19 - Machine Learning in Lisp

Objetivos: ID3 algorithm and inducing decision trees from lists of examples.
Uma implementação básica do ID3 demonstrada em um exemplo simples
de de avaliação de crédito.

Conteúdo: 19.1 Learning: O algorítimo ID3
19.2 Implementando o ID3

19.1 - Learning: The ID3 Algorithm

Nesta sessão implementaremos o algoritmo de indução ID3 descrito em Luger (2009, Section 10.3). ID3 infere árvores de decisão a partir de um conjunto de exemplo para treinamento, na qual permite a classificação de um objeto na base de suas propriedades. Cada nó interno da arvore de decisão testa uma das propriedades do objeto candidato, e usa o valor resultante para selecionar um ramo da arvore. Assim continua atraves dos nós da arvore testando várias propriedades, até que chegue a uma folha, onde cada no da folha denota uma classificação. O ID3 usa uma função de informações de seleção teste teórica para pedir testes, a fim de construir uma (quase) árvore de decisão optimizada.
Veja a Tabela 19.1 para uma amostra de um conjunto de dados e a Figura 19.1 para uma árvore induzida ID3. Os detalhes dos algoritmos de indução árvore pode ser encontrado em Luger (2009,Section 10.3) e em Quinlan (1986).

Um histórico de crédito:

Neste capítulo demonstraremos a implementação do ID3 usando um simples exemplo de avaliação de crédito. Suponha que queremos determinar crédito de uma pessoa risco (alto, moderado, baixo) com base em dados gravados a partir de empréstimos passados. Nós podemos representar isso em uma árvore de decisão, onde cada nó examina um aspecto do perfil de crédito de uma pessoa. Por exemplo, se um dos fatores que nos interessa é colateral, em seguida o nó colateral terá dois ramos: não colateral e colateral adequado.

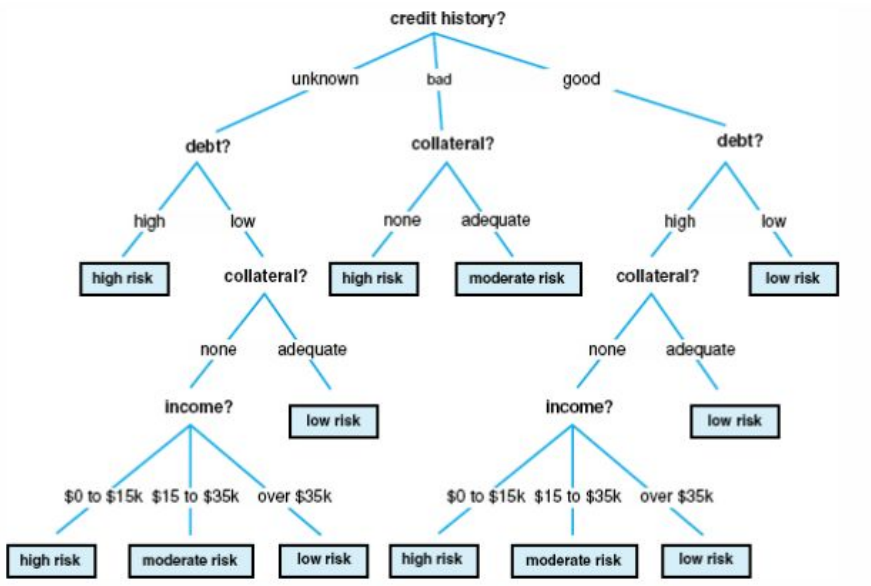
O desafio que algoritmo de aprendizado de máquina enfrenta é o de construir a "melhor" árvore de decisão dado um conjunto de exemplos de treino. Conjuntos de treinamento exaustivos são raros em machine learning, ou porque os dados não estão disponíveis, ou porque tais conjuntos seria grande demais para gerir eficazmente. ID3 constrói árvores de decisão sob a suposição de que a árvore mais simples que classifica corretamente todas as instâncias de formação é mais provável que seja correto em novas instâncias, desde que faz a menor suposição de que os dados de treinamento. ID3 infere uma árvore simples de dados de treinamento usando um greedy algorithm: selecione a propriedade de texto que dá mais informações sobre o conjunto de treinamento, particiona o problema nesta propriedade e se repete. A aplicação que apresentamos ilustra este algoritmo.

Vamos testar o nosso algoritmo sobre os dados de table 19.1.

No.	Risk	Credit History	Debt	Collateral	Income
1.	high	bad	high	none	$0 to $15k
2.	high	unknown	high	none	$15k to $35k
3.	moderate	unknown	low	none	$15k to $35k
4.	high	unknown	low	none	$0 to $15k
5.	low	unknown	low	none	over $35k
6.	low	unknown	low	adequate	over $35k
7.	high	bad	low	none	$0 to $15k
8.	moderate	bad	low	adequate	over $35k
9.	low	good	low	none	over $35k
10.	low	good	high	adequate	over $35k
11.	high	good	high	none	$0 to $15k
12.	moderate	good	high	none	$15k to $35k
13.	low	good	high	none	over $35k
14.	high	bad	high	none	$15k to $35k

Table 19.1 Training data for the credit example

Figure 19.1 mostra uma árvore de decisão que classifica corretamente esses dados. **defstruct** nos permite criar estruturas e itens de dados em Lisp. Por exemplo, usando **defstruct**, nós podemos definir um novo tipo de dados, employee, por avaliando um formulário. *employee* é o nome do tipo definido; *name*, *address*, *serial-number*, *department*, *salary*.



Definindo estruturas usando Defstruct

```
(defstruct employee
        name
        address
        serial-number
        department
        salary)
```

Aqui *defstruct* obtém como argumento um símbolo, no qual será o nome do novo tipo, e o numero de campos especificadores. Aqui nós definimos cinco campos por nome; slot specifiers que também nos permite definir diferentes propriedades de campos, incluindo tipo e informação de inicialização, veja Steele (1990). A avaliação da forma *defstruct* tem o número de efeitos, por exemplo:

```
(defstruct <type name>
    <slot name 1>
    <slot name 2>
       …
    <slot name n>)
```

*
** pulando o blablabla sobre o que eh o defscruct
*
Assim, vemos que o uso de estruturas, podemos definir predicados e avaliadores de um tipo de dados em uma única forma Lisp. Essas definições são fundamentais para a nossa implementação do algoritmo ID3.
Quando dado um conjunto de exemplos de classificações conhecidas, nós usamos a informação da amostra oferecidos em Table 19.1, ID3 induz uma árvore que irá classificar corretamente todas as instâncias de formação, e tem uma alta probabilidade de corretamente classificar novas pessoas que solicitam crédito, veja a Figura 19.1.
Na discussão de ID3 em Luger (2009, Seção 10.3), as instâncias de formação são oferecido em forma de tabela, listando explicitamente as propriedades e seus valores para cada instância. Assim, a Tabela 19.1 lista um conjunto de instâncias para aprender a prever o risco de crédito de um indivíduo. Em toda esta seção, continuaremos para se referir a este conjunto de dados.
abelas são apenas uma forma de representar exemplos; É a forma mais geram de considerá-los como objetos que podem ser testados para várias propriedades. Nossa implementação faz algumas suposições sobre a representação de objetos. Para cada propriedade, requer uma função de um argumento de que pode ser aplicado a um objecto para devolver um valor de propriedade que. Por exemplo, se credit-profile-1 está ligado com o primeiro exemplo na Tabela 19.1,e a história é uma função que retorna o valor de histórico de crédito de um objeto, então:

```
> (history credit-profile-1)
bad
```

Da mesma forma, precisamos de funções para as outras propriedades de um perfil de crédito:

```
> (debt credit-profile-1)
high

> (collateral credit-profile-1)
 none

 > (income credit-profile-1)
0-to-15k

 > (risk credit-profile-1)
high
```

Em seguida, selecionamos uma representação para o exemplo de cessão de crédito,fazer objetos como listas associativas em que as chaves são nomes de propriedades e seus dados são valores de propriedade. Assim, o primeiro exemplo da Tabela 19.1 é representado pela seguinte association list:

```
(defparameter *examples*
  '(((risk . high) (history . bad) (debt . high) (collateral . none) (income . 0-to-15k))
    ((risk . high) (history . unknown) (debt . high) (collateral . none) (income . 15k-to-35k))
    ((risk . moderate)(history . unknown) (debt . low) (collateral . none) (income . 15k-to-35k))
    ((risk . high)(history . unknown) (debt . low) (collateral . none) (income . 0-to-15k))
    ((risk . low)(history . unknown) (debt . low) (collateral . none) (income . over-35k))
    ((risk . low)(history . unknown) (debt . low) (collateral . adequate) (income . over-35k))
    ((risk . high)(history . bad) (debt . low) (collateral . none) (income . 0-to-15k))
    ((risk . moderate)(history . bad) (debt . low) (collateral . adequate) (income . over-35k))
    ((risk . low)(history . good) (debt . low) (collateral . none) (income . over-35k))
    ((risk . low)(history . good) (debt . high) (collateral . adequate) (income . over-35k))
    ((risk . high)(history . good) (debt . high) (collateral . none) (income . 0-to-15k))
    ((risk . moderate)(history . good) (debt . high) (collateral . none) (income . 15k-to-35k))
    ((risk . low)(history . good) (debt . high) (collateral . none) (income . over-35k))
    ((risk . high)(history . bad) (debt . high) (collateral . none) (income . 15k-to-35k))))
```

Desde o fim de uma árvore de decisão é a determinação de *risk* para um novo individuo, *test-instance* incluirão todas as propriedades, exceto risco:

```
(defparameter *test-instance*
  '((history . good) (debt . low)
    (collateral . none) (income . 15k-35k)))
```

Dada esta representação de *objects*, a seguir definimos *property*:

```
(defun history (object)
  (rest (assoc 'history object :test #'equal)))

(defun debt (object)
  (rest (assoc 'debt object :test #'equal)))

(defun collateral (object)
  (rest (assoc 'collateral object :test #'equal)))

(defun income (object)
  (rest (assoc 'income object :test #'equal)))

(defun risk (object)
  (rest (assoc 'risk object :test #'equal)))
```

Uma *property* é uma função em *objects*; que representam estas funções como um slot em uma estrutura que inclui outras informações úteis:

```
(defstruct property
  name
  test
  values)
```

O slot de teste de uma instância da propriedade está vinculado a uma função que retorna um valor de propriedade. ***Name*** é o nome da propriedade, e está incluído apenas para ajudar o usuário inspecionar definições. *values* é uma lista de todos os valores que podem ser retornados por *test*. A exigência de que os valores de cada propriedade ser conhecido antecipadamente simplifica a implementação muito, e não é razoável.

```
(defstruct decision-tree
  test-name
  test
  branches)

(defstruct leaf
  value)
```

*

** pulando o blabla do test

*

Apesar de um conjunto de exemplos de treino é, conceitualmente, apenas uma coleção de objetos, vamos torná-lo parte de uma estrutura que inclui slots para outras informações utilizadas pelo algoritmo. Nós definimos example-frame como :

```
(defstruct example-frame
 instances
 properties
 classifier
 size
 information)
```

instances é uma lista de objetos de classificação conhecida; este é o conjunto de treinamento usado para construir uma árvore de decisão. *properties* é uma lista de objetos do tipo de propriedade; estas são as *properties* que pode ser utilizado em que os nós de árvore. *classifier* é também um exemplo de *property*; que representa a classificação ID3 que está a tentar aprender. Uma vez que os exemplos são de classificação conhecido, nós o incluimos como uma outra propriedade. *size* é o número de exemplos em slot; *information* é o conteúdo da informação do conjunto de exemplos. Nós computamos o conteudo de *size* e *information* a partir de exemplos. Uma vez que estes valores levam tempo para calcular e será usado várias vezes, nós salvá-los nesses slots. Uma vez que estes valores levam tempo para calcular e será usado várias vezes, nós o salvamos nesses slots.

ID3 constrói árvores de forma recursiva. Dado um conjunto de exemplos, cada uma instância de exemplo-frame, ele seleciona uma propriedade e usa-lo para particionar o conjunto de instâncias de formação em subconjuntos não-interseção. Cada subconjunto contém todas as instâncias que têm o mesmo valor para essa propriedade.

A propriedade selecionada torna-se o teste no nó actual da árvore para cada subconjunto na partição, ID3 recursivamente constrói uma sub-árvore usando as propriedades restantes. O algoritmo pára quando um conjunto de exemplos todos pertencem à mesma classe, ao ponto em que ele cria uma folha.

Nossa definição de estrutura final é *partition*, uma divisão de um exemplo dado em subproblemas usando uma propriedade particular. Seguindo a definição de *partition:*

```
(defstruct partition
    test-name
    test
    components
    info-gain)
```

Em um exemplo de *partition*, *test* slot está ligado à propriedade usada para criar *partition*. *test-name* é o nome de *test*, incluído para facilitar a leitura. *components* ficará vinculado aos subproblemas de *partition*. Na nossa implementação, componentes é uma lista de associação: as chaves são os diferentes valores do teste selecionados; cada dado é uma instância de *example-frame*. *info-gain* é o ganho de informação que resulta da utilização de

teste como o nó da árvore. Como com *size* e *information* na estrutura *example-frame*, esse slot armazena um valor que é dispendioso para calcular e é utilizado várias vezes no algoritmo. Ao organizar o nosso programa em torno desses tipos de dados, nós fazemos a nossa implementação refletir mais claramente a estrutura do algoritmo.

19.2 - Implementing ID3

O coração da nossa função é a função build-tree, que tem uma instância de exemplo a quadro, e de forma recursiva constroi uma árvore de decisão.

```
(defun build-tree (training-frame)
  (cond
                                        ;Case 1: empty example set.
    ((null (example-frame-instances training-frame))
     (make-leaf :value "unable to classify: no examples"))
                                        ;Case 2: all tests have been used.
    ((null (example-frame-properties training-frame))
     (make-leaf :value (list-classes training-frame)))
                                        ;Case 3: all examples in same class.
    ((zerop (example-frame-information training-frame))
     (make-leaf :value (funcall (property-test (example-frame-classifier training-frame))
                                (first (example-frame-instances       training-frame)))))
                                        ;Case 4: select test and recur.
    (t (let ((part (choose-partition (gen-partitions training-frame))))
         (make-decision-tree
          :test-name (partition-test-name part)
          :test (partition-test part)
          :branches (mapcar #'(lambda (x) (cons (first x)
                                                (build-tree (rest x))))
                            (partition-components part)))))))
```

Usando *cond*, *build-tree* analisa os quatro casos possíveis. No caso 1, o exemplo quadro não contém quaisquer instâncias de formação. Isto pode se for dado ao ID3 um conjunto de treinamento incompleto, sem casos para um determinado valor de uma propriedade. Neste caso é então retornado uma folha consistindo da mensagem: “unable to classify: no examples”.
O segundo caso ocorre se o campo *properties* de *training-frame* é vazio. Em forma recursiva a construção da árvore de decisão, uma vez que o algoritmo seleciona uma propriedade, ele é excluído do slot *properties* no quadro exemplo para todos os subproblemas.Se o exemplo dado é inconsistente,o algoritmo pode esgotar todas as propriedades antes de chegar a uma classificação inequívoca de instâncias de formação. Neste caso, cria-se uma folha cujo valor é uma lista de todas as classes restantes no conjunto de instâncias de formação.
O terceiro caso representa um término bem sucedido de um ramo da árvore.Se *training-frame* tem um teor de zero de informações, em seguida, todos os exemplos pertencem à mesma classe; isso decorre definição de informação de Shannon, veja Luger

(2009, Section 13.3). O algoritmo é interrompido, retornando um nó de folha no qual o valor é igual a esta classe remanescente.

Os três primeiros casos terminam a construção árvore; o quarto caso chama recursivamente *build-tree* para construir as subárvores do nó atual. *gen-partitions* produz uma lista de todas as partições possíveis do exemple-set, usando cada teste no slot de propriedades de *training-frame*. *choose-partition* selecciona o teste que dá o maior ganho de informação.Após a ligação a partição resultante para a parte variável em um bloco *let*, *build-tree* constrói um nó de uma árvore de decisão em que o teste é utilizado no partição escolhida, e o slot *branches* está vinculado a uma lista de associação de sub-árvores. Cada chave em *branches* um valor do teste e cada dado é uma árvore de decisão construído por uma chamada recursiva para *build-tree*. Desde o slot componentes da parte já é uma lista de associação no qual as chaves são valores de propriedade e os dados são instâncias de exemple-frame, vamos implementar a construção de sub-árvores usando *mapcar* para aplicar em *build-tree* para cada dado nesta lista associativa.
gen-partitions recebe um argumento, *training-frame*, um objeto do tipo *example-frame-properties*, e gera todas as partições de suas instâncias. Cada *partition* é criada usando uma diferente propriedade do campo *properties*. *gen-partitions* emprega uma função, *partition*, que leva uma instância de um quadro de exemplo e um exemplo de uma propriedade; it partitions the examples using that property. Note-se a utilização de *mapcar* para gerar uma partição para cada elemento do campo *example-frame-properties* de *training-frame*.

```
(defun gen-partitions (training-frame)
  "Generate all different partitions of an example frame."
  (mapcar #'(lambda (x)
              (partition training-frame x))
          (example-frame-properties training-frame)))
```

choose-partition procura uma lista de partições candidatos e escolhe aquele com o maior ganho de informação:

```
(defun choose-partition (candidates)
  "choose-partition searches a list of candidate partitions and chooses the
one with the highest information gain."
  (cond ((null candidates) nil)
        ((= (length candidates) 1)
         (first candidates))
        (t (let ((best (choose-partition (rest candidates))))
           (if (> (partition-info-gain (first candidates))
                  (partition-info-gain best)) (first candidates) best)))))
```

partition é a função mais complexa na implementação. Ele toma como argumentos um quadro de exemplo e uma *property*, e retorna uma instancia da estrutura partition *partition* structure:

```
(defun partition (root-frame property)
                                        ; Initialize parts to to an a-list of empty example frames
                                        ; indexed by the values of property
 (let ((parts (mapcar #'(lambda (x) (cons x (make-example-frame)))
                      (property-values property))))

                                        ; partition examples on property, placing each example
in the appropriate
                                        ; example frame in parts
  (dolist (instance (example-frame-instances root-frame))
   (push instance (example-frame-instances
           (rest (assoc (funcall (property-test property) instance)
                                    parts)))))
                                        ; complete information in each component of the
partition
  (mapcar #'(lambda (x)
        (let ((frame (rest x)))
         (setf (example-frame-properties frame)
            (remove property (example-frame-properties root-frame)))
         (setf (example-frame-*classifier* frame)
            (example-frame-*classifier* root-frame))
         (setf (example-frame-size frame)
            (list-length (example-frame-instances frame)))
         (setf (example-frame-information frame)
            (compute-information
             (example-frame-instances frame)
             (example-frame-*classifier* root-frame)))))
      parts)
                                        ; return an instance of a partition
  (make-partition
   :test-name (property-name property)
   :test (property-test property)
   :components parts
   :info-gain (compute-info-gain root-frame parts))))
```

partition começa por definir uma variável local, *parts*, usando o bloco *let*. Ele inicializa *parts* para uma lista de associação cujas chaves são os possíveis valores do teste em *property*, e cujos dados serão os subproblems do *partition*. *partition* implementa esta usando o macro *dolist*. *dolist* se liga variáveis locais para cada elemento de uma lista e avalia o seu corpo para cada ligação. Neste ponto, eles são exemplos vazios de *example-frame*: tele ranhuras instância de cada subproblema estão vinculados a *nil*. Usando a forma *dolist*, *partition* envia cada elemeto do campo de instancias de *root-frame* para o slot apropriado em *parts*. *push* é um macrp Lisp que modifica a lista adicionando um novo primeiro elemento; diferente de *cons*, *push* adiciona permanentemente um novo elemento à lista.

Esta seção do código realiza o particionamento real de *root-frame*. Depois que *dolist* termina, *parts* vinculado a uma lista de associação na qual cada chave é um valor de *property* e cada dado é um exemplo cujas instâncias compartilham esse valor. Usando *mapcar*, o algoritimo então conclui a informação necessária de cada quadro no exemplo em *parts*, atribuindo valores apropriados para *properties*, *classifier*, *size* e *information*. Então em seguida constrói as instancias dos campos de *partition*, vinculando pelo campo *components* para *parts*.

list-classes é usado no caso 2 de *build-tree* para criar a folha para uma classificação ambígua. Isto emprega o loop *do* para enumerar as classes nas listas de exemplos. O loop *do* inicializa *classes* para todos os valores do classificador em *training-frame*. Para cada elemento de *classes*, Isso adiciona à *classes-present* se foi possível encontrar um elemento no campo de instâncias de *training-frame* que pertence a essa classe.

```
(defun list-classes (training-frame)
  "Lists all the classes in the instances of a training frame"
                                          ; Eliminate those potential classifications not present
                                          ; in the instances of training frame
  (do ((classes (property-values (example-frame-classifier training-frame))
                (rest classes))
     (*classifier* (property-test (example-frame-classifier training-frame)))
     classes-present)
    ((null classes) classes-present)
   (if (member (first classes) (example-frame-instances training-frame)
          :test #'(lambda (x y) (eql x (funcall *classifier* y))))
          (setf classes-present (cons (first classes) classes-present)))))
```

As funções restantes computam a informação do conteúdo de *examples*. *compute-information* determina o conteúdo da informação da lista *examples*. Ele conta o número de instâncias em cada classe, e calcula a proporção do conjunto total de treinamento pertencentes a cada classe. Assumindo que esta proporção é igual a probabilidade de que um objeto pertence a uma classe, ele calcula o conteúdo de informação de exemplos usando a definição de Shannon:

```
(defun compute-information (examples classifier)
 "Computes the information content of a list of examples using a classifier."
 (let ((class-count
      (mapcar #'(lambda (x) (cons x 0)) (property-values classifier)))
     (size 0))
                                          ; count number of instances in each class
  (dolist (instance examples)
   (incf size)
   (incf (rest (assoc (funcall (property-test classifier) instance)
                      class-count))))
                                          ; compute information content of examples
  (sum #'(lambda (x) (if (zerop (rest x)) 0
               (* -1 (/ (rest x) size)
                 (log (/ (rest x) size) 2))))
     class-count)))
```

compute-info-gain obtém o ganho de informação de uma partição subtraindo a média ponderada da informação nos seus componentes a partir dos seus exemplos mãe:

```
(defun compute-info-gain (root parts)
 (- (example-frame-information root)
   (sum #'(lambda (x) (* (example-frame-information (rest x))
                (/ (example-frame-size (rest x))
                  (example-frame-size root))))
      parts)))
```

sum calcula os valores devolvidos pela aplicação de *f* em todos os elementos de *list-of-numbers*:

```
(defun sum (f list-of-numbers)
 "Sum takes the sum of applying f to all numbers in list-of-numbers."
 (apply '+ (mapcar f list-of-numbers)))
```

Isto conclui a implementação de *build-tree*. O componente restante do algoritmo é a função, *classify*, que toma como argumentos uma árvore de decisão, tal como calculado pela *build-tree*, e um objecto a ser classificado; que determina a classificação do objeto por recursivamente percorrer a árvore. A definição de *classify* é simples: *classify* para quando encontra uma folha, caso contrário, aplica-se o teste a partir do nó atual para *instance*, e utiliza o resultado como a chave para selecionar num ramo em uma chamada para *assoc:*

```
(defun classify (instance tree)
  (if (leaf-p tree)
     (leaf-value tree)
     (classify instance
                    (rest (assoc (funcall (decision-tree-test tree) instance)
                                 (decision-tree-branches tree))))))
```

Usando as definições de objeto já definidos, nós agora chamamos *build-tree* para o exemplo de crédito da Table 19.1. Nós vinculamos esses testes em uma lista de definições de propriedade para *history*, *debt*, *collateral* e *income*. *classifier* e testamos o risco de uma ocorrência. Com base nessas definições que ligam os exemplos de crédito para uma instância de *example-frame:*

```
(setq tests
    (list (make-property
              :name 'history
              :test #'history
              :values '(good bad unknown))
             (make-property
              :name 'debt
              :test #'debt
              :values '(high low))
             (make-property
              :name 'collateral
              :test #'collateral
              :values '(none adequate))
             (make-property
              :name 'income
              :test #'income
              :values
              '(0-to-15k 15k-to-35k over-35k))))



(defparameter *classifier*
  (make-property
   :name 'risk
   :test #'risk
   :values '(high moderate low)))

(defparameter *credit-examples*
  (make-example-frame
   :instances *examples*
   :properties *test-set*
   :classifier classifier
   :size (list-length *examples*)
```

```
  :information (compute-information *examples* *classifier*)))
```
Com base nessas definições, agora podemos induzir árvores de decisão, e usá-los para classificar casos de acordo com o seu risco de crédito:

```
> (SETQ credit-tree (BUILD-TREE *CREDIT-EXAMPLES*))

#S(DECISION-TREE
  :TEST-NAME INCOME
  :TEST #<FUNCTION INCOME {10070B874B}>
  :BRANCHES ((0-TO-15K . #S(LEAF :VALUE HIGH))
         (15K-TO-35K
          . #S(DECISION-TREE
             :TEST-NAME HISTORY
             :TEST #<FUNCTION HISTORY {10070B882B}>
             :BRANCHES ((GOOD . #S(LEAF :VALUE MODERATE))
                    (BAD . #S(LEAF :VALUE HIGH))
                    (UNKNOWN
                     . #S(DECISION-TREE
                        :TEST-NAME DEBT
                        :TEST #<FUNCTION DEBT {10070AFE7B}>
                        :BRANCHES ((HIGH . #S(LEAF :VALUE HIGH))
                               (LOW
                                . #S(LEAF
                                   :VALUE MODERATE)))))))))
         (OVER-35K
          . #S(DECISION-TREE
             :TEST-NAME HISTORY
             :TEST #<FUNCTION HISTORY {10070B882B}>
             :BRANCHES ((GOOD . #S(LEAF :VALUE LOW))
                    (BAD . #S(LEAF :VALUE MODERATE))
                    (UNKNOWN . #S(LEAF :VALUE LOW)))))))

> (CLASSIFY  '((HISTORY . GOOD) (DEBT . LOW) (COLLATERAL . NONE) (INCOME .
15k-to-35k)) credit-tree)

MODERATE
```