Programación para la Inteligencia Artificial
Proyecto Final

David Hernández-Enríquez, Arturo Márquez-Flores,
David Martínez-Galicia, José Alberto López-López
Maestría en Inteligencia Artificial
Universidad Veracruzana
CIIA – Centro de Investigación en Inteligencia Artificial
Sebastián Camacho No 5, Xalapa, Ver., México 91000
https://github.com/arturomf94/pia-mia

8 de Enero del 2019

Introducción

El presente trabajo documenta las principales ideas y resultados del proyecto final dirigido a la materia *"Programación para la inteligencia artificial"* dentro de los posgrados en inteligencia artificial ofertados en la Universidad Veracruzana. El principal objetivo de este ejercicio es implementar una serie de modificaciones y extensiones al paquete *cl-id3* desarrollado por el Dr. Alejando Guerra-Hernández para generar arboles de decisión por medio de una adaptación del algoritmo ID3 en Lisp [GH18] y que se basa en el documento [Qui86].

Particularmente, estas modificaciones se enfocan en tres principales acciones, la primera es incluir un método de evaluación y estimación del *error de predicción* del árbol, la segunda se orienta en clasificar nuevos ejemplos que no se encuentren en el conjunto incial de datos a través de la votación de arboles generados y la última dirige los esfuerzos a la traducción automática del mejor árbol obtenido a un conjunto de cláusulas en Prolog para futuras clasificaciones. De esta forma las siguientes secciones exponen los cambios más relevantes y códigos empleados para las mejoras propuestas en el orden anterior mencionado.

1. Validación cruzada

La validación cruzada es uno de los métodos más populares en la literatura de aprendizaje estadístico [HTF01].En general, se obtienen mejores resultados entre más haya disponibilidad de datos para el entrenamiento debido que consiste en separar la muestra de datos en dos conjuntos, un *conjunto de calibración*

de tamaño $n-k$ y un *conjunto de validación* de tamaño k, el primero es usado para ajustar el modelo mientras el segundo se usa para estimar el error de predicción [Zuc00]. Por otro lado, es importante resaltar el empleo del código ya existente dentro del paquete *cl-id3* propuesto por el Dr. Alejandro Guerra-Hernández [GH18]. Para realizar la primera mejora fue necesaria la creación de las variables *classified-int* y *c-classified-int* que almacenan en listas respectivamete, el número de instacias clasificadas e instancias clasificadas correctamente como consecuencia de k iteraciones.

Las acciones anteriores se fundamentan, en la existencia de una función disponible dentro del archivo **cl-id3-cross-validation.lisp** que ejecuta la validación dado un valor que puede ser ingresado a través de un panel de texto en la interfaz gráfica. De esta forma sólo es necesaria la creación de una función que recupere el valor numérico del panel y se ocupe como argumento. Para así después, calcular la eficiencia del modelo como la suma de instancias correctamente clasificadas sobre el total de instancias clasificadas y ser desplegada en el panel de texto *Average accuracy*. El algorimo es expuesto en el Cuadro 1.

Es relevante mencionar que la activación de esta función, requiere la carga de un archivo de ejemplos y la ejecución de la funcionalidad *induce*. Para así acceder a la validación cruzada ingresando el valor de K en el panel de texto y seleccionando la opción *cross-validation* del menú *id3* o bien a través del metacomando *ctrl+c*.

```
;;; Modificaciones
(defvar *classified-int* ())
(defvar *c-classified-int* ())

(defun gui-cross-validation (data interface)
  (declare (ignore data))
  (progn
    ;; Toma K de la interfaz
    (setf k (text-input-pane-text (k-pane interface)))
    (setf *classified-int* '() *c-classified-int* '()
          *k-validation-trees* '() *classify-on* t)
    ;; Lanza la funcion con K
    (cross-validation (parse-integer k))
    ;; Escribe el mejor arbol
    (traducir *best-tree*)
    ;; Calcula y define la eficiencia
    (setf (text-input-pane-text (e-pane interface))
          (princ-to-string (/ (apply #'+ *c-classified-int*)
                              (apply #'+ *classified-int*))))
    (setf (text-input-pane-text (e1-pane interface))
          (princ-to-string (calculate-voting-accuracy *k-validation-trees*)))
    (setf (text-input-pane-text (e2-pane interface))
          (princ-to-string (calculate-best-tree-accuracy *best-tree*)))))
```

Cuadro 1: Modificaciones al Código en **cl-id3-gui.lisp**

A continuación, se presenta un ejemplo de esta funcionalidad. La Figura 1 muestra el aspecto terminado de la interfaz en la cual se introduce un valor k igual a 4 mientras que el valor desplegado en la segunda casilla es la eficiencia general de los arboles generados a través de *cross-validation*. La Figura 2 expone cada árbol con sus instancias clasificadas.

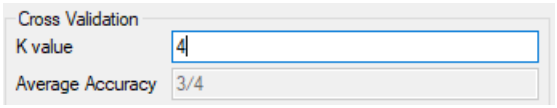


Figura 1: Validación cruzada con K igual a 4.

ÁRBOL 1	ÁRBOL 2	ÁRBOL 3	ÁRBOL 4
HUMEDAD - NORMAL CIELO - NUBLADO -> SI - LLUVIA TEMPERATURA - FRIO -> NO - TEMPLADO -> SI - SOLEADO -> SI - ALTA CIELO - LLUVIA -> NO - NUBLADO -> SI - SOLEADO -> NO	HUMEDAD - NORMAL -> SI - ALTA CIELO - SOLEADO -> NO - LLUVIA VIENTO - DEBIL -> SI - FUERTE -> NO - NUBLADO -> SI	CIELO - SOLEADO HUMEDAD - NORMAL -> SI - ALTA -> NO - LLUVIA VIENTO - FUERTE -> NO - DEBIL -> SI - NUBLADO -> SI **MEJOR ÁRBOL**	HUMEDAD - ALTA CIELO - SOLEADO -> NO - LLUVIA -> NO - NUBLADO -> SI - NORMAL CIELO - SOLEADO -> SI - LLUVIA VIENTO - DEBIL -> SI - FUERTE -> NO - NUBLADO -> SI
Clasificaciones correctas: 2 Clasificaciones incorrectas: 2	Clasificaciones correctas: 3 Clasificaciones incorrectas: 1	Clasificaciones correctas: 4 Clasificaciones incorrectas: 0	Clasificaciones correctas: 3 Clasificaciones incorrectas: 1

Figura 2: Ejemplo de arboles generados aleatoriamente.

Finalmente, es importante notar que la función validación cruzada provista en **cl-id3-cross-validation.lisp** usa el valor *k* como el número de iteraciones que se repite el proceso y de la misma forma como el tamaño del conjunto de datos para la validación de cada árbol.

2. Clasificación por votación

Existen métodos para reducir la varianza de las estimaciones de predicción y mejorar su precisión [HTF01]. En este caso en particular, existe la posibilidad de crear un proceso de votación entre los árboles generados en la validación cruzada para que la clase elegida para un ejemplo nuevo sea aquella que fue más votada por el conjunto de los árboles. Para lograr este objetivo se realizaron adiciones al código original en diversos archivos del paquete, cabe resaltar, que dentro de este reporte no serán incluidos los códigos empleados para la modificación y creación de la interfaz gráfica, pero serán abordados los aspectos más relevantes de la misma.

Como primer punto y continuación de la sección anterior se procede a crear las variables **k-validation-trees** y **best-tree**, en las cuales se almacenan el conjunto de árboles generados por la validación cruzada y el árbol con mayor eficiencia respectivamente. Para brindarles un valor a estas variables se realizaron modificaciones a cross-validation mostrada en el Cuadro 2.

```
(defun cross-validation (k)
  ;; Se limplia la variable *k-validation-trees* cada vez que se ejecuta
  la funcion
  (let* ((long (length *examples*))(k-validation-trees '()))
    (loop repeat k do
         (let* ((trainning-data (folding (- long k) long))
                (test-data (difference trainning-data *examples*))
                (tree (induce trainning-data)))
           ;; Se agrega el arbol creado en cada iteracion
        (setf *k-validation-trees* (append *k-validation-trees* (list tree)))
           (progn
             (report tree test-data))))
             ;; Selecciona el mejor arbol con la funcion "select-bt" obteniendo la
             posicion del arbol que consigue mayor numero de instancias correctas
             (setf *best-tree* (nth (select-bt *c-classified-int*)
                                    *k-validation-trees*))))

;;; Regresa el indice del elemento mas grande de una lista, esta funcion fue
creada para ser usada con la variable *c-classified-int*, si existen mas de un
elemento con el mismo valor regresa el indice del primer elemento
(defun select-bt (lst)
  (let ((cont 0) (index 0) (acc 0))
   (progn
     (loop for x in lst
           do (progn
                (when (> x acc) (and (setf acc x) (setf index cont)))
                (setf cont (+ cont 1))))
     index)))
```

Cuadro 2: Modificaciones al código en **cl-id3-cross-validation.lisp**

Como detalle adicional se modificó la interfaz gráfica con la adición de dos paneles de texto que despliegan la eficiencia de la votación y la eficiencia del mejor árbol respectivamente, cabe resaltar que estos valores son desplegados cuando se realiza la validación cruzada. Las funciones empleadas para el cálculo de la efciencia son mostradas en el Cuadro 3 y los resultados obtenidos del ejemplo de la Sección 2 son mostrados en la Figura 3.

```
    (defun calculate-voting-accuracy (trees)
      (/ (count-voting-positives trees *examples*) (length *examples*)))
    (defun calculate-best-tree-accuracy (best-tree)
      (/ (count-positives best-tree *examples*) (length *examples*)))

     (defun count-voting-positives (tree data)
      (apply #'+
         (mapcar #'(lambda (e)
                     (if (eql (classify-new-instance-votacion e tree)
                              (get-value *target* e))
                         1 0)) data)))
```

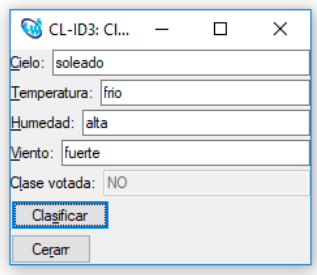
Cuadro 3: Modificaciones al código en **cl-id3-cross-validation.lisp**

Figura 3: Resultados de la clasificación por votación

Asimismo, se en creó una función de clasificación por votación, la cual esta ligada a una ventana que permite introducir nuevos ejemplos para ser clasficados por los árboles creados en la validación y al final desplegar la clase más votada. Todo con el fin de poder comparar su eficiencia con la del mejor árbol de la validación cruzada. Para lograr esto, como se muestra en el Cuadro 4 la función *classify-new-instance* recibe un nuevo ejemplo obtenido de los campos de la interfaz, junto con un conjunto de árboles y regresa la clase que es más recurrente en la clasificación.

```
(defun classifyn-gui (data interface)
  (declare (ignore data))
  (progn
    (setf nsi 0 nno 0)
    ;; Obtiene datos de la interfaz grafica
    (setf new-lst (list (read-from-string (text-input-pane-text (ex1-pane interface)))
                        (read-from-string (text-input-pane-text (ex2-pane interface)))
                        (read-from-string (text-input-pane-text (ex3-pane interface)))
                        (read-from-string (text-input-pane-text (ex4-pane interface)))))
    ;; Guarda en una lista las clases
    (setf new-l (loop for arbol in *k-validation-trees*
                      collect (classify-new-instance new-lst arbol)))
    (setf nsi (length (loop for class in new-l
                            when (equal (string class) "SI")
                            collect class)))
    (setf nno (- (length new-l) nsi))
    ;; Despliga la clase mas comun
    (if (> nsi nno)
        (setf (text-input-pane-text (most-voted-class interface)) (princ-to-string 'si))
      (setf (text-input-pane-text (most-voted-class interface)) (princ-to-string 'no)))))
```

Cuadro 4: Modificaciones al código en **cl-id3-classify.lisp**



RESULTADOS DE LA CLASIFICACIÓN DEL NUEVO EJEMPLO	
ÁRBOL 1	**ÁRBOL 2**
NO	NO
ÁRBOL 3	**ÁRBOL 4**
NO	NO

Figura 4: Clasificación de un ejemplo nuevo.

Finalmente se incluye en la interfaz las estimaciones del error de predicción del método de votación y del mejor árbol inducido en la validación cruzada. Este error se calcula como el número de ejemplos clasificados correctamente sobre el número total de ejemplos clasificados. En este caso todos los 14 ejemplos son utilizados. En la Figura 4 podemos ver un ejemplo de una corrida con $K = 4$. También se puede notar que aunque el desempeño del mejor árbol es mejor que la votación, generalmente la votación provee resultados libres de sobreajustes.

3. Traducción a Prolog

Por último, se agregaron funciones al paquete para poder hacer una traducción automática del mejor árbol inducido por la validación cruzada a Prolog y poder clasificar nuevos ejemplos. Estas funciones se activan cuando la validación cruzada se realiza y automáticamente toman el mejor árbol para escribir el programa **arbol.pl** que se muestra en el Cuadro 7. Las adiciones se muestran en los Cuadros 5 y 6.

```
(defun traducir2 (arbol padre etiqueta)
   ;Hacer que los simbolos se muestren en minusculas.
   (let ((*print-case* :downcase))
   ;Verifica si el argumento es un arbol o un SI/NO.
   (if (listp arbol)
      ;Hacer un loop en los valores del atributo en cuestion para
      ;generar los nodos del arbol de Prolog.
      (loop for i in (cdr (atributo-valores arbol)) do
         (progn
         ;Escribir los nodos del arbol de Prolog.
         (if (eq padre 0)
            (escribir (setf etiqueta (+ etiqueta 1)) (car (atributo-valores arbol)) i 'raiz)
            (escribir (setf etiqueta (+ etiqueta 1)) (car (atributo-valores arbol)) i padre))
         ;Hacer un loop en los subarboles que estan despues de cada valor del atributo
         ;correspondiente para aplicarles esta misma funcion de manera recursiva.
         (loop for j in (cdr arbol) do
            (if (equal i (car j))
               (if (> (length j) 1)
                  (setf etiqueta (traducir2 (cadr j) etiqueta etiqueta))
                  (setf etiqueta (traducir2 (car j) etiqueta etiqueta))))))
         ;Retornar la etiqueta del ultimo nodo creado para que su
         ; valor no se pierda despues de cada recursiva.
         finally (return etiqueta))
      (progn
      ;Si el argumento es de la forma SI/NO escribir un nodo hoja.
      (escribir 'a arbol 'c etiqueta)
      etiqueta))))
```

Cuadro 5: Función principal para la traducción del árbol.

```

;Funcion que sirve como interfaz de "traducir2" y escribe
;el programa de Prolog que clasifica.

(defun traducir (arbol)

   (progn
   (traducir2 arbol 0 0)
   (with-open-file (str "/home/j6/quicklisp/local-projects/cl-id3/cl-id3/arbol.pl"
               :direction :output
               :if-exists :append
               :if-does-not-exist :create)
   (format str "~%~%%Ejemplo:[cielo=soleado,temperatura=alta,humedad=alta,viento=debil].~%
jugarTenis(Ejemplo)␣:-␣member(X=Y,Ejemplo),␣nodo(N,X=Y,raiz),␣jugarTenis(Ejemplo,N),!.~%
jugarTenis(Ejemplo,N)␣:-␣member(X=Y,Ejemplo),␣nodo(N2,X=Y,N),␣jugarTenis(Ejemplo,N2).~%
jugarTenis(_,N)␣:-␣nodo(hoja,[X/_],N),␣write(X)."))))
```

Cuadro 6: Funciones auxiliares para traducir el árbol de Lisp a Prolog.

La función *traducir* actúa como una interfaz para *traducir2* en la cual están incluidos los procesos importantes. La función de *traducir2* es tomar como argumento un árbol en el formato definido para el programa *ID3* de Lisp, por ejemplo:

```
(CIELO (SOLEADO (HUMEDAD (NORMAL SI) (ALTA NO)))
    (NUBLADO SI) (LLUVIA (VIENTO (FUERTE NO) (DEBIL SI))))
```

A partir de dicho árbol, y mediante la función *atributo-valores* se genera una lista cuyo primer elemento es el atributo que está en la raíz del árbol, mientras que los elementos siguientes son los posibles valores que puede tomar dicho atributo, y que aparecen como el primer elemento de las sublistas que contiene el árbol, es decir que para el árbol anterior se generaría la siguiente lista:

```
(CIELO SOLEADO NUBLADO LLUVIA)
```

Mediante la información de esta lista se generan los predicados que dan forma al árbol en el lenguaje Prolog. Mediante un proceso iterativo, la lista anterior generaría los siguientes predicados para Prolog:

```
nodo(1,cielo=soleado,raiz).
nodo(4,cielo=nublado,raiz).
nodo(5,cielo=lluvia,raiz).
```

Aquí los valores numéricos corresponden a las etiquetas de los nodos, los cuales son incrementados en cada iteración.

La función *traducir2* hace esta labor de manera recursiva en los subárboles que están después de cada valor de un cierto atributo. Por ejemplo, después de generar el nodo con la etiqueta 1 el algoritmo se llamaría a si mismo, tomando ahora como entrada al subárbol que viene después del valor *SOLEADO*:

```
(HUMEDAD (NORMAL SI) (ALTA NO))
```

Lo cual generaría a los nodos siguientes mediante el proceso antes descrito:

```
nodo(2,humedad=normal,1).
nodo(3,humedad=alta,1).
```

Las condiciones para terminar una llamada recursiva son: terminar de crear los nodos correspondientes a los valores que toma el atributo en cuestión; o recibir como entrada una lista con únicamente dos símbolos, en la cual el segundo es un *si* o un *no*, en cuyo caso se genera un nodo como alguno de los siguientes:

```
nodo(hoja,[si/\_],2).
nodo(hoja,[no/\_],3).
```

Finalmente la función *traducir* escribe el programa para clasificar ejemplos en Prolog, para ello simplemgente hay que cargar el archivo *arbol.pl* y hacer la consulta *jugarTenis(Ejemplo)*, que toma como argumento un ejemplo con el siguiente formato:

```
[cielo=soleado,temperatura=alta,humedad=alta,viento=debil].
```

El resultado de la consulta es simplemente *si* o *no*.

```

nodo(1,cielo=soleado,raiz).
nodo(2,humedad=normal,1).
nodo(hoja,[si/_],2).
nodo(3,humedad=alta,1).
nodo(hoja,[no/_],3).
nodo(4,cielo=nublado,raiz).
nodo(hoja,[si/_],4).
nodo(5,cielo=lluvia,raiz).
nodo(6,viento=fuerte,5).
nodo(hoja,[no/_],6).
nodo(7,viento=debil,5).
nodo(hoja,[si/_],7).


%Ejemplo:[cielo=soleado,temperatura=alta,humedad=alta,viento=debil].
jugarTenis(Ejemplo) :- member(X=Y,Ejemplo), nodo(N,X=Y,raiz), jugarTenis(Ejemplo,N),!.
jugarTenis(Ejemplo,N) :- member(X=Y,Ejemplo), nodo(N2,X=Y,N), jugarTenis(Ejemplo,N2).
jugarTenis(_,N) :- nodo(hoja,[X/_],N), write(X).
```

Cuadro 7: Mejor árbol en Prolog

Después de haber ejecutado la validación cruzada automáticamente se tendrá el programa de Prolog en el proyecto. Como muestra el Cuadro 8, este programa puede clasificar nuevos ejemplos con el mejor árbol inducido.

```
%%% ?- jugarTenis([cielo=nublado,temperatura=alta,humedad=alta,viento=debil]).
%%% si
%%% true.
```

Cuadro 8: Ejecución de **arbol.pl**

Uno de los problemas principales de esta traducción es que no existe un puente directo entre Lisp y Prolog que nos permita hacer queries desde Lisp en Prolog y poder integrarlos a la interfaz para un mejor análisis. En este caso la conexión entre los dos lenguajes es más rígidia.

Referencias

[GH18] Alejandro Guerra-Hernández. *Lisp en la IA*. https://www.uv.mx/personal/aguerra/files/2018/11/pia-07.pdf, 2018.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[Qui86] JR Quinlan. *Induction of Decision Trees*. Kluwer Academic Publishers, 1986.

[Zuc00] Walter Zucchini. *An Introduction to Model Selection*. Journal of Mathematical Psychology, 2000.