Universidad Veracruzana

Inteligencia Artificial

Programación para la Inteligencia Artificial

Actividad 5

Leonardo Flores Torres

23 de diciembre de 2022

- Utilizando los siguientes conjuntos de entrenamiento del repositorio UCL (`https://archive.ics.uci.edu/ml/index.php`):

  - Car evaluation
  - Tic-tac-toe
  - Qualitative bankruptcy

  llevar a cabo las siguientes actividades:

  1. Implementar en Lisp las mejoras al algoritmo ID3 que adoptaron en su implementación en Prolog. [30/100]
  2. Con base en la validación cruzada, implementar una función de clasificación por votación en donde participen los árboles construidos en este proceso. Si se construyen diez árboles, los diez se utilizan para clasificar un nuevo caso cuya clase será la más votada por esos diez árboles ¿Cómo se compara la eficiencia de este enfoque contra el mejor árbol encontrado? [50/100]
  3. Implementar una función que traduzca un árbol de decisión dado a un programa en Prolog equivalente, probar el clasificador en ese lenguaje. [20/100]

     Solución:

     Para poder utilizar el progama escrito en `sbcl` del algoritmo ID3 fue necesario modificar un par de archivos. Primero se agregó el archivo de `cl-id3-cross-validation` en el archivo para la definición del sistema `cl-id3.asd` para poder acceder a las funciones escritas ahí que tienen que ver con la validación cruzada:

```
(asdf:defsystem :cl-id3
    :depends-on (:split-sequence)
    :components ((:file "cl-id3-package")
        (:file "cl-id3-algorithm"
            :depends-on ("cl-id3-package"))
        (:file "cl-id3-load"
            :depends-on ("cl-id3-package"
                    "cl-id3-algorithm"))
        (:file "cl-id3-classify"
```

```
            :depends-on ("cl-id3-package"
                    "cl-id3-algorithm"
                    "cl-id3-load"))
        ;;; Making cross validation available
        (:file "cl-id3-cross-validation"
            :depends-on ("cl-id3-package"
                        "cl-id3-algorithm"
                        "cl-id3-load"
                        "cl-id3-classify"))))
```

También se agregó el archivo de validación cruzada en el archivo donde se hace la definición del paquete `cl-id3-package.lisp`:

```
(defpackage :cl-id3
    (:use :cl :split-sequence)
    (:export :load-file
        :induce
        :print-tree
        :classify
        :classify-new-instance
        ;;; Making cross validation available
        :cross-validation))
```

La mejora que se realizó en `prolog` fue tomar en cuenta la información individual para cada atributo y usarla como denominador en la división con la ganancia para así encontrar la razón de ganancia. Se escribió una función equivalente en el archivo `cl-id3-algorithm.lisp` para computar la información individual, como se muestra a continuación:

```
;; Improvement to id3 by considering the information value of an attribute.
(defun split-information (examples attribute)
  "It computes the split-information for an ATTRIBUTE in EXAMPLES"
  (let ((parts (get-partition attribute examples))
    (no-examples (count-if #'atom examples)))
    (-
      (apply #'+
             (mapcar
               #'(lambda(part)
                   (let* ((size-part (count-if #'atom
                                               (cdr part)))
                          (proportion (if (eq size-part 0)
                                          0
                                          (* (/ size-part no-examples)
                                             (log (/ size-part no-examples) 2)))))
                     (* proportion 1)))
               (cdr parts))))))
```

Al haber implementado `split-information` se procedió a modificar la función `best-partition`, en el mismo archivo, para cambiar la manera en como se calcula a la mejor particición, en vez de usar solo la ganancia ahora será usando a la razón de ganancia.

```lisp
(defun best-partition (attributes examples)
    "It computes one of the best partitions induced by ATTRIBUTES over EXAMPLES"
    (let* ((info-gains
        (loop for attrib in attributes collect
            (let ((ig (information-gain examples attrib))
                ;; Improvement using split information to consider the gain ratio
                ;; instead of just information gain.
                (si (split-information examples attrib))
                (p (get-partition attrib examples)))
        (when *trace*
            (format t "Partición inducida por el atributo ~s:~%~s~%"
                attrib p)
            (format t "Razón de ganancia: ~s~%"
                ;; GAIN RATIO
                (/ ig (+ 1 si))))
        (list (/ ig (+ 1 si)) p))))
    (best (cadar (sort info-gains #'(lambda(x y) (> (car x) (car y))))))
    (when *trace* (format t "Best partition: ~s~%----------~%" best))
    best))
```

Además, se arregló el mismo problema que surgió anteriormente en la implementación en `prolog` en que si el contenido de información $\mathrm{IV}(\mathrm{A}_i)$ de un atributo era 0 entonces la razón de ganancia diverge. Se puede observar que se tomó la misma solución, sumar una unidad al contenido de información para cada atributo $\mathrm{IV}'(\mathrm{A}_i) = \mathrm{IV}(\mathrm{A}_i) + 1$.

Ahora que ya se agregaron las modificaciones mencionadas se procederaá a encontrar la mejor partición para las 3 bases de datos. Los conjuntos de entrenamiento son lo suficientemente grandes como para que el poner los resultados en este reporte sea impractico, por lo que se decidió incluir algunos de los resultados en archivos de texto individuales. Para calcular la mejor particición de cualquiera de estas bases de datos primero se tiene que hacer lo suguiente:

```lisp
* (asdf:load-system :cl-id3)
* (in-package :cl-id3)
* (load-file "./path/to/myfile.arff")
* (best-partition (remove *target* *attributes*) *examples*)
```

Los resultados de las mejores particiones iniciales para cada uno de los archivos de las bases de datos se guardó en los siguientes archivos dentro del directorio `/outputfiles`,

- `/outputfiles/bestpartition_car.txt`,
- `/outputfiles/bestpartition_tictactoe.txt`,
- `/outputfiles/bestpartition_bankrupcy.txt`.

También se incluyó la información desglosada para cada particición posible, junto con sus respectivas razones de ganancia en

- `/outputfiles/bestpartition_car_trace.txt`,
- `/outputfiles/bestpartition_tictactoe_trace.txt`,

- `/outputfiles/bestpartition_bankrupcy_trace.txt`.

En realidad, la mejor particición inicial sin haber habilitado el `*trace*` es la que corresponde a la que tiene una razón de ganancia mayor. En los archivos con `*trace*` habilitado se pueden ver las razones de ganancia para cada uno y verificar que sí es el caso.

No es necesario computar las particiones iniciales, pero no está de más ver como es que las razones de ganancia que se calculan para cada suposición en cada base de datos. Se dice que no es necesario porque el algoritmo ID3 implementado en la función `id3` llama de manera recursiva a la función `best-partition`, y la función `induce` realiza la construcción del árbol producido en `id3`. Se utilizará la función `print-tree` para imprimir en consola una represetación del árbol producido al ejecutar `induce`. Los arboles producidos se incluyen en los archivos:

- `/outputfiles/tree_car.txt`,
- `/outputfiles/tree_tictactoe.txt`,
- `/outputfiles/tree_bankrupcy.txt`.

El árbol más pequeño, el cual es razonable incluir directamente, es el de la base de datos del banco. Dicho árbol se muestra a continuación:

```
* (print-tree (induce))
CO
- P -> NB
- N -> B
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - P -> NB
    - A -> NB
NIL
```

Para el segundo punto de esta actividad se modificó la función de validación cruzada `cross-validation` en el archivo `cl-id3-cross-validation.lisp` para no imprimir cada árbol generado dentro del `loop`. En casos de árboles pequeños tiene sentido pensar en ver los árboles que se puedan generar, pero cuando dichos árboles sob muy grandes tienden a ocupar demasiado espacio en terminal lo que personalmente produce más confusión que claridad.

```
;; Optional variable if the usr doesn't want to print the tree
(defun cross-validation (k &optional (print_ t))
  ;; Clean the tree list
  (setq *trees* nil)
  (let* ((long (length *examples*)))
    (loop repeat k do
     (let* ((trainning-data (folding (- long k) long))
```

```
        (test-data (difference trainning-data *examples*))
        (tree (induce trainning-data)))
      ;; Only report the tree if wanted, will do it by default.
      (if print_
        (report tree test-data))
      ;; Save tree
      (push tree *trees*)))))
```

Además, se declaró una variable global `*trees*` para guardar cada árbol producido durante la llamada de la función para tenerlos disponibles después durante el proceso de votación. Los resultados de llamar a la función `cross-validation` en cada una de las bases de datos se guardaron en los siguientes archivos

- `/outputfiles/crossvalidation_car.txt`,
- `/outputfiles/crossvalidation_tictactoe.txt`,
- `/outputfiles/crossvalidation_bankruptcy.txt`.

Se comenzó a realizar la validación cruzada con el archivo de los autos, pero la salida no se incluye aquí sino en un respectivo archivo:

```
* (asdf:load-system :cl-id3)

* (load-file "./databases/car.arff")

* (cross-validation 10)
; output removed
```

La función `let-them-vote` toma un ejemplo como argumento, y clasifica el ejemplo respecto a cada árbol en *trees*, y guarda la clase más votada en una variable global `*main-class*`. Ahora a hacerlos votar, para esto se tomará el siguiente ejemplo `(high med 5more more big high acc)`,

```
* (let-them-vote '(high med 5more more big high acc))

Classification for tree no. 1: ACC

Classification for tree no. 2: ACC

Classification for tree no. 3: ACC

Classification for tree no. 4: ACC

Classification for tree no. 5: ACC

Classification for tree no. 6: ACC

Classification for tree no. 7: ACC

Classification for tree no. 8: ACC
```

```
Classification for tree no. 9: ACC

Classification for tree no. 10: ACC

Class UNACC was detected 0 time(s).

Class ACC was detected 10 time(s).

Class GOOD was detected 0 time(s).

Class VGOOD was detected 0 time(s).

The most voted class was: ACC
NIL
```

Se obtienen 10 clasificaciones (una por árbol generado al llamar a la función de validación cruzada), el conteo de ocurrencias de las clases y la clase más votada, la cual en este caso es `acc`.

Ahora se repetirá lo mismo para la base de datos del banco, tomando como ejemplo `(P N N N N N B)`:

```
* (asdf:load-system :cl-id3)
T

* (in-package :cl-id3)
#<PACKAGE "CL-ID3">

* (load-file "./databases/car.arff")
The ID3 setting has been reset.
Training set initialized after "./databases/car.arff".
NIL

* (load-file "./databases/bankruptcy.arff")
The ID3 setting has been reset.
Training set initialized after "./databases/bankruptcy.arff".
NIL

* (cross-validation 10)
CO
- N -> B
- P -> NB
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - P -> NB
    - A -> NB

```

```
Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- A
    CR
    - P -> NB
    - A -> NB
    - N
        FF
        - N -> B
        - A -> NB
- N -> B
- P -> NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- P -> NB
- N -> B
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - A -> NB
    - P -> NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- N -> B
- P -> NB
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - P -> NB
    - A -> NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- N -> B
- P -> NB
- A
    CR
    - N
```

```
        FF
        - A → NB
        - N → B
    - A → NB
    - P → NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- N → B
- A
    CR
    - N
        FF
        - A → NB
        - N → B
    - P → NB
    - A → NB
- P → NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- A
    CR
    - A → NB
    - P → NB
    - N
        FF
        - A → NB
        - N → B
- N → B
- P → NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- A
    CR
    - N
        FF
        - N → B
        - A → NB
    - P → NB
    - A → NB
- P → NB
- N → B

Instances classified correctly: 10
Instances classified incorrectly: 0

```

```
CO
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - A -> NB
    - P -> NB
- N -> B
- P -> NB

Instances classified correctly: 10
Instances classified incorrectly: 0

CO
- P -> NB
- N -> B
- A
    CR
    - N
        FF
        - A -> NB
        - N -> B
    - A -> NB
    - P -> NB

Instances classified correctly: 10
Instances classified incorrectly: 0

NIL

* (let-them-vote '(P N N N N N B))

Classification for tree no. 1: B

Classification for tree no. 2: B

Classification for tree no. 3: B

Classification for tree no. 4: B

Classification for tree no. 5: B

Classification for tree no. 6: B

Classification for tree no. 7: B

Classification for tree no. 8: B

Classification for tree no. 9: B

Classification for tree no. 10: B

```

```
Class B was detected 10 time(s).

Class NB was detected 0 time(s).

The most voted class was: B
NIL
```

El resultado de la validación cruzada y la votación se muestra solamente para el caso del banco por las mismas razones ya mencionadas anteriormente acerca de la longitud de las salidas en las otras bases de datos.

Finalmente, se repite lo mismo para el caso de la base de datos del juego de gato tomando como ejemplo `(x o o x o x b o x negative)`:

```
* (asdf:load-system :cl-id3)
T

* (in-package :cl-id3)
#<PACKAGE "CL-ID3">

* (load-file "./databases/bankruptcy.arff")
The ID3 setting has been reset.
Training set initialized after "./databases/bankruptcy.arff".
NIL

* (cross-validation 10)
; output removed

* (let-them-vote '(x o o x o x b o x negative))

Classification for tree no. 1: NEGATIVE

Classification for tree no. 2: NEGATIVE

Classification for tree no. 3: NEGATIVE

Classification for tree no. 4: NEGATIVE

Classification for tree no. 5: NEGATIVE

Classification for tree no. 6: NEGATIVE

Classification for tree no. 7: NEGATIVE

Classification for tree no. 8: NEGATIVE

Classification for tree no. 9: NEGATIVE

Classification for tree no. 10: NEGATIVE

Class NEGATIVE was detected 10 time(s).

```

```
Class POSITIVE was detected 0 time(s).

The most voted class was: NEGATIVE
NIL
```

Para terminar la actividad hace falta implementar una función que convierta un árbol, como el mostrado para el caso del banco, a su equivalente en `prolog`. Para esto se añadió un archivo extra al conjunto de archivos que conforman el paquete `:cl-id3`, este archivo fue llamado `cl-id3-prolog-tree.lisp`.

La parte importante para realizar esto es generar un conjunto de ramas

```lisp
;; Convierte un arbol generado con el paquete en su respectivo conjunto de ramas.
(defun tree-to-branches (tree)
  (if (leaf-p tree)
      (list (list tree))
      (mapcan (lambda (node)
                (mapcar (lambda (path)
                          (cons (root tree) path))
                        (tree-to-branches node)))
              (children tree))))

;;; Funcion auxiliar para la conversion de un arbol en lisp a clausulas en
↪  prolog.
(defun prolog-tree-aux (tree filename)
  (let ((branches (tree-to-branches tree)))
    (dribble filename)
    (loop for branch in branches
          do (format t "branch(~a, [" (string-downcase (car (last branch))))
          do (let ((attributes (butlast *attributes*)))
               (dotimes (n (length attributes))
                 (if (member (nth n attributes) branch)
                   (when t
                     (downcase-attr (nth n attributes))
                     (downcase-value (nth (+ (position (nth n attributes) branch
                     ↪  :test #'equal) 1) branch)))
                   (print-empty))
                 (if (eql n (- (length attributes) 1))
                   (print-dot)
                   (print-comma))))
          do (print-newline))
    (dribble)))

;; Convierte un arbol en lisp a prolog. Guarda el arbol en el directorio
↪  /home/usr
;; por defecto.
(defun prolog-tree (ramas &optional (filename "~/prolog_tree.pl"))
  (if (not (probe-file filename))
    (prolog-tree-aux ramas filename)
    (format t "File already exists.")))
```

Por ejemplo, las ramas del árbol que corresponde a la base de datos del banco se vería

como se muestra a continuación:

```
* (asdf:load-system :cl-id3)
T

* (in-package :cl-id3)
#<PACKAGE "CL-ID3">

* (load-file "./databases/bankruptcy.arff")
The ID3 setting has been reset.
Training set initialized after "./databases/bankruptcy.arff".
NIL

* (induce)
(CO (P NB) (N B) (A (CR (N (FF (A NB) (N B))) (P NB) (A NB))))

* (tree-to-branches (induce))
((CO P NB) (CO N B) (CO A CR N FF A NB) (CO A CR N FF N B) (CO A CR P NB)
 (CO A CR A NB))
```

Ya al tener todas las ramas solo es necesario convertir esto a su equivalente en `prolog` usando la función `prolog-tree`:

```
* (prolog-tree (induce) "~/prolog_tree_bank.pl")
branch(nb, [_, _, _, _, co/p, _]).
branch(b, [_, _, _, _, co/n, _]).
branch(nb, [_, _, ff/a, cr/n, co/a, _]).
branch(b, [_, _, ff/n, cr/n, co/a, _]).
branch(nb, [_, _, _, cr/p, co/a, _]).
branch(nb, [_, _, _, cr/a, co/a, _]).
```

No es necesario computar las ramas de antemano ya que no se guardan en ninguna variable global, éstas son computadas internamente en la función auxiliar `prolog-tree-aux` en una variable interna `branches`. La función `prolog-tree` envuelve a su contraparte auxiliar solamente para verificar que no existe ya un archivo en el camino especificado por el `usr` con el mismo nombre, si no existe entonces se procede a guardar el resultado.

Los árboles equivalentes en `prolog` de las otras dos bases de datos se incluyen en la misma carpeta correspondiente a los archivos de salida con los siguientes nombres:

- `/outputfiles/prolog_tree_car.pl`,
- `/outputfiles/prolog_tree_tictactoe.pl`,
- `/outputfiles/prolog_tree_bank.pl`.

Ahora se deben comparar los resultados de la representación del árbol en ambos lenguajes. Cargando el archivo donde se guardó el árbol equivalente de `prolog` de la base de datos del banco en `prolog` y probando con el siguiente ejemplo (correspe a la línea 88 del archivo `bankruptcy.arff` cuyo valor de clase es `nb`),

- `sbcl`: `(a p a p a p)`,
- `prolog`: `[ir/a, mr/p, ff/a, cr/p, co/a, op/p]`.

Después de haber cargado el archivo correspondiente en `sbcl`, el resultado es el siguiente:

```
* (setq *current-tree* (induce))
(CO (P NB) (N B) (A (CR (N (FF (A NB) (N B))) (P NB) (A NB))))

* (classify-new-instance '(a p a p a p) *current-tree*)
NB
```

De manera similar, en `prolog`:

```
?- [prolog_tree_bank].
true.

?- branch(X, [ir/a, mr/p, ff/a, cr/p, co/a, op/p]).
X = nb.
```

Probando lo mismo ahora con la base de datos del juego de gato con el siguiente ejemplo (línea 520 del con valor de clase `positive`)

- `sbcl`: `(b x o o x b o x x)`,
- `prolog`: `[top-left/b, top-middle/x, top-right/o, middle-left/o, middle-middle/x, middle-right/b, bottom-left/o, bottom-middle/x, bottom-right/x]`.

El resultado, después de haber cargado la base de datos del juego de gato en `sbcl`, y asignado el resultado de la inducción a la variable `*current-tree*` se obtiene lo siguiente:

```
* (classify-new-instance '(b x o o x b o x x) *current-tree*)
POSITIVE
```

Mientras que el resultado equivalente en `prolog` es:

```
?- [prolog_tree_tictactoe].
true.

?- branch(X, [top-left/b, top-middle/x, top-right/o, middle-left/o,
↪  middle-middle/x, middle-right/b, bottom-left/o, bottom-middle/x,
↪  bottom-right/x]).
X = positive.
```

Para concluir, se muestra lo equivalente para el archivo de la base de datos de los autos tomando el siguiente ejemplo (línea 1420 del con valor de clase `unacc`):

- `sbcl`: `(low high 2 2 med high)`,
- `prolog`: `[buying/low, maint/high, doors/2, persons/2, lug_boot/med, safety/high]`.

El resultado para este último caso, después de haber cargado la base de datos de los autos en `sbcl`, y asignado el resultado de la inducción a la variable `*current-tree*` se obtiene lo siguiente:

```
* (classify-new-instance '(low high 2 2 med high) *current-tree*)
UNACC
```

Y su equivalente resultado en `prolog`:

```
?- [prolog_tree_car].
true.

?- branch(X, [buying/low, maint/high, doors/2, persons/2, lug_boot/med,
↪  safety/high]).
X = unacc.
```

Referencias

[1] Alejandro Guerra-Hernandez. Programación para la inteligencia artificial. `https://www.uv.mx/personal/aguerra/pia/`, 2022. Visitado: 2022-10-07.

[2] Peter Seibel. *Practical common lisp*. Apress, 2006.

[3] Dheeru Dua and Casey Graff. Uci machine learning repository. `http://archive.ics.uci.edu/ml`, 2017.