9:0,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:1,x6:1,x7:1,x8:0,x9:1,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:0,x6:0,x7:1,x8:1,x9:1,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:1,x6:1,x7:1,x8:0,x9:1,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:1,x9:0,x10:0],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:1,x6:0,x7:0,x8:1,x9:1,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:1,x6:0,x7:1,x8:0,x9:0,x10:0],
```

```
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:0,x6:1,x7:0,x8:0,x9:1,x10:1],
   [x1:0,x2:1,classe:0,x3:0,x4:0,x5:0,x6:0,x7:0,x8:0,x9:0,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:0,x6:0,x7:1,x8:1,x9:0,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:1,x6:0,x7:0,x8:1,x9:1,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:1,x6:1,x7:1,x8:0,x9:0,x10:0]]

?- arvore_bin(Arvore,Flag).

Arvore = arv(1,x2,arv(arv(arv(arv(0,x9,arv(1,x1,0)),
                    x6,arv(1,x1,0)),x4,null),x3,1))
Flag = exata ->;
no
```

A árvore construída é exata e relativamente grande. Alguns atributos são irrelevantes e possui nós folhas rotuladas com `null`.

Exemplo 3.1.2.4

```
Atributos = [x1,x2,x3,x4,x5,x6,x7,x8,x9,x10]

Exemplos =
  [[x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:0,x9:0,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:0,x6:1,x7:1,x8:0,x9:0,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:0,x9:1,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:0,x6:0,x7:0,x8:0,x9:0,x10:0],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:0,x6:0,x7:1,x8:0,x9:0,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:1,x7:0,x8:1,x9:1,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:0,x6:0,x7:0,x8:1,x9:1,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:1,x9:0,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:0,x6:0,x7:0,x8:0,x9:0,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:0,x6:1,x7:1,x8:0,x9:1,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:1,x9:1,x10:0],
   [x1:0,x2:1,classe:1,x3:0,x4:0,x5:0,x6:1,x7:1,x8:0,x9:1,x10:1],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:1,x9:0,x10:0],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:0,x6:0,x7:0,x8:1,x9:1,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:0,x6:0,x7:1,x8:0,x9:0,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:1,x7:0,x8:0,x9:1,x10:1],
   [x1:0,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:0,x8:0,x9:0,x10:0],
   [x1:1,x2:1,classe:0,x3:0,x4:0,x5:1,x6:0,x7:1,x8:1,x9:0,x10:1],
   [x1:1,x2:1,classe:1,x3:1,x4:1,x5:0,x6:0,x7:0,x8:1,x9:1,x10:1],
   [x1:1,x2:0,x3:1,classe:1,x4:1,x5:0,x6:1,x7:1,x8:0,x9:0,x10:0]]

?- arvore_bin(Arvore,Flag).

Arvore = arv(1,x5,0)
```

```
Flag = exata ->;
no
```

A árvore gerada é exata e pequena, consistindo apenas de um nó de decisão — atributo x5 — que divide os exemplos em duas classes.

3.1.3 Listagem do Programa

Como já mencionado, o procedimento a ser ativado é arvore_bin/2. É responsabilidade do usuário a implementação dos procedimentos atributos/1 e exemplos/1, para pré-processamento dos exemplos, como explicado na seção 3.1 pg. 6. É bom observar que parâmetros extras — tais como o arquivo onde se encontram os exemplos, etc. — podem ser inseridos nestes procedimentos. Segue a listagem do programa onde a instanciação dos procedimentos referem-se a ativação do programa para construir a árvore.

```
% ------- arvore_bin(-,-) -------------------------------
arvore_bin(Arvore,Flag) :-
        atributos(Atributos),
        exemplos(Exemplos),
        forme_arv_bin(Exemplos,Atributos,Arvore,Flag),
        !.

% ------- forme_arv_bin(+,+,-,-) ------------------------
forme_arv_bin(E,A,T,F) :-
        forme_arvore_bin(E,A,T,F),
        ifthen(var(F),F=exata).

% ------- forme_arvore_bin(+,+,-) -----------------------
forme_arvore_bin([],_,null,F).

forme_arvore_bin(Exs,_,Classe,F) :-
        mesma_classe(Exs,Classe),
        !.

forme_arvore_bin(Exs,[],Classe,nao_exata) :-
        maior_classe(Exs,Classe),
        !.

forme_arvore_bin(Exs,Ats,arv(ArvEsq,MelhorAt,ArvDir),F) :-
        melhor_at_bin(Ats,Exs,MelhorAt,NovosAts,[SubEsq,SubDir]),
        forme_arvore_bin(SubEsq,NovosAts,ArvEsq,F),
        forme_arvore_bin(SubDir,NovosAts,ArvDir,F).

% ------- melhor_at_bin(+,+,-,-,-) ----------------------
melhor_at_bin(Ats,Exs,MelhorAt,NovosAts,[SubEsq,SubDir]) :-
        calcule_bin(Ats,Exs,Lista),
        calcule_entropia_bin(Lista,[MelhorAt,_]),
        retira(MelhorAt,Ats,NovosAts),
```

```
        divide_bin(Exs,MelhorAt,SubEsq,SubDir).

% ------- calcule_bin(+,+,-) --------------------------
calcule_bin(Ats,Exs,LCalc) :-
        lista_inicial_bin(Ats,ListaInicial),
        realiza_calculo_bin(Exs,ListaInicial,LCalc).

% ------- lista_inicial_bin(+,-) ----------------------
lista_inicial_bin([],[]).

lista_inicial_bin([Atributo|NLista], [[Atributo,0:0,0:0]|FNLista]) :-
        lista_inicial_bin(NLista,FNLista).

% ------- realiza_calculo_bin(+,+,-) ------------------
realiza_calculo_bin([],L,L).

realiza_calculo_bin([EX|LEX],ListaAtual,ListaFinal) :-
        class(EX,Classe),
        realiza_calculo_bin1(EX,Classe,ListaAtual,NovaLista),
        realiza_calculo_bin(LEX,NovaLista,ListaFinal).

% ------- realiza_calculo_bin1(+,+,+,-) ---------------
realiza_calculo_bin1([],_,L,L).

realiza_calculo_bin1([Atributo:Valor|Exs],ValorClasse,
                                        ListaAtual,NovaLista) :-
        Atributo \= classe, !,
        atualiza_lista_bin(Atributo,ListaAtual,Valor,
                                        ValorClasse,OutraLista),
        realiza_calculo_bin1(Exs,ValorClasse,OutraLista,NovaLista).

realiza_calculo_bin1([Prim|Exs],ValorClasse,ListaAtual,NovaLista) :-
        realiza_calculo_bin1(Exs,ValorClasse,ListaAtual,NovaLista).

% ------- atualiza_lista_bin(+,+,+,+,-) ---------------
atualiza_lista_bin(At,[[At,A,B]|Exs],ValorAtributo,
                                ValorClasse,[[At,C,D]|Exs]) :-
        !,
        atualiza_bin((At,A,B),ValorAtributo,ValorClasse,(At,C,D)).

atualiza_lista_bin(At,[L|Exs],ValorAtributo,ValorClasse,[L|Exs1]) :-
        atualiza_lista_bin(At,Exs,ValorAtributo,ValorClasse,Exs1).

% ------- atualiza_bin(+,+,+,-) -----------------------
atualiza_bin((At,C00:C01,C10:C11),0,0,(At,B00:C01,C10:C11)) :-
        B00 is C00 + 1, !.

atualiza_bin((At,C00:C01,C10:C11),0,1,(At,C00:B01,C10:C11)) :-
        B01 is C01 + 1, !.

atualiza_bin((At,C00:C01,C10:C11),1,0,(At,C00:C01,B10:C11)) :-
        B10 is C10 + 1, !.
```

```
atualiza_bin((At,C00:C01,C10:C11),1,1,(At,C00:C01,C10:B11))  :-
        B11 is C11 + 1, !.

% -------- calcule_entropia_bin(+,-) ---------------------
calcule_entropia_bin(Lista,W) :-
        calcule_entropia_bin(Lista,L,W).

calcule_entropia_bin([],L,L).

calcule_entropia_bin([[At,A:B,C:D]|R],L,W) :-
        entropia([A,B],E),
        entropia([C,D],I),
        Ent is ((A+C)*E) + ((B+D)*I),
        guarda_entropia(At,Ent,L,L1),
        calcule_entropia_bin(R,L1,W).

% -------- guarda_entropia(+,+,-,-) ----------------------
guarda_entropia(At,E,[],[At,E]).

guarda_entropia(At,Ent,[A,E],[A,E]) :-
        Ent =< E,
        !.

guarda_entropia(At,Ent,_,[At,Ent]).

% -------- entropia(+,-) ---------------------------------
entropia(L,E):-
        entropia(L,0,E).

entropia([],Ent,Ent).

entropia([A|R],Temp,E):-
        logaritmo(A,Alog),
        NovoTemp is Alog * A + Temp,
        entropia(R,NovoTemp,E).

% -------- divide_bin(+,+,-,-) ---------------------------
divide_bin(Exs,MelhorAt,SE,SD) :-
        divide_bin(Exs,MelhorAt,[],SE,[],SD).

% -------- divide_bin(+,+,-,-,-,-) -----------------------
divide_bin([],_,SE,SE,SD,SD).

divide_bin([Ex|RExs],MelhorAt,SEac,SE,SDac,SD) :-
        retira(MelhorAt:Valor,Ex,ExNovo),
        ifthenelse(Valor = 1,
                divide_bin(RExs,MelhorAt,SEac,SE,[ExNovo|SDac],SD),
                divide_bin(RExs,MelhorAt,[ExNovo|SEac],SE,SDac,SD)).

% -------- logaritmo(+,-) --------------------------------
logaritmo(A,0):-
```

```
        A =:= 0,
        !.

logaritmo(A,Alog):-
        Alog is log(A).

% ------- mesma_classe(+,-) ----------------------------
mesma_classe([X],C) :-
        !,
        class(X,C).

mesma_classe([X|Y],C) :-
        class(X,C),
        mesma_classe(Y,C).

% ------- class(+,-) ----------------------------------
class([classe:C|_],C) :-
        !.

class([X|Y],C) :-
        class(Y,C).

% ------- maior_classe(+,-) ----------------------------
maior_classe(Exs,Classe) :-
        maior_classe(Exs,[],Classe).

maior_classe([],L1,Classe) :-
        c_maior(L1,Classe),
        !.

maior_classe([Ex|R],L1,Classe) :-
        class(Ex,C),
        insere_estat(C,L1,L2),
        maior_classe(R,L2,Classe).

% ------- c_maior(+,-) ---------------------------------
c_maior([C1:N1|R],Classe) :-
        c_maior(R,N1,C1,Classe).

c_maior([],_,Classe,Classe).

c_maior([Classe1:N1|R],NAnterior,CA,Classe) :-
        N1 >= NAnterior,
        !,
        c_maior(R,N1,Classe1,Classe).

c_maior([_|R],NAnt,CA,Classe) :-
        c_maior(R,NAnt,CA,Classe).

% ------- insere_estat(+,+,-) --------------------------
insere_estat(Classe,[],[Classe:1]) :-
        !.
```

```
insere_estat(C1,[C1:N|_],[C1:N1|_]) :-
        N1 is N + 1.

insere_estat(C1,[X:_|R],[X:_|R1]) :-
        insere_estat(C1,R,R1).

% -------- retira(+,+,-) --------------------------------
retira(X,[X|Y],Y) :-
        !.

retira(X,[Z|Y],[Z|Y1]) :-
        retira(X,Y,Y1).
```

4 Poda em Árvores de Decisão

Na construção de regras de conhecimento a partir de árvores de decisão, é necessário considerar três fases distintas:

1. a criação de uma árvore de decisão a partir de um conjunto de exemplos,
2. a poda da árvore de maneira a remover ramos com pouca representatividade estatística e
3. o processamento da árvore gerada visando a promoção do seu entendimento, isto é, as regras geradas pela árvore.

A criação da árvore de decisão é realizada utilizando um conjunto de dados denominado conjunto de treinamento. Existem aplicações onde os dados para o aprendizado estão imperfeitos, isto é, os dados no conjunto de treinamento podem estar apenas parcialmente especificados, podendo existir a possibilidade de erros nos valores de atributos e nos valores de classes — *ruídos*. Muitos dos ramos de uma árvore induzida através de dados com ruídos, refletem apenas ocorrências acidentais nos dados em questão ao invés de representarem relações existentes entre eles. Consequentemente, tais ramos, dificilmente, serão representativos na classificação de outros exemplos. Ou seja, ruído nos dados influenciam a tarefa de aprendizado tornando-a mais difícil, sendo assim necessário mecanismos especiais, para tentar eliminar esses ruídos. Alguns problemas que são encontrados quando é induzida uma árvore de decisão que contém dados com ruídos são:

- a árvore induzida não é confiável na classificação de novos exemplos e
- árvores induzidas serão, provavelmente, maiores e, consequentemente, mais difíceis de serem compreendidas.

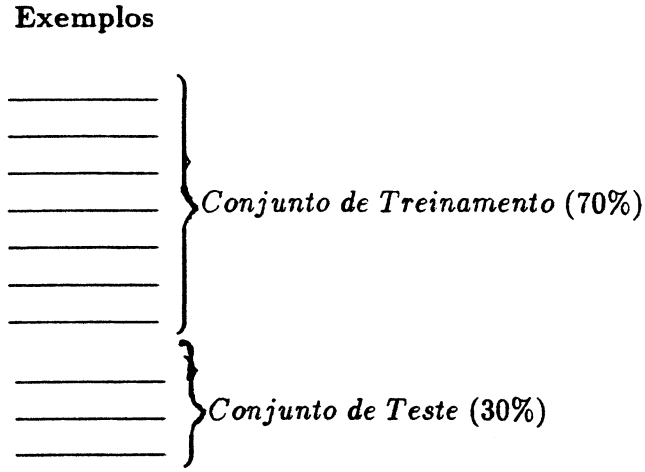
A poda em árvores de decisão, é um mecanismo importante para o aprendizado de dados com ruídos. É importante lembrar que a poda de uma árvore irá aumentar o

número de erros de classificação dos exemplos do conjunto de treinamento, mas em contra partida, deve diminuir o índice de ocorrência de erros na classificação de novos exemplos. Basicamente existem dois tipos de poda:

1. *Pré-poda*, realizada durante o processo de construção da árvore. A ramificação de um nó da árvore pára quando a construção da árvore deixa de ser confiável, e
2. *Pós-poda*, realizada após a construção completa da árvore, quando o erro de classificação de novos exemplos é calculado e, em função desse erro, é decidido podar algumas subárvores que, provavelmente, não são confiáveis.

Em geral, a pós-poda produz resultados mais satisfatórios que a pré-poda, uma vez que, com a pós-poda, tanto a precisão quanto a complexidade das árvores tendem a melhorar [Mingers 89b]. Para aplicar alguns dos mecanismos de pós-poda existentes, é necessário dividir o conjunto de exemplos disponível em dois subconjuntos.

1. *conjunto de treinamento* para construir a árvore e
2. *conjunto de teste* para podar a árvore. Na prática esta divisão deve ser feita de forma aleatória na proporção 70/30 [Mingers 89b] a fim de obter resultados confiáveis.



Diversos métodos de pós-poda, bem como resultados empíricos da aplicação desses métodos, são encontrados na literatura, onde é mostrado que não há uma diferença marcante nos resultados obtidos pelos diversos métodos de pós-poda propostos. Neste trabalho é descrito e implementado um desses métodos, denominado Método de Poda de Redução do Erro [Quinlan 87].

4.1 Método de Poda de Redução do Erro

O Método de Poda de Redução do Erro — MPRE —, faz uso do conjunto de treinamento e conjunto de teste. Este método produz uma série de árvores podadas, usando

o conjunto de teste diretamente, ao invés de usá-lo apenas na seleção da melhor árvore. A árvore de decisão obtida é aquela que possui o menor número de exemplos não classificados para o conjunto de teste. O MPRE necessita da árvore de decisão construída utilizando o conjunto de treinamento, bem como do conjunto de testes, para produzir a árvore podada. Primeiramente, o conjunto de testes é introduzido na árvore de decisão original contando, em cada nó da árvore, o número de exemplos em cada classe. Esta árvore é denominada árvore estatística — Figura 3, pg. 23. A seguir, para toda subárvore não folha S de T, são contados

1. o número de exemplos dos nós folhas não classificados (d_s),
2. o número de exemplos que não seriam classificados, se a subárvore S em questão fosse substituída por sua melhor folha (d_{ps}).

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

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

Assim, o MPRE gera um conjunto de árvores e termina com a obtenção da árvore podada que melhor classifica, segundo este critério, o conjunto de testes. Pode ser que aconteça a situação de existirem várias subárvores com o mesmo valor de g. Quinlan não especifica qual subárvore escolher nesta situação. Entretanto, resultados experimentais [Mingers 89b] mostram que a escolha entre essas subárvores interfere muito pouco na precisão de classificação; escolhe-se, portanto, a maior, na certeza que essa escolha irá reduzir o número de iterações necessárias para podar completamente a árvore.

Este método possui duas desvantagens: a necessidade de um conjunto de teste separado do conjunto de treinamento, e a poda indesejável de partes da árvore original que poderiam corresponder a casos especiais mais raros, eventualmente não presentes no conjunto de teste.

Como ilustração do MPRE, considere uma subárvore de uma árvore estatística binária de classificação — Figura 3, pg. 23.

Considerando o nó I, já que o algoritmo trata a poda a partir dos nós folhas até chegar à raiz, a Figura 3 mostra que este nó classifica um total de 19 exemplos, sendo 15 da classe 0 e 4 da classe 1. Se a subárvore for podada, o nó I se torna uma folha rotulada com o valor da classe com o maior número de exemplos classificados — neste caso 0 — e, consequentemente, 4 dos 19 exemplos estarão sendo classificados erradamente (d_{ps}).

Somando os erros atuais de classificação dos nós folhas da subárvore, tem-se:

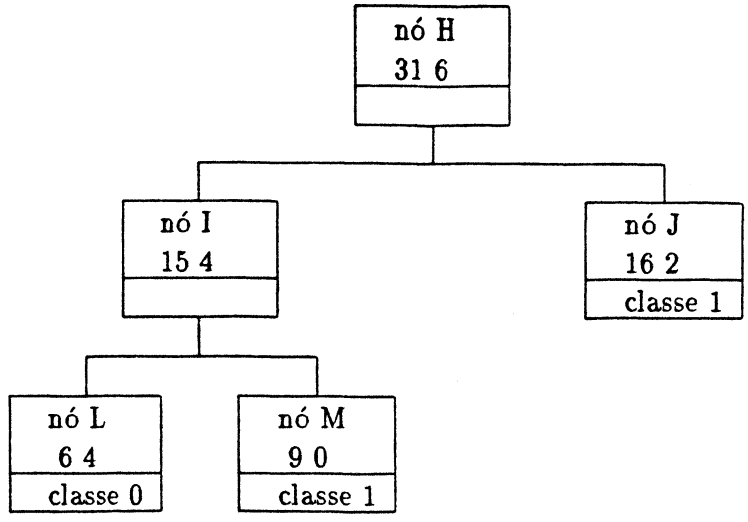


Figura 3: Subárvore de uma Árvore Estatística de Classificação

- para o nó L o erro de classificação é 4
- para o nó M o erro de classificação é 9
- o total de erros de classificação dos nós folhas é $ds = 13$.

Assim, o erro de classificação dos nós folhas é maior que o erro de classificação do nó I se ele for podado

$$g_s = d_s - d_{ps} = 13 - 4 = 9 \geq 0.$$

Com isso, decide-se podar o nó I. Isto é feito recursivamente com todos os nós da árvore de decisão até não ser possível continuar a poda, isto é, incrementa o erro de classificação.

4.2 Descrição da Implementação

O algoritmo implementado para realizar a poda de uma árvore de decisão binária, utilizando o Método de Poda de Redução do Erro, faz uso da árvore de decisão criada, a partir do conjunto de treinamento, pelo algoritmo descrito na seção 3.1.1, pg. 7. Como já mencionado, a estrutura dessa árvore de decisão é:

arv(SubarvEsq,MelhorAtributo,SubarvDir)

com base nessa árvore e utilizando os exemplos do conjunto de teste, a árvore estatística é criada. A estrutura que representa a árvore estatística tem a seguinte forma:

arv(SubarvEsq/L1,MelhorAtributo/L,SubarvDir/L2)

onde *L*, *L1* e *L2* são listas de dois 2 elementos (números inteiros maiores ou iguais a zero), cuja soma representa o número total de exemplos nesse nó da árvore. O primeiro e o segundo elemento da lista representam, respectivamente, o número de exemplos com valor da classe = 0 e classe = 1. A poda é realizada sobre esta árvore estatística e, quando o processo de poda termina, a árvore podada é novamente expressa com a mesma estrutura da árvore inicial.

Os principais procedimentos utilizados no processo de poda da árvore, são discutidos a seguir.

4.2.1 Procedimentos: Definição e Discussão

O procedimento no nível mais alto para realizar a poda da árvore é o seguinte:

Procedimento 4.2.1.1 podar/3

```
podar(Arvore,Exs,ArvorePodada) :-
    estat(Arvore,Exs,ArvEstat),
    poda_arv(ArvEstat,ArvPod),
    muda_notacao(ArvPod,ArvorePodada), !.
```

Os argumentos de podar/3 são:

[+] < arg_1 >: árvore de decisão criada pelo **Procedimento 3.1.1.1** pg. 7, utilizando o conjunto de exemplos de treinamento

[+] < arg_2 >: conjunto de exemplos de teste

[−] < arg_3 >: árvore podada

O procedimento **estat/2** é responsável pela construção da árvore estatística. Por exemplo, considere **Arvore** e **Exs** instanciadas da seguinte forma:

```
Arvore = arv(1,x1,arv(0,x3,1))

Exs = [[x1:1, x2:1, classe:0, x3:1, x4:0],
       [x1:0, x2:0, classe:1, x3:1, x4:0],
       [x1:0, x2:0, classe:0, x3:0, x4:1],
       [x1:1, x2:1, classe:1, x3:1, x4:1],
       [x1:0, x2:0, x3:1, classe:0, x4:1]].
```

a interrogação `?- estat(Arvore,Exs,ArvEstat)` será bem sucedida com

```
ArvEstat = arv(1/[2,1],x1/[3,2],arv(0/[0,0],x3/[1,1],1/[1,1]))
```

O segundo procedimento — **poda_arv/2** — é definido a seguir.

Procedimento 4.2.1.2 poda_arv/2

```
poda_arv(ArvE,ArvPodada) :-
    poda(ArvE,ArvP),
    ArvE \= ArvP,
    poda_arv(ArvP,ArvPodada), !.

poda_arv(Arv,Arv).
```

Os argumentos de poda_arv/2 são:

[+] $< arg_1 >$: árvore estatística

[−] $< arg_2 >$: árvore podada

O procedimento poda_arv/2 é responsável por ativar o procedimento que realmente realiza a poda da árvore — poda/2. Quando o procedimento poda/2 for bem sucedido, independente se a árvore for podada ou não, é feito uma comparação entre a árvore anterior e a árvore posterior à poda. Se as duas árvores são árvores diferentes, significa que houve poda, então é necessário ativar o procedimento, recursivamente, poda_arv/2 a fim de verificar se uma nova poda é necessária. Se este não for o caso — as duas árvores são iguais —, a árvore permanece a mesma e o procedimento finaliza.

Procedimento 4.2.1.3 poda/2

```
poda(Classe/L,Classe/L)
    atomic(Classe), !.

poda(arv(Se/L1,R/L,Se/L2),Se/L) :-
    sao_folhas(Se/L1),
    sao_folhas(Se/L2), !.

poda(arv(Se,R/L,Sd),C/L) :-
    sao_folhas(Se),
    sao_folhas(Sd),
    candidata_folha(L,C,S),
    soma_erros_class(Se,Sd,S1),
    S1 >= S, !.

poda(arv(Se,R/L,Sd),arv(Se,R/L,Sd)) :-
    sao_folhas(Se),
    sao_folhas(Sd), !.

poda(arv(Se,R/L,Sd),arv(Se1,R/L,Sd1)) :-
    poda(Se,Se1),
    poda(Sd,Sd1).
```

O procedimento poda/2 tem cinco cláusulas, uma cláusula para cada uma das cinco situações que poderão ocorrer durante o processo de poda da árvore de decisão. A primeira cláusula se refere à situação onde o nó considerado é um nó folha rotulado com o valor da classe. Neste caso, não há poda e o nó permanece igual.

A segunda cláusula é ativada quando a subárvore esquerda e a subárvore direita de um nó são folhas e contém o mesmo valor da classe. É importante notar que isto somente acontece se a árvore já foi podada, pelo menos uma vez, e a classe majoritária de um nó for igual ao valor da classe do outro nó. Neste caso, a subárvore é podada e o nó é rotulado com o valor da classe em questão.

A terceira cláusula verifica, também, se suas subárvores são nós folhas. Se isto ocorre é ativado o procedimento `candidata_folha/3` que é bem sucedido com o valor da classe que classifica os exemplos em maior número e o número de exemplos que não seriam classificados pelo nó, caso fosse podada a subárvore rotulando o nó folha com o valor da classe. Por exemplo, se `candidata_folha/3` é ativada com `L = [2,1]`, será bem sucedida com `C = 0` (o número 2 da lista `L` significa que o nó classifica 2 exemplos cujo valor da classe é 0) e `S = 1`.

O procedimento `soma_erros_class/3` faz a soma dos erros de classificação de cada nó folha. Isto é, se o nó folha é rotulado com valor 1 (classe = 1), por exemplo, a soma dos erros de classificação será igual ao número de exemplos, pertencentes àquele nó, cujo valor da classe é 0. Com os dois valores instanciados `S` e `S1`, é feito uma comparação entre eles. Se o erro de classificação dos nós — `S1` — é maior ou igual ao erro de classificação se a subárvore for substituída por sua classe majoritária — `S` —, então a subárvore é podada.

A quarta cláusula é ativada se as subárvores do nó são folhas e caso a terceira cláusula falhe quando o erro de classificação dos nós não for maior ou igual ao erro do nó da classe majoritária. Neste caso a subárvore não é podada.

Como o algoritmo somente trata da poda em nós folhas ou quando as subárvores de um nó são folhas, a quinta cláusula faz a recursão até chegar nessas situações.

Voltando ao procedimento podar/3, seção 4.2.1.1 pg. 24, o último procedimento `muda_notacao/2`, simplesmente transforma a estrutura da árvore estatística podada para a estrutura de árvore binária padrão.

4.2.2 Exemplos de Execução

Nesta seção, são apresentados vários exemplos de execução do algoritmo implementado — `podar/3` — para diversas árvores de decisão. Em cada caso é mostrada a instanciação das variáveis **Arvore** e **Exs** consideradas e o resultado da interrogação.

Exemplo 4.2.2.1

```
Arvore = arv(1,x1,arv(0,x3,1))
```

```
Exs = [[x1:1, x2:1, classe:0, x3:1, x4:0],
       [x1:0, x2:0, classe:1, x3:1, x4:0],
       [x1:0, x2:0, classe:0, x3:0, x4:1],
       [x1:1, x2:1, classe:1, x3:1, x4:1],
       [x1:0, x2:0, x3:1, classe:0, x4:1]]

?- podar(Arvore,Exs,ArvorePodada).

ArvorePodada = 0 ->;
no
```

A árvore podada é rotulada com um simples nó com valor da classe 0.

Exemplo 4.2.2.2

```
Arvore =
  arv(0,x3,arv(arv(arv(1,x4,0),x2,null),x1,arv(null,x2,arv(0,x4,1))))

Exs = [[x1:1, x2:1, classe:0, x3:0, x4:0],
       [x1:0, x2:1, classe:1, x3:0, x4:0],
       [x1:1, x2:1, classe:0, x3:0, x4:0],
       [x1:1, x2:1, classe:1, x3:1, x4:1],
       [x1:1, x2:0, x3:1, classe:1, x4:1]]

?- podar(Arvore,Exs,ArvorePodada).

ArvorePodada = arv(0,x3,1) ->;
no
```

A árvore é podada até chegar a um simples nó de decisão — atributo **x3** — que divide os exemplos em duas classes.

Exemplo 4.2.2.3

```
Arvore =
  arv(0,x3,arv(arv(arv(1,x4,0),x2,null),x1,arv(null,x2,arv(0,x4,1))))

Exs = [[x1:1, x2:1, classe:0, x3:1, x4:0],
       [x1:0, x2:0, classe:1, x3:1, x4:0],
       [x1:0, x2:0, classe:0, x3:0, x4:1],
       [x1:1, x2:1, classe:1, x3:1, x4:1],
       [x1:1, x2:0, x3:1, classe:0, x4:1]]

?- podar(Arvore,Exs,ArvorePodada).

ArvorePodada = arv(0,x3,arv(1,x1,arv(null,x2,arv(0,x4,1)))) ->;
no
```

A árvore é podada mas mantém algumas folhas rotuladas null.

4.2.3 Listagem do Programa

O procedimento a ser ativado é podar/3, sendo responsabilidade do usuário fornecer o conjunto de exemplos de teste — Exs — diferente do conjunto de exemplos de treinamento utilizado na construção da árvore de decisão. Segue a listagem do programa.

```
% -------- podar(+,+,-) ---------------------------------
podar(Arvore,Exs,ArvorePodada) :-
        estat(Arvore,Exs,ArvEstat),
        poda_arv(ArvEstat,ArvPod),
        muda_notacao(ArvPod,ArvorePodada), !.

% -------- poda_arv(+,-) --------------------------------
poda_arv(ArvE,ArvPodada) :-
        poda(ArvE,ArvP),
        ArvE \= ArvP,
        poda_arv(ArvP,ArvPodada),
        !.

poda_arv(Arv,Arv).

% -------- estat(+,+,-) ---------------------------------
estat(Arv,Exs,ArvE) :-
        expande(Arv,ArvM),
        verifica(ArvM,Exs,ArvE),
        !.

% -------- expande(+,-) ---------------------------------
expande(A,A/[0,0]) :-
        atomic(A).

expande(arv(Se,Ma,Sd),arv(Se1,Ma/[0,0],Sd1)) :-
        expande(Se,Se1),
        expande(Sd,Sd1).

% -------- verifica(+,+,-) ------------------------------
verifica(Arv,[],Arv).

verifica(Classe/L,Exs,Classe/L1) :-
        atomic(Classe),
        !,
        faz_contagem(Classe/L,Exs,Classe/L1).

verifica(arv(Se,R/L,Sd),Exs,arv(Se1,R/L1,Sd1)) :-
        faz_contagem(R/L,Exs,R/L1),
        divide(Exs,R,Esq,Dir),
        verifica(Se,Esq,Se1),
        verifica(Sd,Dir,Sd1).
```

```
% -------- faz_contagem(+,+,-) ---------------------------
faz_contagem(R/L,[],R/L).

faz_contagem(R/L,[Ex|Exs],R/L1) :-
        conta_um_ex(R/L,Ex,R/W),
        faz_contagem(R/W,Exs,R/L1).

% -------- conta_um_ex(+,+,-) ----------------------------
conta_um_ex(Classe/[A,B],Ex,Classe/L1) :-
        atomic(Classe),
        !,
        class(Ex,C),
        ifthenelse(C=1,(B1 is B+1,L1=[A,B1]),(A1 is A+1,L1=[A1,B])).

conta_um_ex(R/L,Ex,R/W) :-
        class(Ex,C),
        conta(R/L,C,R/W).

% -------- conta(+,+,-) ----------------------------------
conta(R/[A0,A1],0,R/[AX,A1]) :-
        AX is A0 + 1, !.

conta(R/[A0,A1],1,R/[A0,AX]) :-
        AX is A1 + 1.

% -------- poda(+,-) -------------------------------------
poda(Classe/L,Classe/L) :-
        atomic(Classe), !.

poda(arv(Se/L1,R/L,Se/L2),Se/L) :-
        sao_folhas(Se/L1),
        sao_folhas(Se/L2), !.

poda(arv(Se,R/L,Sd),C/L) :-
        sao_folhas(Se),
        sao_folhas(Sd),
        candidata_folha(L,C,S),
        soma_erros_class(Se,Sd,S1),
        S1 >= S, !.

poda(arv(Se,R/L,Sd),arv(Se,R/L,Sd)) :-
        sao_folhas(Se),
        sao_folhas(Sd), !.

poda(arv(Se,R/L,Sd),arv(Se1,R/L,Sd1)) :-
        poda(Se,Se1),
        poda(Sd,Sd1).

% -------- sao_folhas(+) ---------------------------------
sao_folhas([]).

sao_folhas(Cl/L).
```

```
% ------- candidata_folha(+,-,-) ------------------------
candidata_folha([A,B],0,B) :-
        A >= B, !.

candidata_folha([A,B],1,A).

% ------- soma_erros_class(+,+,-) -----------------------
soma_erros_class(Se,Sd,S) :-
        erro(Se,S1),
        erro(Sd,S2),
        S is S1 + S2.

% ------- erro(+,-) -------------------------------------
erro(Cl/L,S) :-
        qual_classe(Cl/L,S).

erro(arv(Se,Ma/L,Sd),S) :-
        erro(Se,S1),
        erro(Sd,S2),
        S is S1 + S2.

% ------- qual_classe(+,-) ------------------------------
qual_classe(0/[A,B],B) :- !.

qual_classe(1/[A,B],A) :- !.

qual_classe(null/[A,B],C) :-
        C is A+B.

% ------- muda_notacao(+,-) -----------------------------
muda_notacao(Classe/L,Classe) :-
        atomic(Classe),!.

muda_notacao(arv(Se,R/L,Sd),arv(Se1,R,Sd1)) :-
        muda_notacao(Se,Se1),
        muda_notacao(Sd,Sd1).

% ------- divide(+,+,-,-) -------------------------------
divide(Exs,Ma,Se,Sd) :-
        divide(Exs,Ma,[],Se,[],Sd).

divide([],_,Se,Se,Sd,Sd).

% ------- divide(+,+,+,-,+,-) ---------------------------
divide([Ex|Exs],Ma,Seac,Se,Sdac,Sd) :-
        retira_exemplo(Ma:V,Ex,ExNovo),
        ifthenelse(V=1,
                divide(Exs,Ma,Seac,Se,[ExNovo|Sdac],Sd),
                divide(Exs,Ma,[ExNovo|Seac],Se,Sdac,Sd)).

% ------- retira_exemplo(+,+,-) -------------------------
```

```
retira_exemplo(Ma,[Ma|Exs],Exs).

retira_exemplo(Ma,[Ma1|Exs],[Ma1|Exs1]) :-
        retira_exemplo(Ma,Exs,Exs1).
```

5 Procedimentos Utilitários

Nesta seção são apresentados alguns procedimentos que são úteis para analisar a complexidade da árvore de decisão binária gerada pelos algoritmos implementados. Por exemplo, a altura da árvore de decisão corresponde ao máximo comprimento das regras de conhecimento representadas pela árvore de decisão correspondente. O número de nós terminais dá o número de regras de conhecimento geradas, enquanto que a relação entre o número de nós e o número de nós terminais, dá o comprimento médio das regras de conhecimento correspondentes a essa árvore de decisão. A seguir são descritos cada um destes procedimentos.

Procedimento 5.1 altura/2

```
altura(A,0) :-
    atomic(A), !.

altura(arv(Se,Raiz,Sd),X) :-
    altura(Se,Y),
    altura(Sd,Z),
    max(Y,Z,Max),
    X is Max + 1.

max(X,Y,X) :-
    X >= Y, !.

max(X,Y,Y).
```

A altura de uma árvore é o nível máximo entre os nós da árvore. Isto é, a altura de uma árvore binária é:

1. um, se a árvore possuir apenas a raiz, caso contrário
2. é o máximo entre as alturas de suas subárvores (esquerda e direita).

Por exemplo, considerando a variável **Arvore** instanciada com

```
Arvore = (1,x1,arv(0,x3,1))
```

tem-se a seguinte resposta para a interrogação

```
?- altura(Arvore,X).

X = 3 ->;
no
```

Procedimento 5.2 nos/2

```
nos(A,1) :-
    atomic(A), !.

nos(arv(Se,Raiz,Sd),X) :-
    nos(Se,Y),
    nos(Sd,Z),
    X is Y + Z + 1.
```

O procedimento **nos/2** é bem sucedido com a variável **X** instanciada com o número total de nós de uma árvore binária (nós de decisão e nós folhas).

Segue um exemplo de interrogação com **Arvore** instanciada como anteriormente.

```
?- nos(Arvore,X).

X = 5 ->;
no
```

Procedimento 5.3 folhas/2

```
folhas(A,1) :-
    atomic(A), !.

folhas(arv(Se,Raiz,Sd),X) :-
    folhas(Se,Y),
    folhas(Sd,Z),
    X is Y + Z.
```

O procedimento **folhas/2** é bem sucedido com a variável **X** instanciada com o número de folhas de uma árvore binária. Isto é todos os nós terminais pertencentes à árvore.

Segue um exemplo de interrogação com **Arvore** instanciada como anteriormente.

```
?- folhas(Arvore,X).

X = 3 ->;
no
```

6 Conclusões

Neste trabalho foi apresentado e discutido implementações Prolog de diversos programas que tem como objetivo:

1. a construção, a partir de um conjunto de treinamento, de árvores de decisão binárias
2. a poda, a partir de um conjunto de teste, da árvore binária criada no passo anterior e
3. procedimentos utilitários para analisar a complexidade da árvore de decisão construída.

Para construir a árvore de decisão é utilizado a metodologia da Família TDIDT no modo não incremental, onde é necessário que os exemplos do conjunto de treinamento estejam disponíveis simultaneamente na memória, para que seja induzido um conceito. No processo de construção da árvore foram considerados apenas atributos que assumem dois valores possíveis, isto é, o atributo está presente ou ausente no conjunto de treinamento. Caso isso não ocorra, é necessário um pré-processamento dos dados de entrada — conjunto de treinamento — antes que a árvore seja construída.

O mecanismo de pós-poda da árvore binária implementado é baseado no Método de Poda de Redução do Erro. Este método trabalha buscando o erro de classificação dos nós folhas de uma subárvore qualquer, decidindo podar ou não a subárvore, dependendo do erro de classificação do conjunto de teste.

Alguns procedimentos utilitários são considerados. Estes procedimentos tais como altura, número de nós e número de folhas da árvore de decisão, são úteis para fazer a análise da complexidade da árvore gerada pelos algoritmos implementados.

Deve ser ressaltado que, a estrutura de árvores de decisão binárias é importante no processo de construção automática de *features*, que será tratado em um trabalho futuro.

Referências

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

[Castineira 91] Castineira, M.I.; Monard, M.C.; Nicoletti, M.C. *Aprendizado de Máquina: Descrição e Implementação de um Algoritmo Geral para a Construção de Árvores de Decisão.* Notas do ICMSC-USP, Nº 98, 42 pg., 1991.

[Cestnik 87] Cestnik, B.; Kononenko, I.; Bratko, I. *Assistant 86: A Knowledge-Elicitation Tool for Sophisticated Users.* Progress in Machine Learning, I. Bratko & N. Lavrac (Ed.), Sigma Press, pp. 31-45, 1987.

[Mingers 89a] Mingers, J. *An Empirical Comparison of Selection Measures for Decision Tree Induction.* Machine Learning 3, pp. 319-42, 1989.

[Mingers 89b] Mingers, J. *An Empirical Comparison of Pruning Methods for Decision Tree Induction.* Machine Learning 4, pp. 227-43, 1989.

[Nicoletti 92a] Nicoletti, M.C.; Castineira, M.I.; Monard, M.C. *Descrição e Implementação dos Mecanismos de Janela e Poda em Árvores de Decisão no Aprendizado Indutivo de Máquina.* Notas do ICMSC-USP, Nº 110, 37 pg., Maio 1992.

[Nicoletti 92b] Nicoletti, M.C.; Bezerra, P.C.; Monard, M.C. *Learning Boolean Functions Using Prime-implicants as Features.* Proceedings International AMSE Conference on "Signals, Data, Systems", Chicago, USA, September 2-4, AMSE Press, Vol. 1, pp. 177-83, 1992.

[Nicoletti 93a] Nicoletti, M.C.; Monard, M.C. *Empirical Evaluation of Two Pruning Methods Applied to Constructive Learning of Boolean Functions.* Anales del Primer Congreso Internacional de Informatica, Computacion y Teleinformatica, INFORMATICA 93, Mendoza, Argentina, pp. 265-77, Junho 1993.

[Nicoletti 93b] Nicoletti, M.C.; Monard, M.C. *The Role of Description Languagens in Inductive Concept Learning.* Anais I SBAI, Rio Claro, SP, pp. 403-12, Setembro 1993.

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

[Quinlan 86] Quinlan, J.R.; Compton, P.J.; Horn, K.A.; Lazarus, L. *Inductive Knowledge Acquisition: A Case Study.* Proceedings of the 2nd Australian Conference on Applications of E.S., Sydney, pp. 183-203, 1986.

[Quinlan 87] Quinlan, J.R. *Simplifying Decision Trees.* Int. J. Man-Machine Studies 27, pp. 221-34, 1987.