Geração de Árvore de Decisão Utilizando o Algoritmo ID3

Hugo Parente Tomás Correia Dalton Cézane

27 de setembro de 2007

Abstract

Este artigo descreve o funcionamento e implementação de um gerador, visualizador e testador de árvore de decisão a partir de base de dados disponivéis em arquivos CSV (Comma Separated Values) implementado em Ruby utilizando o algoritmo ID3 (Indutive Decision Tree)

Sumário

Lista de Figuras

Lista de Listagens

1 Introdução

O Objetivo do projeto é gerar uma árvore de decisão utilizando o algoritmo ID3 a partir de uma base de dados quaisquer, e com a árvore de decisão gerada será possível:

- Visualizar a representação textual da árvore.
- Visualizar a representação gráfica da árvore.
- Executar a árvore de decisão
- Testar a árvore de decisão contra outra base de dados.

O projeto foi escrito na linguagem Ruby, uma linguagem bastante produtiva que possibilitou uma considerável redução no tempo de implementação do projeto caso ele fosse implementado em outra linguagem imperativa como C++ ou Java.

No decorrer do relatório há trechos de códigos acompanhados de suas devidas explanações, fazendo com que mesmo sem saber nada sobre Ruby seja possível entender sobre como foi feita a implementação do algoritmo.

2 *Metodologia*

O processo de desenvolvimento foi a programação em trio. Foi decidido em apenas uma reunião algo para implementar que pertencesse ao escopo da disciplina. Optamos por Árvore de Decisão. Visto que os algoritmos implementados são de complexidade relativamente simples os testes de corretude foram baseados em comparações com vários exemplos com resultados conhecidos.

3 O algoritmo ID3

Originalmente desenvolvido na Universidade de Sydney por J. Ross Quinlan que o apresentou em 1975 em seu livro Machine Learning, vol. 1. o algoritmo tem como objetivo gerar uma árvore de decisão a partir de uma base de conhecimento.

Suponha que você tem uma base de dados com vários elementos, seus atributos e sua classificação. O algoritmo ID3 (Inductive Decision Tree) gera uma árvore de decisão a partir dessa base de dados e a utiliza para classificar futuros elementos que venham a surgir.

A árvore de decisão é utilizada para esta classificação, cada nó não-folha da árvore é uma pergunta sobre um atributo do elemento avaliado, cada nó folha é uma classificação e os arcos de um nó pai para um nó filho são os possíveis valores para o atributo que o nó pai representa.

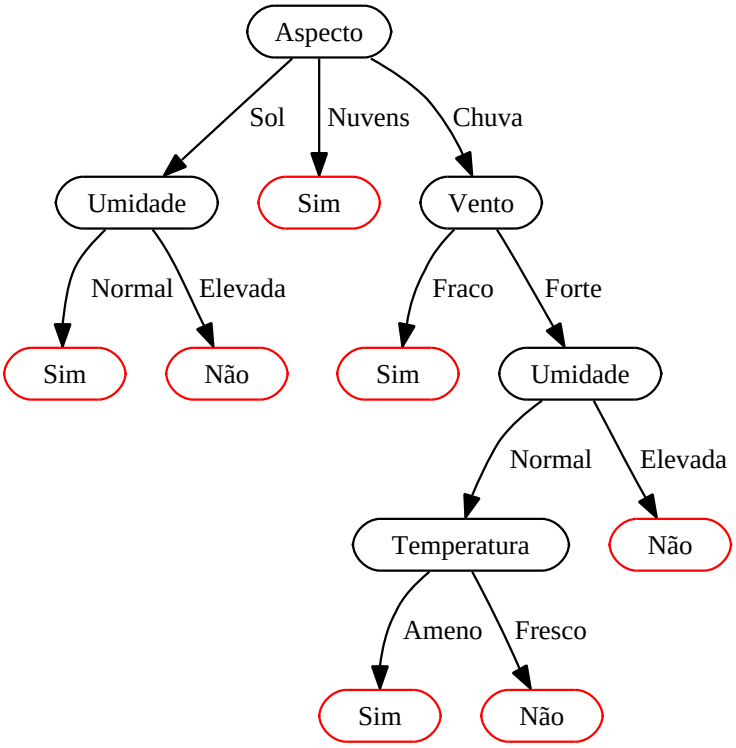

Figura 3.1: *Árvore de decisão para a questão "Jogar ou não jogar tênis?"*

De acordo com a árvore da figura 3.1 se o clima estiver ensolarado e com umidade elevada eu provavelmente não jogaria tênis.

Como é de se esperar a ordem das perguntas fazem total diferença no número de perguntas necessárias para

obter uma resposta, por isso ao montar a árvore de decisão o algoritmo ID3 tenta escolher primeiro os atributos que melhor dividem as amostras da base de dados, estas escolhas são baseadas em duas propriedades estatísticas, a entropia e o ganho.

A entropia é um valor de 0 à 1 que diz o quão desordenado esta a classificação de uma base de dados. Entropia igual à 1 significa que se eu pegar um elemento qualquer da base de dados a probabilidade dele estar na classificação X é igual a dele estar em qualquer outra classificação, se igual à 0 significa que existe a probabilidade de 100% dele estar na classificação X e é claro, 0% de estar em qualquer outra.

$$Entropia(S) = \sum_{i=0}^{n} -p_i log_2 p_i \tag{3.1}$$

A entropia de um conjunto S de dados dividido em n classes é calculada pela equação 3.1, onde p_i é a probabilidade de um elemento pertencer a classe i.

O ganho, mede o peso que um determinado atributo tem para classificar um elemento.

$$Ganho(S,A) = Entropia(S) - \sum_{v=0}^{n} \left(\frac{|S_v|}{|S|} \cdot Entropia(S_v) \right) \tag{3.2}$$

O ganho de um atributo A, que tem n possíveis valores, sobre uma base de dados S é dado pela equação 3.2, onde $|X|$ é o número de elementos no conjunto X e S_v é o subconjunto de S onde todos os elementos possuem o atributo A com o valor v.

O algoritmo ID3 recebe inicialmente um conjunto de dados S e um nó pai N que inicialmente é nulo.

- Se a base de dados tem entropia zero, crie um nó folha com o valor da classificação que restou e retorne.
- Selecione o atributo com maior ganho e crie um nó na árvore para ele.
- Se N é nulo então este será o nó raiz, caso contrário N será o pai deste nó.
- Para cada possível valor V desse atributo T:
  - Crie um subconjunto da base de dados onde $T = V$.
  - Chame o algoritmo recursivamente com o subconjunto da base de dados e o nó criado.

4 Implementação

O projeto foi implementado em Ruby, possui uma interface via linha de comando e é capaz de gerar uma árvore de decisão a partir de uma base de dados no formato CSV [1] (Comma-separated values) que pode ser gerada em qualquer[1] planilha eletrônica.

O programa recebe uma base de dados, constroi a árvore de decisão e depois faz alguma coisa com ela, essa alguma coisa pode ser: imprimir a árvore na saída padrão, gerar um arquivo PNG com a árvore, executar a árvore ou testar a árvore contra outro banco de dados.

4.1 Carregando a base de dados

O programa possui uma classe `Table`, que representa uma tabela, ou seja uma base de dados. Ruby já oferece uma classe para carregar arquivos CSV em memória, logo podemos criar instâncias de `Table` a partir de arquivos CSV, para isso há o método estático `Table.fromCSV` que recebe o nome do arquivo CSV que será carregado.

Listagem 4.1: Criando uma tabela a partir de um arquivo CSV

```
1   # Cria uma tabela a partir de um arquivo CSV
2   def self.fromCSV(csv_file)
3       data = CSV.open(csv_file, 'r').entries
4       raise "Dados insuficientes" if data.length < 2
5       header = data.shift
6       return Table.new(header, data)
7   end
```

É esperado que a base de dados tenha pelo menos 2 linhas, a primeira será usada como cabeçalho, caso contrário uma exceção será liberada na linha 4. Também é determinado que a última coluna será a classificação.

4.2 Calculando entropia e ganho

Uma instância de `Table` depois de criada pode calcular a própria entropia, calcular o ganho de um atributo ou criar uma subtabela. O cálculo da entropia é feito assim que a tabela é criada através do método privado `calc_entropy` como mostrado na listagem 4.2.

Listagem 4.2: Cálculo da entropia

[1] Apesar do nome do formato ser "Valores separados por **vírgula**", o Microsoft Excel salva os valores separados por ponto-vírgula, ignorando a RFC 4180.

```
1     # log na base 2
2     def log2(val)
3         return val == 0 ? 0 : Math.log(val)/Math.log(2)
4     end
5
6     # calcula a entropia da tabela
7     def calc_entropy
8         n = @header.length - 1
9         results = {}
10        @data.each do |row|
11            if results.include? row[n]
12                results[row[n]] += 1
13            else
14                results[row[n]] = 1
15            end
16        end
17        entropy = 0.0;
18        results.each do |k,v|
19            prob = v/@data.length.to_f
20            entropy -= prob*log2(prob)
21        end
22        return entropy
23    end
```

Atributos de classe em Ruby começam com uma arroba, na classe `Table` o atributo `@header` guarda uma array com o cabeçalho da base de dados, o atributo `@data` guarda uma array de arrays, que é o conteúdo da tabela. Para calcular a entropia contamos antes todos os possíveis classificações que existem no conjunto de dados, linha 8 à 16, e guardamos em um hash onde a chave é o valor da classificação e o valor é número de ocorrências na base de dados.

Com isso em mãos calcumamos a entropia, linha 17 à 21, usando a equação 3.1.

Para criar um subconjunto da base de dados a classe `Table` provê o método `subset` que recebe 2 parâmetros, o índice do atributo utilizado e o valor que se quer no subconjunto. O índice do atributo é o número da coluna onde ele se encontra na base de dados, começando de zero.

Listagem 4.3: Geração do subconjunto de uma tabela

```
1     def subset(attr_idx, attr_value)
2         sub_data = @data.select do |row|
3             row[attr_idx] == attr_value
4         end
5         return Table.new(@header, sub_data, @values)
6     end
```

A última coisa que a classe `Table` pode fazer é calcular o ganho de um atributo, isto é feito pelo método `gain` que recebe o índice do atributo no qual se quer saber o ganho.

Listagem 4.4: Cálculo do ganho

```
1   def gain(attr_idx)
2       attr_name = @header[attr_idx]
3
4       gain = entropy
5       @values[attr_name].each do |attr_value|
6           sub_table = subset(attr_idx, attr_value)
7           gain -= (sub_table.num_rows/num_rows.to_f)*sub_table.entropy
8       end
9       return gain
10  end
```

4.3 Gerando a árvore de decisão

A classe `Table` e apenas uma classe utilitária usada no processo de criação da árvore de decisão, após a árvore criada todas as instâncias de `Table` podem ser descartadas.

A árvore de decisão em si é gerada pela classe `DecisionTree`, seu constructor recebe o nome de um arquivo CSV, que chama o método `expand_node` que por sua vez recebe 4 argumentos: o nome do arco que sairá do nó atual, o nó atual, a base de dados e a coleção de atributos que ainda podem ser usados em futuros nós.

Listagem 4.5: Expansão dos nós de uma árvore de decisão

```
1       return if table.empty?
2
3       # Se a entropia da tabela é zero é por que temos um nó folha
4       if table.entropy.zero?
5           value = table.value_of(0, table.header.length-1)
6           node = DecisionNode.new(value)
7           parent.append_child(arc_name, node)
8           return
9       end
10
11      # calcula todos os ganhos para saber quem é o próx. nó
12      gain = {}
13      attrs.each do |attr_idx|
14          gain[attr_idx] = table.gain(attr_idx)
15      end
16      # Ordena os ganhos para saber qual o melhor
17      best_attr_idx, best_gain = gain.sort {|x, y| x[1] <=> y[1]}.pop
18      best_attr_name = table.header[best_attr_idx]
19      # depois que escolheu tira ele da tabela
20      attrs.delete(best_attr_idx)
21      # cria o nó da árvore
22      node = DecisionNode.new(best_attr_name)
23      if parent.nil?
```

```
24              @root = node
25          else
26              parent.append_child(arc_name, node)
27          end
28
29          # Tenta criar mais nós filhos
30          table.values[best_attr_name].each do |value|
31                  subtable = table.subset(best_attr_idx, value)
32                  expand_node(value, node, subtable, attrs)
33          end
34          attrs << best_attr_idx
35      end
36  end
```

O algoritmo da listagem 4.5 é recursivo com duas condições de parada, quando o conjunto recebido é vazio (linha 3) e quando a entropia do conjunto é 0 (linhas 5-11), neste caso ele cria um nó folha na árvore antes de retornar.

Nas linhas 14 à 17 ele calcula o ganho de todos os atributos que restam para ser usados na árvore, guardados na variável `attrs`, e pega o maior para ser usado no próximo nó, caso a árvore ainda não possua uma raiz o nó recém criado será a raiz (linha 26), caso contrário ele será adicionado como filho do nó atual (linha 28).

Após isto, nas linhas 31-35, tenta-se expandir o nó recém criado fazendo uma chamada recursiva para cada possível valor do nó.

4.4 Representação textual da árvore

A representação textual da árvore será impressa na saída padrão do programa de forma tabulada para que seja possível visualizar a relação pai/filho entre os nós. Segue abaixo o exemplo de uma chamada do programa que imprime a árvore de decisão na saída padrão.

```
$ ./main.rb dumptree data/tenis.csv
Aspecto
Sol=>Umidade
          Normal=>Sim
          Elevada=>Não
Nuvens=>Sim
Chuva=>Vento
          Fraco=>Sim
          Forte=>Umidade
                    Normal=>Temperatura
                              Ameno=>Sim
                              Fresco=>Não
                    Elevada=>Não
```

4.5 Representação gráfica da árvore

A geração da representação gráfica da árvore é feita com o auxílio do Graphviz [2].

Graphviz é um pacote que contém vários programas relacionados a grafos, utilizaremos um desses programas, o `dot`, que é especializado em gerar layouts para grafos. Ele funciona da seguinte forma, você passa para ele um script descrevendo o grafo, qual nó esta conectado a qual nó, qual o nome de cada nó ou arco, formato do nó (elipse, circulo, quadrado, etc) e ele gera uma imagem do grafo, esta imagem pode ser em vários formatos, SVG, PNG, PS, etc, aqui estamos usando o formato PNG. Logo tudo o que nosso programa faz é gerar este script descrevendo o grafo (árvore de decisão em questão) e mandar o `dot` criar uma imagem no formato PNG com a árvore. Um exemplo de saída ja foi mostrado na figura 3.1.

4.6 Execução da árvore

Cada nó da árvore de decisão guarda o nome do atributo e um hash[2] onde a chave é o valor do nó e o valor é o nó filho.

Sendo assim começamos perguntando um valor para o atributo do nó raiz, dada a resposta consultamos no hash qual a próxima pergunta (nó), quando chegarmos em um nó sem filhos chegamos em uma resposta.

Listagem 4.6: Algoritmo de execução da árvore de decisão

```
1  def execute(csv_file)
2      tree = DecisionTree.new(csv_file)
3
4      puts "Pergunta que não quer calar: \"#{tree.question}?\""
5
6      node = tree.root
7      answer = "Não sei!!!!"
8
9      while not node.nil?
10         if node.children.empty?
11             answer = node.name
12             break
13         end
14         puts "#{node.name}? [#{node.children.keys.join('|')}]"
15         user_answer = $stdin.gets
16         node = node.children[user_answer.strip]
17     end
18     puts "Resposta: #{answer}"
19 end
```

[2]Em Ruby hash são arrays que podem ser indexadas por quaisquer outros objetos, neste caso uma String

4.7 Teste da árvore

Testa uma árvore de decisão é a tarefa mais importante, pois com testes é que se pode saber se sua base de treinamento é ou não uma boa amostra da população, quanto melhor a amostra melhor será a probabilidade de acertos da árvore.

O teste é feito usando uma base de dados para responder as perguntas de cada nó da árvore anotando os erros e acertos para que se possa visualizar as respectivas probabilidades ao término dos testes.

Listagem 4.7: Algoritmo de teste da árvore de decisão

```
1  def test_against(csv_file)
2      resultados_ok = 0
3
4      table = Table.fromCSV(csv_file)
5      table.data.each do |row|
6          node = @root
7          while node.children?
8              idx = table.header.rindex(node.name)
9              node = idx.nil? ? nil : node.children[row[idx]]
10             break if node.nil?
11         end
12         resultados_ok += 1 if not node.nil? and node.name == row[table.header.
               length - 1]
13     end
14     return resultados_ok, table.data.length
15 end
```

Seguindo a listagem 4.7, na linha 5 a base de testes é carregada e depois é feito uma iteração sobre todos os items da base de testes, em cada iteração temos que caminhas na árvore (linhas 7-12) utilizando os dados da base de testes. Ao final o contador de sucessos é incrementado o mesmo ocorrer. Este método retorna 2 valores, o total de sucessos e o número de casos testados.

5 Resultados obtidos

Os resultados obtidos por uma árvore de decisão estão intimamente ligados a qualidade da amostra utilizada no treinamento, base de dados pequenas, sem representação estatística e com fins único e exclusivamente didáticos como as utilizadas em nossos testes não nos permite esboçar a eficiência do algoritmo ID3 em predizer a classe de elementos que não estão em sua base de treinamento, sendo esta uma base formada por uma amostra que representa bem a população em questão.

6 Considerações Finais

Neste projeto podemos compreender bem o funcionamento de um dos modelos mais práticos e usados em inferência indutiva. Utilizadas no processo de aprendizagem para agentes usados na inteligência artificial as Árvores de Decisão são treinadas de acordo com um conjunto de treino (exemplos previamente classificados) e posteriormente, outros exemplos são classificados de acordo com essa mesma árvore havendo uma generalização. Espero que nosso projeto possa ajudar professores a diversificar seus exemplos em sala de aula e apresentar a sua turma uma implementação prática desta técnica.

Anexo: Código fonte

Listagem 6.1: main.rb

```
#!/usr/bin/env ruby

require 'csv'

def show_help
    puts """Para gerar o grafo da árvore de decisão use:
  #{$0} creategraph CSV_FILE PNG_OUTPUT

Para imprimir a árvore de decisão na tela use:
  #{$0} dumptree CSV_FILE

Para executar a árvore de decisão use:
  #{$0} execute CSV_FILE

Para testar a árvore de decisão contra uma base de dados use:
  #{$0} test CSV_FILE CSV_TEST_FILE

Onde:
  CSV_FILE é um arquivo no formato CSV que será usado para criar a árvore de decisão
      .
  PNG_OUTPUT é o nome do arquivo PNG que será gerado após a execução do programa.
  CSV_TEST_FILE é um arquivo no formato CSV que será usado como base de testes para
      a árvore de decisão
"""
    exit 1
end

def create_graph(csv_file, png_file)
    tree = DecisionTree.new(csv_file)
    proc = open("|dot -Tpng -o#{png_file}", "wb")
    proc.write(tree.dot_script)
    proc.close
end

def show_tree(csv_file)
    DecisionTree.new(csv_file).root.dump_tree
end

```

```
37 def execute(csv_file)
38     tree = DecisionTree.new(csv_file)
39
40     puts "Pergunta que não quer calar: \"#{tree.question}?\""
41
42     node = tree.root
43     answer = "Não sei!!!!"
44
45     while not node.nil?
46         if node.children.empty?
47             answer = node.name
48             break
49         end
50         puts "#{node.name}? [#{node.children.keys.join('|')}]"
51         user_answer = $stdin.gets
52         node = node.children[user_answer.strip]
53     end
54     puts "Resposta: #{answer}"
55 end
56
57 def test_tree(csv_data_file, csv_test_file)
58     tree = DecisionTree.new(csv_data_file)
59     ok, total = tree.test_against(csv_test_file)
60
61     puts """Número de testes: #{total}
62 Acertos: #{ok*100.0/total}% (#{ok})
63 Erros: #{(total-ok)*100.0/total}% (#{total-ok})
64 """
65 end
66
67 if $0 == __FILE__
68     Dir.chdir(File.dirname(__FILE__))
69     require 'decisiontree'
70
71     show_help if ARGV.length < 2
72
73     begin
74         case ARGV[0]
75             when "creategraph"
76                 show_help if ARGV.length < 3
77                 create_graph(ARGV[1], ARGV[2])
78             when "dumptree"
79                 show_tree(ARGV[1])
80             when "execute"
81                 execute(ARGV[1])
82             when "test"
83                 show_help if ARGV.length < 3
```

```
84                  test_tree(ARGV[1], ARGV[2])
85              else
86                  show_help
87          end
88      rescue
89          puts "Erro: #{$!}"
90      end
91  end
```

Listagem 6.2: table.rb

```
1   require 'csv'
2
3   # Uma tabela =]
4   class Table
5       # Cabecalho da tabela
6       attr_reader :header
7       # Hash com valores que cada atributo pode ter, ex.: { "attr_name" => [valor1,
            valor2], ...}
8       attr_reader :values
9       # Conteúdo da tabela
10      attr_reader :data
11      # Entropia da tabela
12      attr_reader :entropy
13
14      # Constructor, recebe o cabeçalho da tabela e os dados, opcionalmente pode
15      # receber os possiveis valores de cada atributo, caso não seja dado a tabela
16      # será varrida e eles serão calculados
17      def initialize(header, data, values = nil)
18          @header = header
19          @data = data
20
21          if values.nil?
22              # Verifica os possiveis valores de cada atributo
23              # attr_name => value_set
24              @values = {}
25              @header.each_with_index do |attr, attr_idx|
26                  @values[attr] = Set.new
27                  @data.each do |row|
28                      @values[attr] << row[attr_idx]
29                  end
30              end
31          else
32              @values = values
33          end
34          # calcula entropia
35          @entropy = calc_entropy
36      end
```

```
37
38      # Cria uma tabela a partir de um arquivo CSV
39      def self.fromCSV(csv_file)
40          data = CSV.open(csv_file, 'r').entries
41          raise "Dados insuficientes" if data.length < 2
42          header = data.shift
43          return Table.new(header, data)
44      end
45
46      def num_rows
47          @data.length
48      end
49
50      def empty?
51          @data.empty?
52      end
53
54      def value_of(row, col)
55          @data[row][col]
56      end
57
58      # calcula o ganho sobre o atributo com id attr_idx
59      def gain(attr_idx)
60          attr_name = @header[attr_idx]
61
62          gain = entropy
63          @values[attr_name].each do |attr_value|
64              sub_table = subset(attr_idx, attr_value)
65              gain -= (sub_table.num_rows/num_rows.to_f)*sub_table.entropy
66          end
67          return gain
68      end
69
70      # Retorna uma tabela com o sub conjunto dos dados onde todo mundo tem o valor
            value no atributo attr
71      def subset(attr_idx, attr_value)
72          sub_data = @data.select do |row|
73              row[attr_idx] == attr_value
74          end
75          return Table.new(@header, sub_data, @values)
76      end
77
78  private
79      # log na base 2
80      def log2(val)
81          return val == 0 ? 0 : Math.log(val)/Math.log(2)
82      end
```

```
83
84      # calcula a entropia da tabela
85      def calc_entropy
86          n = @header.length - 1
87          results = {}
88          @data.each do |row|
89              if results.include? row[n]
90                  results[row[n]] += 1
91              else
92                  results[row[n]] = 1
93              end
94          end
95          entropy = 0.0;
96          results.each do |k,v|
97              prob = v/@data.length.to_f
98              entropy -= prob*log2(prob)
99          end
100         return entropy
101     end
102 end
```

Listagem 6.3: decisiontree.rb

```
1  require 'set'
2  require 'table'
3
4  # Um nó da árvore de decisão
5  class DecisionNode
6      attr_reader :name
7      attr_reader :id
8      attr_reader :children
9
10     @@nextid = 0
11
12     def initialize(name)
13         @name = name
14         @children = {}
15         @id = @@nextid
16         @@nextid += 1
17     end
18
19     # Adiciona um filho à árvore
20     # arc_name - o nome do arco
21     # node - o nó filho
22     def append_child(arc_name, node)
23         @children[arc_name] = node
24     end
25
```

```
26      def children?
27          not @children.empty?
28      end
29
30      # Imprime a árvore em stdout
31      def dump_tree
32          print_tree(self, 0)
33      end
34
35      # Retorna um script (dot) descrevendo o grafo formado por este nó e seus filhos.
36      def dot_script
37          buffer = String.new
38          buffer << "digraph G {\n";
39          buffer << "node [shape=box, style=rounded, fontsize=12, height=0.2]\n";
40          buffer << "edge [fontsize=12]\n"
41          generate_dot_stript(self, buffer)
42          buffer << "}\n"
43          return buffer
44      end
45
46  private
47      def generate_dot_stript(node, buffer)
48          style = node.children.empty? ? ', color=red' : ''
49          buffer << "node#{node.id} [label=\"#{node.name}\" #{style}]\n"
50          node.children.each do |arc_name, child|
51              generate_dot_stript(child, buffer)
52              buffer << "node#{node.id} -> node#{child.id} [label=#{arc_name}];\n"
53          end
54      end
55
56      def print_tree(node, deep)
57          puts node.name
58          node.children.each do |arc_name, child|
59              (deep*10).times{ putc(' ') }
60              print arc_name, '=>'
61              print_tree(child, deep+1)
62          end
63      end
64  end
65
66  # Árvore de decisão
67  class DecisionTree
68      # Raiz da árvore
69      attr_reader :root
70      attr_reader :question
71
72      # ctor, receve o nome de um arquivo no formato CSV
```

```
73      def initialize(csv_file)
74          table = Table.fromCSV(csv_file)
75
76          @question = table.header.last
77          attrs = Set.new((0..(table.header.length-2)).to_a);
78          raise "Dados viciados" if table.entropy.zero?
79          expand_node(nil, nil, table, attrs)
80      end
81
82      def dot_script
83          @root.dot_script
84      end
85
86      def test_against(csv_file)
87          resultados_ok = 0
88
89          table = Table.fromCSV(csv_file)
90          table.data.each do |row|
91              node = @root
92              while node.children?
93                  idx = table.header.rindex(node.name)
94                  node = idx.nil? ? nil : node.children[row[idx]]
95                  break if node.nil?
96              end
97              resultados_ok += 1 if not node.nil? and node.name == row[table.header.
                    length - 1]
98          end
99          return resultados_ok, table.data.length
100     end
101
102 private
103     # expande os nós da árvore até a morte
104     def expand_node(arc_name, parent, table, attrs)
105         return if table.empty?
106
107         # Se a entropia da tabela é zero é por que temos um nó folha
108         if table.entropy.zero?
109             value = table.value_of(0, table.header.length-1)
110             node = DecisionNode.new(value)
111             parent.append_child(arc_name, node)
112             return
113         end
114
115         # calcula todos os ganhos para saber quem é o próx. nó
116         gain = {}
117         attrs.each do |attr_idx|
118             gain[attr_idx] = table.gain(attr_idx)
```

```
119             end
120             # Ordena os ganhos para saber qual o melhor
121             best_attr_idx, best_gain = gain.sort {|x, y| x[1] <=> y[1]}.pop
122             best_attr_name = table.header[best_attr_idx]
123             # depois que escolheu tira ele da tabela
124             attrs.delete(best_attr_idx)
125             # cria o nó da árvore
126             node = DecisionNode.new(best_attr_name)
127             if parent.nil?
128                 @root = node
129             else
130                 parent.append_child(arc_name, node)
131             end
132
133             # Tenta criar mais nós filhos
134             table.values[best_attr_name].each do |value|
135                     subtable = table.subset(best_attr_idx, value)
136                     expand_node(value, node, subtable, attrs)
137             end
138             attrs << best_attr_idx
139         end
140     end
```

Referências Bibliográficas

[1] COMMA separated values. 2007. Disponível em: <http://en.wikipedia.org/wiki/Comma-separated_values>.

[2] GRAPHVIZ. 2007. Disponível em: <http://www.graphviz.org/>.

[3] THE ID3 Algorithm. 2007. Disponível em: <http://www.cise.ufl.edu/ ddd/cap6635/Fall-97/Short-papers/2.htm>.

[4] RUBY Programming Language. 2007. Disponível em: <http://www.ruby-lang.org/>.