ISSN - 0103-2577

Aprendizado de máquina: descrição e implementação de um algoritmo geral para a construção de árvores de decisão

MARIA INÉS CASTIÑEIRA
MARIA CAROLINA MONARD
MARIA DO CARMO NICOLETTI

Nº 98

N O T A S D O I C M S C

São Carlos (SP)
Out./91

Aprendizado de Máquina: Descrição e Implementação de um Algoritmo Geral para a Construção de Árvores de Decisão

Maria Inés Castiñeira[1]
Maria Carolina Monard
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

Sumário

Aprendizado de Máquina é uma importante área de pesquisa em Inteligência Artificial uma vez que lida com a capacidade de aprender, essencial para um comportamento ser considerado inteligente. Um dos objetivos da pesquisa em Aprendizado de Máquina é o de auxiliar o processo de aquisição de conhecimento, atividade considerada como o gargalo na construção de Sistemas Baseados em Conhecimento.

Uma das formas de aprendizagem é por generalizações, isto é, através do uso de processos indutivos. São várias as estratégias desenvolvidas que viabilizam Aprendizado de Máquina por Indução. Uma delas está baseada na construção de árvores de decisão. Esta estratégia é adotada por uma família de sistemas de aprendizado por indução conhecida como família TDIDT — Top Down Induction Decision Trees.

Este trabalho descreve a implementação de um sistema que constrói árvores de decisão a partir de um conjunto de exemplos, utilizando o algoritmo geral da família TDIDT. Tal implementação utiliza técnicas avançadas de Programação Prolog, bem como uma estrutura de dados apropriada que promove a eficiência do sistema proposto.

[1]Trabalho realizado com auxílio do RHAE/ILTC

Conteúdo

1 Introdução

Sistemas Baseados em Conhecimento — SBC — são programas que procuram reproduzir a habilidade humana de resolver problemas usando raciocínio. A Base de Conhecimento é o componente do SBC que contém o modelo de resolução de problemas em um determinado domínio de aplicação.

Uma das tarefas mais difíceis na implementação de um SBC é a construção da Base de Conhecimento. Nesta tarefa, a maior parte do tempo é gasta conferindo com a fonte de conhecimento — geralmente um especialista humano no assunto —, os processos de tomada de decisão, bem como verificando se as regras de conhecimento criadas simulam satisfatoriamente o conhecimento do domínio. Esta atividade é denominada aquisição de conhecimento.

A habilidade de um especialista na resolução de problemas em seu domínio de conhecimento não está necessariamente ligada com a sua habilidade em expressar e articular seu conhecimento. Usualmente, o conhecimento necessário para um SBC é extraído do especialista através de um lento processo de entrevistas realizadas pelo Engenheiro de Conhecimento.

O problema da técnica que usa entrevistas é que nelas o especialista tende a expressar suas conclusões em termos gerais, muitas vezes amplos demais para serem analisados pela máquina. Os elementos básicos que sedimentam seu conhecimento são ativados e combinados tão rapidamente que é difícil para ele descrever este processo.

Uma conseqüência direta disso é que a produtividade do método de entrevistas é baixa, o que torna a aquisição de conhecimento um processo demorado; devido a isto, a aquisição do conhecimento é considerada como o gargalo na construção de Sistemas Baseados em Conhecimento.

Uma das estratégias utilizadas para acelerar o processo da aquisição, amplamente defendida por Donald Michie [Michie 90] e outros, utiliza métodos indutivos para extrair regras gerais a partir de exemplos concretos. O especialista fornece um conjunto tutorial de exemplos do domínio de aplicação, junto com as classes às quais eles pertencem; o sistema gera regras a partir desses exemplos utilizando métodos indutivos. Tais regras usualmente são validadas e refinadas por especialistas humanos.

Entre os diversos métodos indutivos, alguns se caracterizam por representar as regras inferidas na forma de árvores de decisão. Quinlan denominou os sistemas que apresentam estas características como pertencendo à família TDIDT — Top Down Induction Decision Trees — pois estes sistemas constroem árvores de decisão indutivas em forma "top-down", isto é, começam na raiz e descem até as folhas [Quinlan 86a].

O objetivo deste trabalho é o de apresentar e discutir os principais detalhes da implementação — realizada na linguagem de programação Prolog [Arity 88] — de um sistema para construção de árvores indutivas de decisão, e não os princípios do processo da

construção deste tipo de árvores. O leitor não familiarizado com este processo poderá consultar [Arariboia 89], [Castiñeira 90a], [Quinlan 86a].

O trabalho está organizado da seguinte forma: na seção 2 é descrito brevemente o processo de construção de árvores indutivas de decisão. A seção 3 contém considerações gerais sobre a implementação realizada. A seção 4 descreve efetivamente os detalhes da implementação e a seção 5 contém as conclusões e sugestões de trabalhos futuros.

2 Construção de Árvores Indutivas de Decisão

Os sistemas da família TDIDT não são limitados a uma área de aplicação específica, tal como química, xadrez ou medicina, isto é, são sistemas de propósito geral. O objetivo de todos eles é o de classificar objetos; eles produzem regras ou descrições de um determinado número de classes de objetos. Quando novos objetos são observados, estas regras devem predizer a que classe estes objetos pertencem. As seguintes tarefas, por exemplo, são tarefas de classificação:

- diagnóstico de um problema de saúde a partir dos sintomas do paciente; as classes poderiam ser as diversas doenças ou as possíveis terapias;
- decidir, a partir de observações climáticas e condições dos solos, se uma determinada região geográfica é apta para um dado plantio ou não. Aqui as classes são *apta, não apta*;
- decidir, segundo diversas condições particulares e do mercado financeiro, qual o melhor investimento a realizar. As classes poderiam ser: ouro, dólar, over, poupança, etc., e as regras geradas seriam do tipo:

Se (condições)
então aplique_em (Classe_ Ki)

Os sistemas que fazem parte da família TDIDT se caracterizam por representar o conhecimento na forma de árvores de decisão, usadas para tarefas de classificação. Esta forma de representação, relativamente simples, não tem o poder expressivo de redes semânticas ou outras representações de primeira ordem. Apesar da simplicidade, consegue-se expressar problemas complexos usando árvores de decisão. Como consequência da simplicidade da representação do conhecimento utilizada, estes algoritmos são significativamente menos complexos que aqueles que fazem uso de uma representação de conhecimento mais elaborada.

Formalmente, pode-se definir uma árvore de decisão como sendo [Utgoff 89]:

1. um nó folha – ou nó resposta –, que contém um nome de classe, ou

2. um nó intermediário - ou nó de decisão –, que contém um teste de atributo que, para cada um dos possíveis valores de atributo, tem um ramo para uma outra árvore de decisão.

Os exemplos classificados, a partir dos quais é criada a árvore de decisão, são conhecidos somente através de um conjunto de atributos e de seus valores; geralmente costuma-se considerar a classe do exemplo como um atributo.

2.1 Um Exemplo de Árvore de Decisão

Para classificar pacientes com hipotireoidismo, por exemplo, o atributo de classe pode assumir os valores

$$\{\textit{hipotireoidismo primário, hipotireoidismo secundário, hipotireoidismo compensado, negativo}\}$$

Alguns dos atributos utilizados neste caso podem ser contínuos, tais como medições de *TSH* e *FTI*, outros podem ser discretos, como *sexo* com valores $\{f, m\}$, ou *cirurgia tireoides* com valores $\{f, v\}$. A Figura 1 mostra uma árvore para classificação de hipotireoidismo, gerada utilizando este paradigma [Quinlan 87].

```
TSH < 6.05 : negativo
TSH > 6.05 :
        FTI < 64.5 : hipotireoidismo primario
        FTI > 64.5 :
                tiroxina = v : negativo
                tiroxina = f :
                        cirurgia tireoides = v : negativo
                        cirurgia tireoides = f :
                                TT4 < 150.5 : hipotireoidismo compensado
                                TT4 > 150.5 : negativo
```

Figura 1: Árvore de Classificação de Hipotireoidismo

Cada um dos ramos desta árvore representa uma regra. Algumas delas são descritas a seguir:

1 Se TSH < 6.05
então hipotireoidismo negativo.
2 Se TSH > 6.05 **e**
FTI < 64.5
então hipotireoidismo primário.
...
6 Se TSH > 6.05 e
FTI > 64.5 e
tiroxina = falso e
cirurgia tireoides = falso e
TT4 > 105.5
então negativo.

2.2 Algoritmo Geral dos Sistemas da Família TDIDT

Para ramificar cada nó da árvore de decisão 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 classifique 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 89].

No exemplo da Figura 1, pg. 3 a função de avaliação utilizada indicou o atributo *TSH* como melhor atributo para começar a ramificar os exemplos. Desta forma, todas as observações de hipotireoidismo foram divididas em dois grupos, dependendo do valor do TSH ser menor ou maior que 6.05, isto é, eles foram ramificados em dois nós.

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.

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:

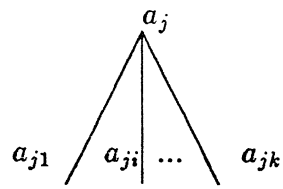
Dados um conjunto de exemplos de aprendizado E;
uma condição de parada $t(E)$;
uma função de avaliação $aval(E, atributo)$

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_{j1}, a_{j2}, ... a_{jk} 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 86b] 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 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.

2.3 Entropia

Entropia é uma grandeza que mede desordem, tanto de objetos físicos quanto de informações. Quanto maior a entropia maior a desordem. A introdução desta grandeza

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

para avaliar atributos no processo de construção de árvores de decisão foi realizada por Quinlan no sistema ID3 [Quinlan 79].

Em [Arariboia 89] o conceito de entropia é explicado de uma forma muito clara:

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

Para calcular a entropia de um determinado atributo é necessário calcular a entropia de uma ramificação. Por exemplo, para calcular a entropia do atributo *TSH* da Figura 1, pg. 3, deve-se calcular a entropia gerada pelos exemplos contidos no ramo $TSH > 6.05$ e do ramo $TSH < 6.05$. Se um ramo contém observações de uma única classe pode-se dizer que ele está ordenado e possui entropia zero; sua desorganização é nula. A entropia do ramo $TSH < 6.05$ é zero pois ele tem somente exemplos da classe negativa; já o outro ramo tem exemplos de diferentes classes e portanto tem entropia maior que zero. Segue-se o cálculo da entropia de um determinado ramo ainda desorganizado.

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

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

onde $A = v_j$ significa que o atributo A tem o valor v_j. Isto é, $E(A = v_j)$ é a entropia do ramo correspondente ao valor v_j do atributo A.

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,

E_1, E_2, E_3, ...E_k as entropias dos k ramos associados ao atributo escolhido, e

N_1, N_2, N_3 ...N_k 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:

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

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 da Implementação

É possível realizar uma implementação geral dos algoritmos da família TDIDT e redefinir, para algoritmos particulares, o critério de parada t e a função de avaliação *aval*. Esta foi a idéia adotada nesta implementação, feita na linguagem de programação Prolog [Arity 88]. Várias das idéias e técnicas avançadas de programação usadas na implementação foram extraídas de [Turing 87].

As estruturas de dados utilizadas são listas, árvores e árvores binárias. Com o objetivo de realizar o cálculo da função de avaliação *aval* eficientemente, é construída uma árvore binária duplamente indexada que será discutida mais detalhadamente em seções posteriores.

A estrutura geral da árvore de decisão construída pela implementação é:

$$MelhorAtributo - [v_1 : Subarv_1, v_2 : Subarv_2, \ldots, v_n : Subarv_n]$$

onde $v_1, v_2, \ldots, v_n$ são os valores que $MelhorAtributo$ pode assumir e cada uma das $Subarv_i$,$i = 1, 2, \ldots, n$ é uma subárvore que contém todos os exemplos para os quais o valor de $MelhorAtributo$ é v_i. Estas subárvores possuem uma estrutura semelhante ou então representam um valor da classe, quando o nó for uma folha.

Por exemplo, a árvore da Figura 1 é representada da seguinte forma:

```
TSH -
 [ <6.05 : negativo,
   >6.05 :
      FTI -
       [ <64.5 : hipo_primario,
         >64.5 :
            tiroxina -
             [ t : negativo,
               f :
                 cirurgia_tir -
                   [ t : negativo,
                     f :
                       TT4 -
                        [ <150.5 : hipo_compensado,
                          >105.5 : negativo ]
                   ]
             ]
       ]
 ]
```

A implementação realizada leva em conta várias situações que podem acontecer durante o processo de construção da árvore.

Durante o processo de construção da árvore, não é possível classificar exatamente exemplos que tenham atributos com o mesmo valor, mas 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 correspondente com o valor da classe mais provável.

Outro problema tratado é o seguinte: algumas implementações precisam conhecer, durante o processo de construção da árvore, todos os valores de um atributo. Se um determinado valor de um atributo considerado pela árvore de decisão não estiver no conjunto de aprendizado, é criado, associado a esse valor, uma folha adicional, rotulada como *null*. Esta abordagem, que requer conhecimento não apenas dos atributos, mas de todos os possíveis valores que cada atributo pode assumir, exige o pré-processamento de todo o conjunto de exemplos. Além disso, não há garantia que exemplos considerados posteriormente possuam os mesmos valores de atributos que o conjunto de aprendizado.

Para resolver este problema, o critério adotado neste trabalho cria as folhas *null* dinamicamente — após a construção da árvore — se necessário. Isto acontece em uma situação em que é processado um exemplo que não consegue ser classificado pela árvore por possuir um valor de atributo que não está presente na árvore. Tal fato significa que não é possível uma classificação exata para esse exemplo; contudo, há várias formas de contornar este

problema. Uma delas, adotada neste trabalho, é a de computar as probabilidades condicionais de cada classe no ramo que vai da raiz à folha *null* da árvore, e rotular a folha *null* com a classe mais provável.

4 Descrição da Implementação

4.1 Considerações Iniciais

Programar em Prolog consiste em definir relações e interrogar essas relações. Um programa consiste de cláusulas. Elas são de três tipos: *fatos*, *regras* e *interrogações*.

Uma relação pode ser especificada simplesmente por fatos, através da definição da n-upla de objetos que satisfazem a relação, ou então por regras referentes a essa relação.

Um *procedimento* é o conjunto de cláusulas referente à uma mesma relação ou predicado.

Predicados em Prolog 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 será utilizada neste texto. Além disso, quando for conveniente, os argumentos dos procedimentos serão descritos da seguinte forma:

[+] $< arg_n >$: 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 e saída

A seguir será descrita em detalhe a parte central da implementação do sistema, isto é, os programas que constroem a árvore de decisão utilizando como função de avaliação o conceito de entropia.

4.2 Dados Utilizados para Exemplificar a Ação dos Procedimentos

Para exemplificar a ação dos procedimentos serão utilizados os dados correspondentes ao seguinte sistema simples de classificação.

Seja a Tabela 1 de exemplos a respeito de candidatos que participaram do programa "namoro na TV". Para maiores detalhes consultar [Castiñeira 90a]. Cada linha desta tabela é um exemplo ou observação; os rótulos das colunas (**inteligência**, **beleza**, etc.) são os atributos, sendo que o último atributo (**arrumou namorada**) é o atributo de classe. É este atributo que divide os exemplos nas diferentes classes — neste caso uma classe positiva e

outra negativa. Os candidatos que arrumaram namorada formam a classe positiva (+) e os que não arrumaram namorada formam a classe negativa (−).

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

Tabela 1: Conjunto de Observações ou Exemplos

4.3 Procedimentos: Definição e Discussão

No nível mais alto do programa tem-se:

Procedimento 4.3.1 `id3_Turing/1`

```
id3_Turing(T):-
    atributos(A),
    exemplos(E),
    forme_arvore(E,A,T,F).
```

onde os dois primeiros predicados: `atributos/1` e `exemplos/1` referem-se aos procedimentos que fazem o pré-processamento dos dados de entrada; eles estão internamente ligados à forma com que os exemplos são fornecidos ao sistema. O procedimento `atributos(A)` é bem sucedido com `A` instanciado com uma lista que contém todos os atributos dos exemplos considerados — a ordem em que estes atributos aparecem na lista não é importante. Já o procedimento `exemplos(E)` constrói uma lista contendo todos os exemplos a serem considerados. Cada um destes exemplos é por sua vez uma lista de atributos e valores de atributos. Deve ser observado que a classe a que pertence cada um dos exemplos é também considerada um atributo.

Com os dados da Tabela 1, o predicado `atributos(A)` será bem sucedido com:

```
A = [inteligencia, beleza, dinheiro]
```

```
[[inteligencia(s), beleza(b), dinheiro(r), classe(cp)],
 [inteligencia(n), beleza(f), dinheiro(p), classe(cn)],
 [inteligencia(s), beleza(f), dinheiro(p), classe(cp)],
 [inteligencia(s), beleza(f), dinheiro(m), classe(cp)],
 [inteligencia(n), beleza(b), dinheiro(p), classe(cn)],
 [inteligencia(n), beleza(b), dinheiro(m), classe(cp)],
 [inteligencia(n), beleza(b), dinheiro(r), classe(cp)],
 [inteligencia(n), beleza(f), dinheiro(r), classe(cp)] ]
```

Figura 2: Lista de Dados

e o predicado `exemplos(E)`, será bem sucedido com E instanciado com a lista mostrada na Figura 2.

O terceiro procedimento, — **`forme_arvore/4`** — é o que deflagra a construção da árvore de decisão. A idéia geral da atuação desse procedimento é explicada a seguir.

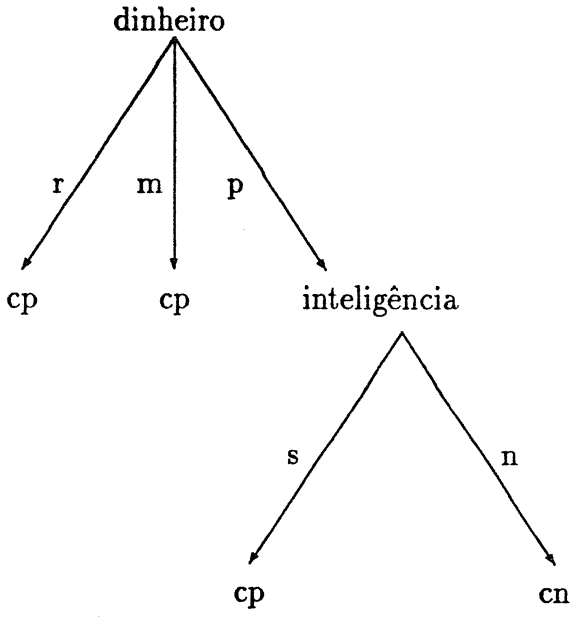
Dada uma lista de exemplos e uma lista de atributos:

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

A entrada para `forme_arvore/4` consiste da lista de exemplos e da lista de atributos. A saída do programa é a árvore de decisão. Para o caso do exemplo considerado — Figura 2 — o procedimento `forme_arvore/4` fornece a seguinte solução:

```
dinheiro - [r:cp, p:inteligencia - [n:cn, s:cp], m:cp]
```

ou, graficamente:



A árvore de decisão é representada pela estrutura geral:

```
Melhor_Atributo - [V1 :  [..], V2 :  [..], .., Vn :  [..]]
```

onde `V1`, `V2`, ..., `Vn` são os valores que `Melhor_Atributo` pode assumir.

O procedimento **`forme_arvore/4`**:

- invoca **`forme_arv/4`**, que efetivamente é o responsável pela construção da árvore;
- identifica se a árvore construída por `forme_arv/4` é exata.

Os argumentos de ambos predicados 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

Procedimento 4.3.2 `forme_arvore/4` e `forme_arv/4`

```
forme_arvore(E,A,T,F):-
    forme_arv(E,A,T,F),
```

```
    ifthen( var(F), F=exata).

forme_arv([],_,null,F).

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

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

forme_arv(Exs,Ats,MelhorAt-SubArvores,F):-
    melhor_at(Ats,Exs,MelhorAt,Exs_p),
    retira(MelhorAt,Ats,NovosAts),
    forme_arvores(Exs_p,NovosAts,SubArvores,F).
```

As três primeiras cláusulas de **forme_arv/4** constituem o critério de parada.

A segunda cláusula de **forme_arv/4** 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. Essa situação é verificada pelo procedimento **mesma_classe/2**, definido a seguir.

Procedimento 4.3.3 mesma_classe/2

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

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

onde o procedimento que fornece a classe de um exemplo é dado por:

Procedimento 4.3.4 class/2

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

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

Deve ser observado que na segunda cláusula de **forme_arv/4**, o conjunto de atributos é irrelevante, razão pela qual a variável anônima _ é usada para representá-lo.

Por exemplo, a interrogação Prolog:

```
?- instancia(X),mesma_classe(X,C).
```

onde instancia/1 é simplesmente utilizado para fins de instanciação, produz como resultado[3]:

```
X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
     [inteligencia(s),beleza(b),dinheiro(p),classe(cp)]]
C = cp
```

A terceira cláusula de **forme_arv/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, e o quarto argumento de **forme_arv/4** é instanciado com o átomo **nao_exata**. Quando isso acontece, determina-se qual é a classe com o maior número de exemplos — classe majoritária — e essa classe rotula o nó folha correspondente. Os procedimentos **classe_majoria/2** e **clase_majoria/3** fazem isso.

Procedimento 4.3.5 **classe_majoria/2** *e* **classe_majoria/3**

```
classe_majoria(Exs,Classe):-
    classe_majoria(Exs,[],Classe).

% criterio de parada, procura a maior classe na lista L1
classe_majoria([],L1,Classe):-
    c_maior(L1,Classe),
    !.

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

A primeira cláusula de **classe_majoria/3** é a condição de parada, quando o procedimento **c_maior/2** determina a classe mais frequente de uma lista de classes, lista esta criada pela segunda cláusula.

[3]As respostas Prolog às interrogações, quando necessário, foram reformatadas, de maneira a promover uma melhor compreensão.

A segunda cláusula varre o conjunto de exemplos e constrói uma lista cujos elementos têm a estrutura

$$classe_i{:}nro\_elementos\_classe_i.$$

Nesta cláusula, para cada exemplo do conjunto de exemplos, `class/2` determina a classe do exemplo e o procedimento `insere_estat/3` incrementa de 1 a contagem dessa classe.

Procedimento 4.3.6 `insere_estat/3`

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

insere_estat(Cl,[Cl:N|R], [Cl:N1|R]):-
    N1 is N + 1,
    !.

insere_estat(Cl,[X:N|R], [X:N|R1]):-
    insere_estat(Cl,R,R1).
```

Seguem-se dois exemplos de execução desse procedimento.

```
?- insere_estat(c,[c1:2,c2:1,c3:4],L).

 L = [c1 : 2,c2 : 1,c3 : 4,c : 1]


?- insere_estat(c2,[c1:2,c2:4,c3:3],L).

 L = [c1 : 2,c2 : 5,c3 : 3]
```

A função do procedimento `c_maior/2` é a de invocar `c_maior/4`. O procedimento `c_maior/2` identifica e separa o primeiro elemento da lista formada por estruturas

$$classe_i{:}nro\_elementos\_classe_i$$

do resto da lista, bem como separa essa estrutura em

$$classe_i \text{ e } nro\_elementos\_classe_i$$

Essa separação é feita com objetivo de aplicar a técnica de determinação do maior elemento de uma lista: o primeiro elemento da lista é o *maior* até que um elemento maior que ele seja encontrado, quando então este elemento encontrado passa a ser o *maior*.

O procedimento `c_maior/4` inicia a execução tendo como seu 3º, 2º e 1º argumentos a primeira classe da lista, o número de elementos dessa primeira classe, e o resto da lista respectivamente. O quarto argumento será instanciado com a classe de maior frequência.

Procedimento 4.3.7 `c_maior/2` *e* `c_maior/4`

```
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).
```

Por exemplo, a interrogação:

```
?- c_maior([c2:3,c3:1,c1:6,c4:8],C).
```

fornece como resultado `C = c4`

A primeira cláusula de `c_maior/4` é a condição de parada, que acontece quando toda a lista foi varrida — o primeiro argumento é a lista vazia. O quarto argumento, que representa a classe com maior frequência da lista, e até então não instanciado, é instanciado nesse momento com o terceiro argumento.

A segunda cláusula de `c_maior/4` trata a situação em que é encontrada uma classe na lista, cuja frequência é maior (ou igual) à da classe até então com a maior frequência. O processo de determinação da maior classe continua, com uma chamada recursiva a `c_maior/4`, com seu terceiro e segundo argumentos instanciados com a nova classe e sua frequência, respectivamente. Nessa chamada o primeiro argumento está instanciado com o resto da lista a ser examinado.

Quando a frequência da nova classe é menor do que a da atual — situação tratada pela terceira cláusula —, acontece uma chamada recursiva a `c_maior/4`, com o primeiro argumento instanciado com o resto da lista, mantendo a classe e a correspondente frequência identificadas até então.

Voltando ao procedimento **forme_arv/4**, (proc. 4.3.2, pg. 12) sua quarta cláusula trata o caso geral. Invoca os procedimentos **melhor_at/4**, **retira/3** e **forme_arvores/4**, cada um deles discutido detalhadamente a seguir.

O procedimento **melhor_at/4** 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 —, dado um conjunto de exemplos. Além de encontrar o melhor atributo, este procedimento particiona o conjunto de exemplos de acordo com os valores que este melhor atributo pode assumir e instancia seu último argumento com uma lista na forma:

$$[Valor_1{:}[E_{11},E_{12},\ldots,E_{1i}],\ldots,Valor_n{:}\ [E_{n1},E_{n2},\ldots,E_{nin}]]$$

onde:

- $Valor_1,\ldots,\ Valor_n$ são os valores que o melhor atributo pode assumir,
- $[E_{j1},\ldots,E_{jij}]$ é a lista de exemplos para os quais o valor do melhor atributo é $Valor_j$, $(1 \leq j \leq n)$.

O procedimento **melhor_at/4** quando atua nos dados da Tabela 1 (pg. 10) evidencia *dinheiro* como o melhor atributo — o de menor entropia — e unifica seu último argumento com uma lista da forma:

```
[ r : [[inteligencia(s), beleza(b), dinheiro(r), classe(cp)],
       [inteligencia(n), beleza(b), dinheiro(r), classe(cp)],
       [inteligencia(n), beleza(f), dinheiro(r), classe(cp)] ],
  p : [[inteligencia(n), beleza(f), dinheiro(p), classe(cn)],
       [inteligencia(s), beleza(f), dinheiro(p), classe(cp)],
       [inteligencia(n), beleza(b), dinheiro(p), classe(cn)] ],
  m : [[inteligencia(s), beleza(f), dinheiro(m), classe(cp)],
       [inteligencia(n), beleza(b), dinheiro(m), classe(cp)] ]
]
```

Esquematicamente, esta lista tem a seguinte forma:

```
[r : Lista_ricos, p : Lista_pobres, m : Lista_mais_ou_menos]
```

Os argumentos de **melhor_at/4** são:

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

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

[?] $< arg_3 >$: o melhor atributo,

[?] $< arg_4 >$: lista de exemplos particionados segundo os valores do melhor atributo.

Procedimento 4.3.8 `melhor_at/4` *e* `melhor_at/7`

```
melhor_at([At_1|Ats],Exs,MelhorAt,Exs_p):-
    avalia(At_1,Exs,Conj_1,Valor_1),
    melhor_at(Ats,Exs,MelhorAt,Exs_p,At_1,Conj_1,Valor_1).

% melhor_at(+,+,?,?,+,+,+)
melhor_at([],_,Melhor,Conjs,Melhor,Conjs,_).

melhor_at([NovoAt|Ats],Exs,M,C,MelhorAt,Conj,Valor):-
    avalia(NovoAt,Exs,NovoConj,NovoValor),
    ifthenelse( NovoValor>Valor,
    %then
       melhor_at(Ats,Exs,M,C,NovoAt,NovoConj,NovoValor),
     %else
       melhor_at(Ats,Exs,M,C,MelhorAt,Conj,Valor)
     ).
```

A interrogação:

```
?-inst(X),melhor_at([dinheiro,beleza,inteligencia],X,M,N).
```

é bem sucedida com:

```
X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(m),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)]]
M = dinheiro
N = [r : [[inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
     m : [[inteligencia(n),beleza(f),dinheiro(m),classe(cn)]],
     p : [[inteligencia(n),beleza(f),dinheiro(p),classe(cp)]]]
```

A função básica de `melhor_at/4`, composto de uma única cláusula, é a de selecionar o primeiro atributo da lista de atributos e através da invocação de `avalia/4`, calcular a entropia desse primeiro atributo, gerar a lista dos exemplos particionada pelos valores

deste primeiro atributo e, com esses valores, ativar o procedimento `melhor_at/7`, que recursivamente ramifica os exemplos e calcula a entropia para cada um dos atributos restantes, escolhendo, a cada passo, o atributo com menor entropia.

Os quatro primeiros argumentos de `melhor_at/7`, têm, respectivamente, o mesmo significado que os quatro primeiros de `melhor_at/4`. O quinto e o sexto argumentos serão instanciados com o melhor atributo temporário e com a lista de exemplos particionados pelos valores desse atributo, respectivamente. O sétimo argumento é o valor da entropia associado ao atributo selecionado.

A condição de parada de `melhor_at/7` é dada pela primeira cláusula, quando a lista de atributos encontra-se vazia. Quando isso acontece, o terceiro argumento (melhor atributo), se instancia com o quinto (melhor atributo temporário) e o quarto argumento (lista de exemplos particionada pelos valores do melhor atributo) se instancia com o sexto argumento (lista de exemplos particionada pelos valores do melhor atributo temporário).

Como o valor da entropia do atributo já foi usado para selecioná-lo, esse valor não interessa mais, motivo pelo qual o sétimo argumento de `melhor_at/7` está representado pela variável anônima.

Na segunda cáusula de `melhor_at/7`, o procedimento `avalia/4` calcula a entropia de um atributo e ramifica os exemplos de acordo com os valores que este atributo assume. Os exemplos são ordenados numa árvore binária duplamente indexada. Isso é feito por razões de eficiência, uma vez que o cálculo da entropia consome um tempo razoável e deve ser realizado em cada nó da árvore de decisão, para cada um dos atributos. Os argumentos de `avalia/4` são:

[+] $< arg_1 >$: atributo selecionado para calcular a entropia

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

[?] $< arg_3 >$: lista de exemplos particionada segundo os valores do atributo

[?] $< arg_4 >$: valor calculado da entropia

Procedimento 4.3.9 `avalia/4`

```
avalia(At,Exs,Exs_p,Entropia):-
    divide(At,Exs,ArvIndex),
    entropia(ArvIndex,Entropia),
    rehash(ArvIndex,Exs_p).
```

A seguinte interrogação:

```
?- ins(X),avalia(dinheiro,X,Y,Z),avalia(inteligencia,X,P,Q),
   avalia(beleza,X,M,N).
```

é bem sucedida com:

```
X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
Y = [r : [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
     p : [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
          [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
          [inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
     m : [[inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(m),classe(cp)]]]
Z = -0.82930377
P = [s : [[inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
          [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
     n : [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
          [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
          [inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(m),classe(cp)]]]
Q = -1.46142627
M = [b : [[inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(p),classe(cn)]],
     f : [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
          [inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(p),classe(cp)]]]
N = -1.9537524
```

Para o procedimento `avalia/4` calcular a entropia e ramificar os exemplos segundo o atributo selecionado — seu primeiro argumento — os exemplos são ordenados numa árvore binária duplamente indexada.

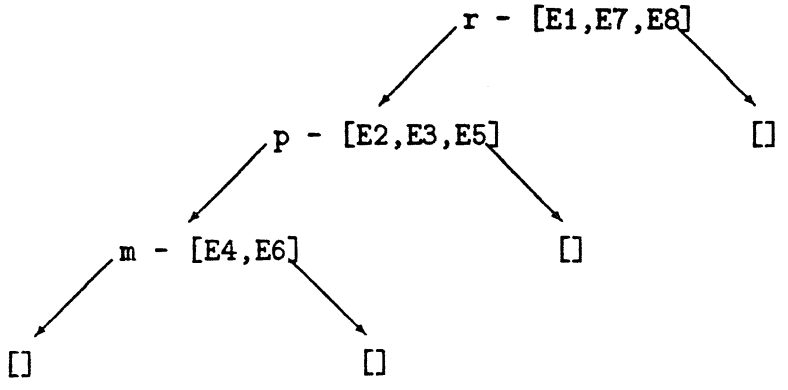
A árvore binária é indexada segundo os valores do atributo e da classe — a sua construção é detalhada logo a seguir.

A estrutura de árvore duplamente indexada é utilizada internamente com o único objetivo de calcular a entropia de forma eficiente; devido à sua importância, ela será explicada em detalhe, utilizando os dados da Figura 2, pg. 11.

A fim de simplificar cada um dos exemplos que compoêm a lista de entrada, cada um deles será referenciado como Ei, ($1 \leq i \leq 8$). Isto é:

```
E1 = [inteligencia(s), beleza(b), dinheiro(r), classe(cp)]
E2 = [inteligencia(n), beleza(f), dinheiro(p), classe(cn)]
E3 = [inteligencia(s), beleza(f), dinheiro(p), classe(cp)]
E4 = [inteligencia(s), beleza(f), dinheiro(m), classe(cp)]
E5 = [inteligencia(n), beleza(b), dinheiro(p), classe(cn)]
E6 = [inteligencia(n), beleza(b), dinheiro(m), classe(cp)]
E7 = [inteligencia(n), beleza(b), dinheiro(r), classe(cp)]
E8 = [inteligencia(n), beleza(f), dinheiro(r), classe(cp)]
```

Se cada um dos elementos da lista `[E1,E2, ...,E8]` for inserido numa árvore binária ordenada pelo valor do atributo `dinheiro` — onde cada nó da árvore é uma lista e as folhas vazias são representadas por uma lista vazia — a seguinte árvore será gerada:



Se a representação Prolog escolhida para árvore for a estrutura:

```
t(Sub_Arvore_E,No,Sub_Arvore_D)
```

onde o primeiro e o terceiro argumentos representam as subárvores esquerda e direita, da árvore cuja raiz é `No`, a árvore anterior será representada pela estrutura:

```
t(t(t([],m-[E4,E6],[]),
```

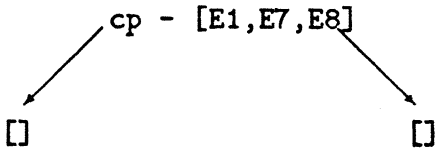
```
 p-[E2,E3,E5],
 []),
r-[E1,E7,E8],
[])
```

Cada nó da árvore, na representação acima, é representado pela seguinte estrutura:

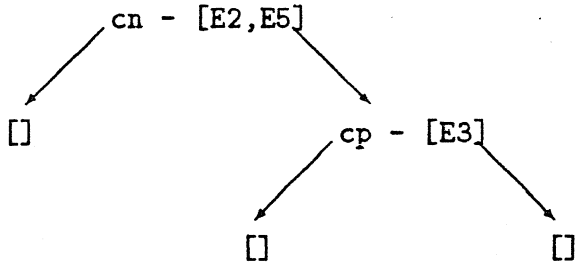
```
Valor_atributo - Lista_de_exemplos.
```

É importante, para o cálculo da entropia, que para cada valor de atributo — representado por um nó da árvore —, sua lista associada de exemplos seja particionada por valor de classe; assim sendo, cada nó será também representado como uma árvore binária semelhante, usando agora na criação desta árvore, o valor da classe. As seguintes estruturas serão obtidas:

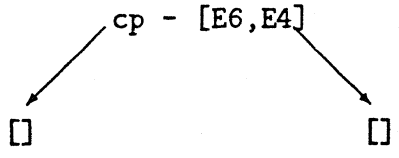
1° nó:



2° nó:



3° nó:



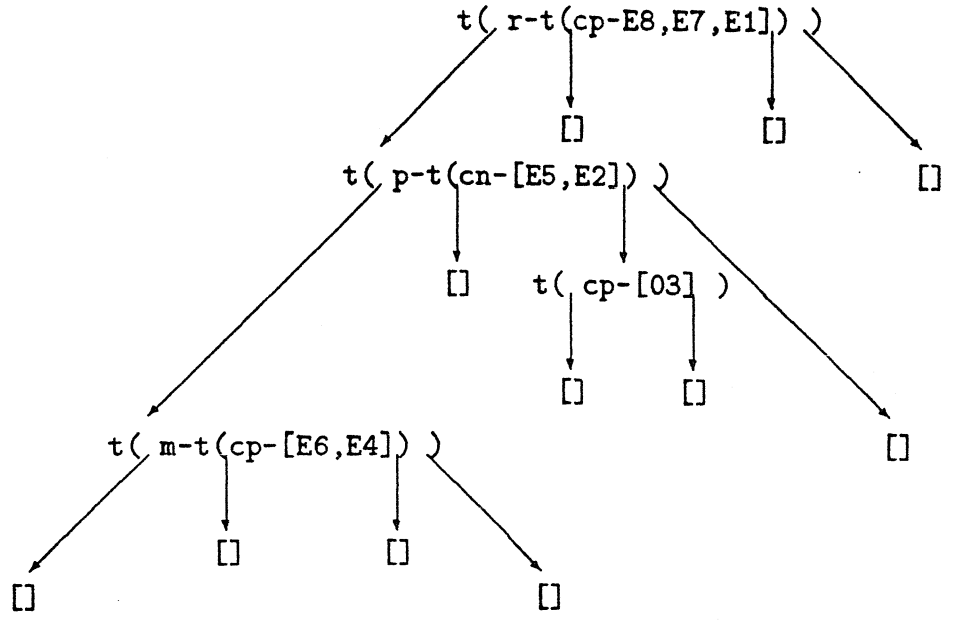


Figura 3: Árvore Duplamente Indexada

Substituindo recursivamente as listas que constituem os nós, obtém-se a estrutura mostrada na Figura 3, cuja representação é dada a seguir.

```
t(t(t([],
       m-t([], cp-[E6,E4], []),
       []
      ),
      p-t([],
          cn-[E5,E2],
          t([], cp-[E3], [])
         ),
      []
    ),
    r-t([], cp-[E8,E7,E1], []),
    []
  )
```

Voltando ao procedimento `avalia/4`, ele invoca os procedimentos `divide/3`, `entropia/2` e `rehash/2`, discutidos a seguir. O procedimento `divide/3` é o responsável pela construção da árvore duplamente indexada, seu terceiro argumento. O procedimento `entropia/2` calcula a entropia do atributo utilizando essa árvore e `rehash/2` restaura a árvore na estrutura de lista de listas, descrita anteriormente. Segue-se um exemplo de execução da conjunção desses três procedimentos, que definem `avalia/4`.

```
?- ins(X),divide(dinheiro,X,H),entropia(H,E),rehash(H,L).
```

```
X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
H =
t(t(t([],
      m - t([],cp -
               [[inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
                [inteligencia(s),beleza(f),dinheiro(m),classe(cp)]],
            []),
      []),
      p - t([],cn -
               [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
                [inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
            t([],cp -
                 [[inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
              [])
           ),
      []),
  r - t([],cp -
           [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
           [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
           [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
        []),
  [])
E = -0.82930377
L = [r : [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
     p : [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
          [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
          [inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
     m : [[inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(m),classe(cp)]]]
```

A função de **divide/3**, na verdade, é a de invocar **divide/4**, que efetivamente constrói a árvore duplamente indexada associada a um atributo, cujos nós são também árvores duplamente indexadas. O argumento que **divide/4** tem a mais do que **divide/3** é o último, inicializado com a lista vazia e utilizado com a função de acumulador — nele é inserida a estrutura da árvore, à medida que ela vai sendo construída. Os argumentos de

`divide/3` são:

[+] < arg_1 >: o atributo que será o primeiro indexador da árvore (o segundo indexador é sempre a classe),

[+] < arg_2 >: lista de exemplos,

[?] < arg_3 >: árvore binária duplamente indexada

Procedimento 4.3.10 `divide/3` e `divide/4`

```
divide(At,Exs,ArvB):-
    divide(At,Exs,ArvB,[]).

% divide(+,+,?,+).
divide(_,[],ArvB,ArvB):-
    !.

divide(At,[E|Exs],ArvB,Acum_ArvB):-
    insere(At,E,Acum_ArvB,ProxAcum_ArvB),
    divide(At,Exs,ArvB,ProxAcum_ArvB).
```

A primeira cláusula de **`divide/4`** é a condição de parada, que acontece quando a lista de exemplos — segundo argumento — estiver vazia; o terceiro argumento se instancia, então, com o quarto argumento.

A segunda e última cláusula de **`divide/4`** trata o caso geral, onde é selecionado o primeiro exemplo da lista de exemplos, para o procedimento **`insere/4`** inserí-lo na árvore. Esta árvore e o resto dos exemplos são passados novamente para **`divide/4`** recursivamente construir a árvore. É o procedimento `insere/4` o responsável por inserir um determinado exemplo na árvore. Seus argumentos são:

[+] < arg_1 >: o atributo que será o primeiro indexador da árvore

[+] < arg_2 >: exemplo a ser inserido na árvore binária,

[+] < arg_3 >: árvore binária antes de inserir o exemplo,

[?] < arg_4 >: árvore binária após a inserção do exemplo

Procedimento 4.3.11 `insere/4`

```
insere(A,E,Conj,ProxConj):-
    valor_at(A,E,Val),
    class(E,Classe),
    key_index(Val-Classe,E,Conj,ProxConj).
```

A primeira vez que **insere/4** é invocado por **divide/4**, seu terceiro argumento está instanciado com a lista vazia.

A segunda e última cláusula de **divide/4** trata o caso geral, onde é selecionado o primeiro exemplo da lista de exemplos para o procedimento **insere/4** inserí-lo na árvore. Esta árvore e o resto dos exemplos são passados novamente para **divide/4** recursivamente construir a árvore.

A seguir é mostrado o uso do predicado **insere/4**. Na primeira execução, o exemplo instanciado com X é inserido na árvore vazia, resultando a árvore L; na segunda execução, o exemplo instanciado com Y é inserido na árvore L, resultando a árvore L1.

```
?- ins([X|[Y|Z]],insere(dinheiro,X,[],L),insere(dinheiro,Y,L,L1).

X = [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]
Y = [inteligencia(n),beleza(f),dinheiro(p),classe(cn)]
Z = [[inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
L = t([],r - t([],
               cp -
                 [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
               []),
      [])
L1 = t(t([],p - t([],
                  cn -
                    [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
                  []),
         []),
       r - t([],
             cp -
               [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
             []),
       [])
```

Para inserir o exemplo na árvore é necessário saber o valor do atributo em questão, bem como a classe do exemplo, que são determinados respectivamente pelos procedimentos **valor_at/3** e **class/2**. A implementação destes dois procedimentos depende da representação de dados adotada para descrever os valores dos atributos e das classes.

O procedimento **valor_at/3** retorna o valor de um atributo em um dado exemplo.

Procedimento 4.3.12 `valor_at/3`

```
valor_at(At, [At_v| _], V):-
    At_v=.. [At,V],
    !.

valor_at(A,[_|Y],V):-
    valor_at(A,Y,V).
```

Por exemplo, a interrogação:

```
?- ins([X|Y]),valor_at(dinheiro,X,V).
```

é bem sucedida com:

```
X = [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]
Y = [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
V = r
```

Com os valores do atributo e da classe, o procedimento `key_index/4` insere o exemplo correspondente na árvore binária. Seus argumentos são:

[+] $< arg_1 >$: índice pelo qual o exemplo será inserido na árvore binária. Este índice pode ser o valor do atributo e da classe (Val-Classe), ou apenas o valor da classe (Classe).

[+] $< arg_2 >$: exemplo a ser inserido na árvore,

[+] $< arg_3 >$: árvore binária antes da inserção do exemplo,

[?] $< arg_4 >$: árvore binária após a inserção do exemplo.

Procedimento 4.3.13 `key_index/4`

```
key_index(Val-Classe,Objeto,[],t([],Val-SubArv,[])):-
    !,
    key_index(Classe,Objeto,[],SubArv).
```

```
key_index(Val-Classe,Objeto,t(E,Val-S,D),t(E,Val-Sn,D)):-
    !,
    key_index(Classe,Objeto,S,Sn).

key_index(V-C,O,t(E,Val-S,D),t(En,Val-S,D)):-
    V @< Val,
    !,
    key_index(V-C,O,E,En).

key_index(V-C,O,t(E,Val-S,D),t(E,Val-S,Dn)):-
    V @> Val,
    !,
    key_index(V-C,O,D,Dn).

key_index(Classe,Obj,[],t([],Classe-[Obj],[])):-
    !.

key_index(Classe, Obj, t(E,Classe-Objs,D),t(E,Classe-[Obj|Objs],D)):-
    !.

key_index(C,O,t(E,Cl-Os,D),t(En,Cl-Os,D)):-
    C @<Cl,
    !,
    key_index(C,O,E,En).

key_index(C,O,t(E,Cl-Os,D),t(E,Cl-Os,Dn)):-
     C @> Cl,
     key_index(C,O,D,Dn).
```

O procedimento **key_index/4** tem 8 cláusulas. De uma forma geral, pode-se dizer que ele procura primeiro o nó onde inserir o exemplo, segundo o valor de seu atributo (as quatro primeiras cláusulas). Uma vez encontrado este nó, como o próprio nó também é uma árvore binária ordenada pela classe, o programa volta a procurar onde inserir o exemplo, agora, de acordo com o valor da sua classe. Esta última pesquisa é realizada pelas 4 últimas cláusulas do programa.

A primeira cláusula

```
key_index(Val-Classe,Objeto,[],t([],Val-SubArv,[])):-
    !,
    key_index(Classe,Objeto,[],SubArv).
```

trata a inserção de um exemplo na árvore vazia; ele é inserido na raiz. Pode-se observar nesta cláusula, que **key_index/4** é invocado novamente para inserir o exemplo, desta vez

segundo o valor da classe, numa subárvore que também está vazia.

Segue-se um exemplo de execução do procedimento **key_index/4**, quando da inserção de um exemplo na árvore vazia.

```
?- ins([X|Y]),key_index(r-cp,X,t([],r-S,[])).

X = [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]
Y = [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
S = t([],cp - [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],[])
```

A segunda cláusula

```
key_index(Val-Classe,Objeto,t(E,Val-S,D),t(E,Val-Sn,D)):-
    !,
    key_index(Classe,Objeto,S,Sn).
```

também considera a inserção de um exemplo na raiz da árvore, mas, neste caso, a árvore não está vazia. O valor do atributo do exemplo corresponde ao indexador do nó raiz da árvore. O procedimento **key_index/4** é chamado novamente para agora inserir o exemplo corretamente na subárvore da raiz, segundo o valor da sua classe. Segue-se um exemplo onde apenas as duas primeiras cláusulas de **key_index/4** são utilizadas.

```
?- ins([X|[Y|Z]]),key_index(r-cp,X,[],r-S,[])),key_index(r-cn,Y,t(M,r-
S,N),t(O,P-Q,R)).

X = [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]
Y = [inteligencia(n),beleza(f),dinheiro(p),classe(cn)]
S = t([],cp - [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],[])
M = _01E4
N = _01EC
O = _01E4
P = r
Q = t(t([],cn -
            [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
        []),
      cp -
       [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
```

```
      [])
R = _01EC
```

A terceira cláusula

```
key_index(V-C,O,t(E,Val-S,D),t(En,Val-S,D)):-
    V @< Val,
    !,
    key_index(V-C,O,E,En).
```

insere o exemplo em algum nó da subárvore esquerda; essa situação acontece quando o valor do atributo do exemplo é menor do que o valor do atributo do nó que está sendo comparado. Analogamente, a quarta cláusula

```
key_index(V-C,O,t(E,Val-S,D),t(E,Val-S,Dn)):-
    V @> Val,
    !,
    key_index(V-C,O,D,Dn).
```

insere o exemplo na subárvore direita.

Como já comentado, as quatro últimas cláusulas de `key_index/4` tratam da inserção do exemplo segundo o valor de uma classe (segundo indexador). Isso significa que já foi encontrado o nó onde deve ser inserido o exemplo segundo o valor do seu atributo; entretando, como este nó também é uma árvore, ela deve ser pesquisada de maneira análoga, agora, segundo o valor da classe.

Os termos árvore e subárvore aplicados às quatro próximas cláusulas referir-se-ão às árvores e subárvores que representam cada um dos nós da árvore duplamente indexada classificada por valor de atributo. Essas cláusulas são análogas, uma a uma, na ordem em que aparecem, às quatro anteriores. O primeiro argumento das quatro primeiras cláusulas é o par `Valor - Classe`; das quatro últimas é apenas `Classe`.

A quinta cláusula

```
key_index(Classe,Obj,[],t([],Classe-[Obj],[])):-
    !.
```

insere o exemplo na raiz, uma vez que a árvore é uma árvore vazia. A sexta cláusula

```
key_index(Classe, Obj, t(E,Classe-Objs,D),t(E,Classe-[Obj|Objs],D)):-
    !.
```

insere o exemplo na lista de exemplos associados ao nó raiz de uma subárvore, uma vez que o valor da classe do exemplo é igual ao do nó raiz. A sétima cláusula

```
key_index(C,O,t(E,Cl-Os,D),t(En,Cl-Os,D)):-
    C @<Cl,
    !,
    key_index(C,O,E,En).
```

insere o exemplo na subárvore esquerda. O valor da classe do exemplo é menor que a do nó raiz considerado. Desta forma, **key_index/4** é invocado novamente para pesquisar a subárvore esquerda. A última cláusula

```
key_index(C,O,t(E,Cl-Os,D),t(E,Cl-Os,Dn)):-
    C @> Cl,
    key_index(C,O,D,Dn).
```

insere o exemplo na subárvore direita.

É importante lembrar que o procedimento **avalia/4** (proc. 4.3.9, pg. 19) formado por apenas uma cláusula, indiretamente determina o valor da entropia de um determinado atributo. Como já foi comentado, para um determinado atributo, **avalia/4**:

- constrói uma árvore binária duplamente indexada, ordenada por valores daquele atributo e classes dos exemplos, invocando **divide/4**;
- determina a entropia do atributo em questão, utilizando a árvore binária duplamente indexada, invocando **entropia/2**
- constrói uma lista, cujos elementos têm a estrutura:

$$valor\_de\_atributo : [E_1,\ldots,E_n]$$

onde os exemplos $E_1,\ldots,E_n$ têm como valor do atributo, *valor_de_atributo*, invocando **rehash/2**.

Deve-se ressaltar que o conceito de entropia é o já discutido na seção 2.3. Entretanto, o algoritmo implementado não considera o sinal negativo da equação (1) — pg. 6 — e, portanto, considera como melhor atributo aquele de maior entropia e não o de menor — veja o procedimento **melhor_at/7** (proc. 4.3.8, pg. 18).

Relembrando, para calcular a entropia de um atributo devem ser agrupados 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). Estes números, obtidos para cada valor do atributo, ou para cada ramo, são multiplicados

pelo número de exemplos do correspondente ramo e somados para compor o valor final da entropia — equação (2) — pg. 7. É importante notar que no cálculo final da entropia, esta implementação do algoritmo não divide a somatória pelo número total de exemplos. Este, porém, não é um detalhe relevante uma vez que o número total de exemplos é um valor positivo e constante para todos os atributos.

O procedimento **entropia/2** usa a construção da árvore indexada feita por **divide/3** para efetuar os cálculos acima descritos, uma vez que os exemplos já estão ordenados por valor de atributo e de classe. Os procedimentos **entropia/2** e **entropia/3** contam o número de exemplos pertencentes a cada conjunto da partição.

Procedimento 4.3.14 `entropia/2` *e* `entropia/3`

```
% entropia(+,?).
entropia(ArvB_2Index,Entropia):-
    entropia(ArvB_2Index,0,Entropia).

entropia([],V,V):-
    !.

entropia(Arv_2_Index,M,V):-
    valor(Arv_2_Index,Arv_1_Index,_,NovaArv_2_Index),
    contrib_entropia(Arv_1_Index,Mp),
    M1 is M+Mp,
    entropia(NovaArv_2_Index,M1,V).
```

O procedimento **entropia/2** tem como primeiro argumento a árvore binária indexada e como segundo — argumento de saída — o valor da entropia. Em **entropia/3** o segundo argumento é um argumento acumulador para o valor da entropia, inicialmente unificado com zero.

A primeira cláusula de **entropia/3** é o critério de parada, quando a árvore toda foi varrida e o terceiro argumento é instanciado com o segundo. A segunda cláusula trata do caso geral. A idéia é visitar cada nó da árvore binária e calcular como os exemplos pertencentes a cada nó contribuem para o cálculo da entropia. Os valores obtidos em cada nó são somados e o resultado é unificado com o parâmetro acumulador, obtendo-se assim o valor da entropia.

É importante lembrar que cada nó da árvore binária está indexado segundo o valor de atributo selecionado, ou seja, cada nó contém todos os exemplos que têm um determinado valor do atributo — Figura 3. Desta forma, a contribuição da entropia de cada nó é a contribuição da entropia para cada valor do atributo.

O procedimento **valor(Arv_2_Index,Arv_1_Index,_,NovaArv_2_Index)** retira o nó terminal, ou folha, mais à esquerda da árvore **Arv_2_Index**, unifica esta estrutura — que

por sua vez também é uma arvore binária indexada pela classe — com `Arv_1_Index`, e retorna no seu último argumento, `NovaArv_2_Index`, a árvore duplamente indexada que resta após a retirada da folha.

Procedimento 4.3.15 valor/4

```
valor(t([],V-SArv,[]),SArv,V,[]):-
    !.

% esquerda
valor(t(E,CO,D),SArv,V,t(En,CO,D)):-
    valor(E,SArv,V,En),
    !.

% direita
valor(t(E,CO,D),SArv,H,t(E,CO,Dn)):-
    valor(D,SArv,H,Dn),
    !.
```

O seguinte exemplo ilustra a ação desse procedimento.

```
?- ins(X),divide(dinheiro,X,SH),valor(SH,SH1,_,SH2).

X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
SH =
t(t(t([],
      m - t([],cp -
                [[inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
                 [inteligencia(s),beleza(f),dinheiro(m),classe(cp)]],
            []),
      []),
    p - t([],cn -
              [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
               [inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
              t([],cp -
                   [[inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
                []))
```

```
        ),
    []),
  r - t([],cp -
          [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
           [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
           [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
        []),
  [])
SH1 = t([],cp -
          [[inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(m),classe(cp)]],
        [])
SH2 =
t(t([],
    p - t([],cn -
            [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
             [inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
          t([],cp -
              [[inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
            [])
        ),
    []),
  r - t([],cp -
          [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
           [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
           [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
        []),
  [])
```

A condição de parada de `valor/4`, tratada pela sua primeira cláusula, acontece quando na árvore binária indexada é identificada uma estrutura do tipo

```
t([],V-SArv,[])
```

ou seja, um nó do tipo folha. A segunda cláusula desce na árvore binária indexada buscando pela folha mais à esquerda. Se essa busca da folha mais à esquerda não for bem sucedida, a terceira cláusula é ativada, e ela inicia então uma busca da folha mais à direita.

A folha retirada da árvore por `valor/4` é passada para `contrib_entropia/2` que calcula a contribuição de um determinado valor do atributo no cálculo da entropia.

Após o cálculo da contribuição de um determinado nó da árvore, `entropia/3` é invocado recursivamente com o restante da árvore binária — primeiro argumento — e o valor da entropia contabilizado até o momento — segundo argumento.

O procedimento `contrib_entropia/2` é bem sucedido quando `contrib_entropia/4` também o for.

Procedimento 4.3.16 `contrib_entropia/2`

```
% contrib_entropia(+,?).
contrib_entropia(SubConj,Contrib):-
     contrib_entropia(SubConj,[],0,Contrib).
```

Os argumentos e a definição de `contrib_entropia/4` são:

[+] $< arg_1 >$: árvore binária unicamente indexada por classe

[+] $< arg_2 >$: lista que contém o número de exemplos associados a cada nó da árvore binária indexada por classe.

[+] $< arg_3 >$: número total de exemplos ordenados pela árvore.

[?] $< arg_4 >$: valor calculado da contribuição do primeiro argumento no cálculo da entropia.

Procedimento 4.3.17 `contrib_entropia/4`

```
contrib_entropia([],NroPorClass,Total,Contrib):-
    !,
    calcula_entropia(NroPorClass,Total,0,Contrib).

contrib_entropia(SubConj,Sep,Tot,Contrib):-
    valor(SubConj,L,_,NovoSubConj),
    comprimen(L,N),
    NovoTot is Tot+N,
    contrib_entropia(NovoSubConj,[N|Sep],NovoTot,Contrib).
```

Por exemplo, a interrogação:

```
?-ins(X),divide(beleza,X,SH),valor(SH,SH1,_,SH2),
                                    contrib_entropia(SH1,C).
```

é bem sucedida com:

```
X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
SH =
t([],
  b - t(t([],
          cn - [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)]],
          []),
        cp - [[inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
              [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
              [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
        []),
  t([],f - t([],
             cn - [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
             t([],
               cp - [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
                     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
                     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
               [])
            ),
    [])
 )
SH1 =
t([],
  cn - [[inteligencia(n),beleza(f),dinheiro(p),classe(cn)]],
  t([],
    cp - [[inteligencia(n),beleza(f),dinheiro(r),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
          [inteligencia(s),beleza(f),dinheiro(p),classe(cp)]],
    [])
  )
SH2 =
t([],
  b - t(t([],
          cn - [[inteligencia(n),beleza(b),dinheiro(p),classe(cn)]],
          []),
        cp - [[inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
              [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
              [inteligencia(s),beleza(b),dinheiro(r),classe(cp)]],
        []),
```

```
  [])
C = -0.9768762
```

A primeira cláusula do procedimento `contrib_entropia/4` é o critério de parada. Ela é bem sucedida quando a árvore já foi totalmente percorrida, isto é, quando o seu primeiro argumento for a lista vazia. Neste caso, `NroPorClass` é a lista que contém o número de exemplos contidos em cada nó da árvore, ou seja, o número de exemplos que pertencem a cada classe, e `Total` é o número total de exemplos na árvore.

O primeiro e segundo argumento de `calcula_entropia/4` — `NroPorClass` e `Total` — são utilizados pelo procedimento `calcula_entropia(NroPorClass,Total,0,Contrib)` (proc. 4.3.18, pg. 37) para realizar o cálculo correspondente à equação (1), cujo valor é unificado com o último argumento do procedimento `calcula_entropia/4`, comentado logo a seguir.

A segunda cláusula é responsável por percorrer a árvore retirando cada um dos seus nós e contando o número de elementos que o nó retirado possui. Deve-se lembrar que esta árvore é um nó da árvore duplamente indexada; ela contém o subconjunto dos exemplos que tem um determinado valor do atributo para o qual está sendo calculada a entropia.

O procedimento `valor(ClassTree,L,_,NewClassTree)` retira um nó terminal da árvore `ClassTree` e unifica seu último argumento com a árvore restante. O nó retirado, `L`, é uma lista que contém todos os exemplos que pertencem à mesma classe. O procedimento `comprimen(L,N)` conta o número de elementos do nó `L`. Este número é acumulado para obter o número total de exemplos na árvore — `NovoTot` —, e também é inserido numa lista — segundo argumento da chamada recursiva de `contrib_entropia` — que conterá o número de exemplos que há em cada nó da árvore.

Como já foi dito, é depois de percorrer toda a árvore que `calcula_entropia/4` é invocado. Seus argumentos e a sua definição são dados a seguir:

[+] $< arg_1 >$: lista que contém o número de exemplos que pertencem a cada classe

[+] $< arg_2 >$: número total de exemplos na árvore.

[+] $< arg_3 >$: parâmetro acumulador para a contribuição da entropia

[?] $< arg_4 >$: valor da contribuição da árvore no cálculo da entropia.

Procedimento 4.3.18 `calcula_entropia/4`

```
calcula_entropia([ ],Total,Contrib,ContFin):-
    ContFin is Contrib * Total.

calcula_entropia([N|Rest],Total,Temp,Contrib):-
    X is N/Total,
```

```
    Xlog is log(X),
    NovoTemp is Xlog * X + Temp,
    calcula_entropia(Rest,Total,NovoTemp,Contrib).
```

A primeira cláusula é o critério de parada. O valor já contabilizado — terceiro argumento — é multiplicado pelo número total de exemplos na árvore e unificado com o quarto argumento. A segunda cláusula realiza recursivamente o cálculo da equação (1).

O procedimento `valor/4` (proc. 4.3.15, pg. 33) quando utilizado no cálculo da entropia — invocado por `entropia/3` (proc. 4.3.14, pg. 32) e `contrib_entropia/4` (proc. 4.3.17, pg. 35) — não precisa instanciar o seu terceiro argumento. Esse argumento será utilizado quando `valor/4` for invocado por `rehash/3` (proc. 4.3.19, pg. 38).

O terceiro procedimento invocado por **`avalia/4`** é o **`rehash/2`**, que a partir da árvore duplamente indexada ordenada por valores de um atributo e da classe dos exemplos — primeiro argumento —, constrói uma outra representação para essa árvore: lista de estruturas da forma

$$valor\_de\_atributo : [E_1,\ldots,E_n].$$

Tal construção é efetivamente realizada pelo procedimento **`rehash/3`**.

Procedimento 4.3.19 `rehash/2` e `rehash/3`

```
rehash(ExH,Ex):-
    rehash(ExH,Ex,[]).

rehash([],E,E):-
    !.

rehash(ArvH,Ex,Temp):-
    valor(ArvH,SubArv,V,RestTH),
    !,
    rehash(SubArv,ListaOfListas),
    achata1(ListaOfListas,Lista),
    rehash(RestTH,Ex,[V:Lista|Temp]).

rehash(Lista,[Lista|S],S).
```

O procedimento **`melhor_at/4`** (proc. 4.3.8, pg. 18) avalia cada um dos atributos, calculando a sua entropia, e elege o melhor entre eles para a ramificação da árvore.

Lembrando novamente, a quarta cláusula de **`forme_arv/4`** (proc. 4.3.2, pg. 12) trata do caso geral; invoca **`melhor_at/4`**, para determinar a raiz da árvore e invoca os procedimentos **`retira/3`** e **`forme_arvores/4`**, este último definido a seguir. O procedimento

`retira/3` simplesmente retira um elemento de uma lista de elementos; no caso será usado para retirar um atributo da lista de atributos.

Procedimento 4.3.20 `forme_arvores/4`

```
forme_arvores([],_,[],F):-
    !.

forme_arvores([Val:Exs|Rest],Ats,[Val:Arv|Arvs],F):-
    forme_arv(Exs,Ats,Arv,F),
    forme_arvores(Rest,Ats,Arvs,F).
```

A primeira cláusula de `forme_arvores/4` é a condição de parada, quando o conjunto de exemplos é a lista vazia. A segunda cláusula trata do caso geral. Segue-se um exemplo que mostra a construção da árvore.

```
?- ins(X), Y=[dinheiro,beleza,inteligencia],forme_arvore(X,Y,T,F).

X = [[inteligencia(s),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(p),classe(cn)],
     [inteligencia(s),beleza(f),dinheiro(p),classe(cp)],
     [inteligencia(s),beleza(f),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(p),classe(cn)],
     [inteligencia(n),beleza(b),dinheiro(m),classe(cp)],
     [inteligencia(n),beleza(b),dinheiro(r),classe(cp)],
     [inteligencia(n),beleza(f),dinheiro(r),classe(cp)]]
Y = [dinheiro,beleza,inteligencia]
T = dinheiro - [r : cp,p : inteligencia - [n : cn,s : cp],m : cp]
F = exata
```

Observe que neste caso `forme_arv/4`, que faz parte da definição de `forme_arvore/4`, é bem sucedido e o seu quarto argumento permanece não instanciado. É `forme_arvore/4` que o instancia com o átomo `exata`.

4.4 Sumário dos Procedimentos Definidos

As seguintes tabelas mostram os procedimentos que fazem parte do programa. Nelas foi usada a seguinte notação:

- `predicado/aridade` - página onde está definido
- $\hookleftarrow$ para indicar que o procedimento é recursivo

id3_Turing/1 - 10	atributo/1 exemplos/1 forme_arvore/4 - 12	forme_arv/4 - 12	mesma_classe/2 - 13 classe_majoria/2 - 14	class/2 - 13 classe_majoria/3 - 14 ↩ cont. Tabela 3	
			melhor_at/4 - 18	avalia/4 - 19 melhor_at/7 - 18	avalia/4 - 19 cont. Tabela 4
			retira/3		
			forme_arvores/4 - 39 ↩	forme_arv/4 - 39	

Tabela 2: Procedimentos Implementados

classe_majoria/3 - 14 ↩	c_maior/2 - 16 class/2 - 13 insere_estat/3 - 15	c_maior/4 - 16 ↩

Tabela 3: Continuação

avalia/4 - 19 ↩	divide/3 - 25	divide/4 - 25 ↩	insere/4 - 25	valor_at/3 - 27 ↩ class/2 - 13 key_index/4 - 27 ↩	
	entropia/2 - 32	entropia/3 - 32 ↩	valor/4 - 33 ↩ contrib_entropia/2 - 35	contrib_entropia4/4 - 35	calcula_entropia/4 - 37 ↩ valor/4 - 33 ↩ comprimen/2
	rehash/2 - 38	rehash/3 - 38			

Tabela 4: Continuação

5 Conclusões e Trabalhos Futuros

Neste trabalho foi mostrado, em detalhes, a implementação Prolog de um sistema que constrói árvores indutivas de decisão, bem como foram discutidas as principais idéias que subsidiaram tal implementação.

A implementação realizada é abrangente e permite, fazendo algumas modificações, utilizar diversas funções de avaliação e critérios de parada. Ela também confirma que a utilização de estruturas de dados apropriadas pode influenciar diretamente a eficiência do sistema. Em particular, a estrutura de árvore duplamente indexada, utilizada para calcular a entropia, faz com que este sistema fique muito mais rápido quando comparado com outras implementações.

A implementação realizada utiliza técnicas avançadas de programação Prolog que, infelizmente, não são publicadas. Uma maior divulgação destes tipos de técnicas com certeza contribuiria para uma melhora no nível dos programas Prolog [O'Keefe 90]. Esse problema já foi evidenciado e comentado por Fernando Pereira [Pereira 87].

Com esta publicação, espera-se contribuir para a difusão de técnicas avançadas de programação, visando auxiliar futuros projetistas e programadores Prolog.

Como trabalhos futuros, pretende-se:

- incluir no sistema apresentado opções para construir a árvore de decisão com informações adicionais, tais como o número total de exemplos classificados em cada um dos nós da árvore e o número de exemplos desclassificados;
- implementar o mecanismo de *janela*, fundamental quando o número de exemplos do conjunto de aprendizado E é grande. O mecanismo de janela consiste em selecionar aleatoriamente um subconjunto S de exemplos do conjunto E. A árvore de decisão é gerada com os exemplos contidos no subconjunto S, e então verifica-se se os exemplos não considerados são também classificados por esta árvore. Se houver exemplos não classificados, eles devem ser adicionados à janela e o processo de construção da árvore é repetido até que não existam mais exemplos não classificados pela árvore de decisão.;
- implementação do mecanismo de poda — utilizado para melhorar a precisão da classificação da árvore.

Referências

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

[Arity 88] Arity Corporation. *The Arity/Prolog Programming Language.* 1988.

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

[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, 1987, pp 31-45.

[Michie 90] Michie, D. *Human and Machine Learning of Descriptive Concepts.* ICOT Journal, 27, Institute For New Generation Computer Technology, March 1990, pp 2-20.

[Mingers 89] Mingers, J. *An Empirical Comparison of Selection Measures for Decision-Tree Induction.* Machine Learning 3, 1989, pp 319-342.

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

[Pereira 87] Pereira, F. The Indiscipline of Prolog Programming. AI Expert, June 1987, pp 7-8.

[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, 1979, pp 168-201.

[Quinlan 86a] Quinlan, J.R. *Induction of Decision Trees.* Machine Learning 1, 1986, pp 81-106.

[Quinlan 86b] 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, 1986, pp 183-203.

[Quinlan 87] Quinlan, J.R. *Simplifying Decision Trees.* Int. J. Man-Machine Studies 27, 1987, pp 221-234.

[Turing 87] *Advanced Prolog.* Turing Institute, 1987.

[Utgoff 89] Utgoff, P.F. *Incremental Induction of Decision Trees.* Machine Learning, Nº4,1989, pp 161-186.

NOTAS DO ICMSC

Nº 97/91 - DIAS, I.; MICALI, A. - Ordres d'un LG-anneau et idéaux premiers d'un anneau de Witt

Nº 96/91 - Atas da Reunião de Singularidades Reais e Complexas

Nº 95/91 - NUNES, W.V.L. - On the global well-posedness for the transitional Korteweg-de Vries equation in Sobolev space

Nº 94/91 - ACHCAR, J.A.; ROSALES, O.L.A. - A Bayesian for accelerated life tests assuming an inverse Gaussian distribution

Nº 93/91 - DIAS, I.; MICALI, A. - Anneaux de Witt et extensions de Galois

Nº 92/91 - RODRIGUES, J. - The Kullback - Leiber approximation of the marginal posterior desity: an application to the linear functional model

Nº 91/91 - SCOTT, D.R.; NUNES, M.G.V. - Focus-driven search for deciding what to say in reply to questions

Nº 90/91 - SOUZA, C.S.; NUNES, M.G.V. - On the role of text generation in knowledge based systems interfaces

Nº 89/91 - MORABITO NETO, R.N.; ARENALES, M.N. - On solving large two-dimensional guillotine cutting problems

Nº 88/91 - MICALI, A.; VILLAMAYOR; O.E. - Algèbres de Clifford sur un corps de caracterisque 2