Aprendizaje de Reglas

Eduardo Morales, Hugo Jair Escalante

INAOE

Aprendizaje de Reglas

- El aprendizaje de reglas normalmente produce resultados más fáciles de entender.
- Las reglas son del tipo:

If $Atr_i = val_j \wedge Atr_k = val_l \wedge \ldots$
Then $Clase = val_m$

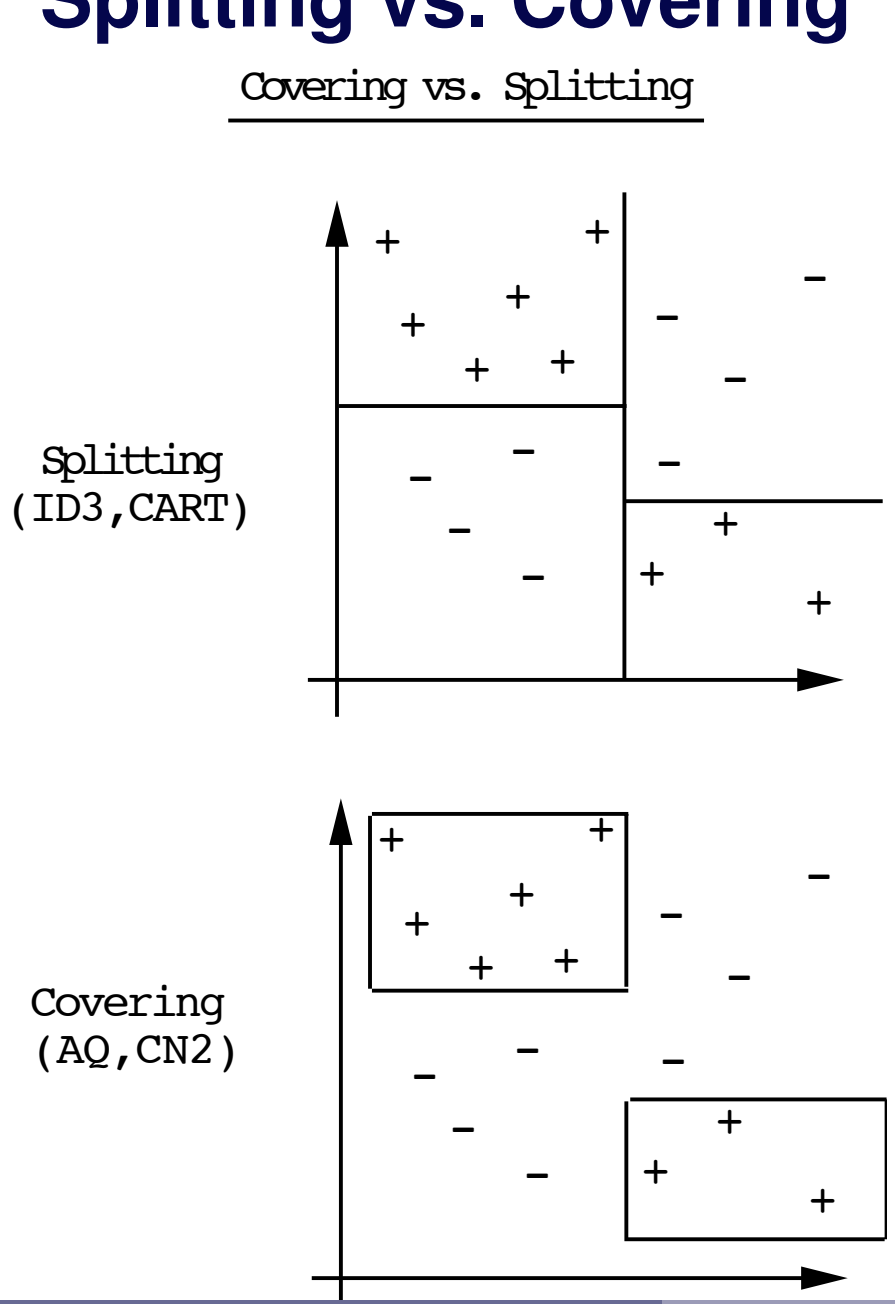
Splitting vs. Covering

- La estrategia básica de la construcción de árboles de decisión se basa en *splitting*:
  - Divide el conjunto de datos en subconjuntos considerando un atributo seleccionado por una heurística particular.
  - Se consideran todas las clases dentro de la partición.
  - La idea básica es añadir atributos al árbol que se está construyendo buscando maximizar la separación entre las clases.

Splitting vs. Covering

- La estrategia utilizada para aprender reglas, está basada en *covering*:
  - Encontrar condiciones de reglas (par atributo-valor) que cubra la mayor cantidad de ejemplos de una clase, y la menor del resto de las clases.
  - Se considera el cubrir una sola clase
  - La idea básica es añadir pruebas a cada regla que se está construyendo buscando maximizar covertura minimizando errores (ver figura).

Splitting vs. Covering



Splitting vs. Covering

- Las reglas pueden expresar disjunciones de manera más fácil que los árboles.
- Por lo mismo, extaer reglas directamente de árboles tiende a producir reglas más complejas de lo necesario.
- Los árboles tienen lo que se conoce como *replicated subtree problem*, ya que a veces repiten subárboles en varios lados.
- Por otro lado, si se quieren construir árboles a partir de reglas, no es trivial y tiende a dejar árboles incompletos.
- Las reglas muchas veces se prefieren con respecto a los árboles por representar “pedazos” de conocimiento relativamente *independientes*.

Listas de Decisión

- Normalmente los sistemas generan lo que se llaman listas de decisión (*decision lists*)
- Son conjuntos de reglas que son evaluadas en orden. Esto facilita la evaluación, pero disminuye su modularidad.
- El tener reglas que pueden ser evaluadas independientemente de su orden, permite que existan más de una predicción para un solo ejemplo y dificulta el producir un solo resultado.

1R

- Vamos a ver primero un sistema muy simple (1R) que es el equivalente a un *decision stump* o árbol de un solo nivel
- La idea es hacer reglas que prueban un solo atributo
- Se prueban todos los pares atributo-valor de cada atributo y se selecciona el atributo que ocasione el menor número de errores

Algoritmo 1R

Para cada atributo
    Para cada valor del atributo, crea una regla:
        cuenta cuántas veces ocurre la clase
        encuentra la clase más frecuente
        asigna esa clase a la regla
    Calcula el error de todas las reglas
Selecciona las reglas del atributo con el error más bajo

1R: Ejemplo

Ambiente	Temp.	Humedad	Viento	Clase
soleado	alta	alta	no	N
soleado	alta	alta	si	N
nublado	alta	alta	no	P
lluvia	media	alta	no	P
lluvia	baja	normal	no	P
lluvia	baja	normal	si	N
nublado	baja	normal	si	P
soleado	media	alta	no	N
soleado	baja	normal	no	P
lluvia	media	normal	no	P
soleado	media	normal	si	P
nublado	media	alta	si	P
nublado	alta	normal	no	P
lluvia	media	alta	si	N

1R: Ejemplo

De la tabla:

- Atributo Ambiente: 4 errores
- Atributo Temperatura: 5 errores
- Atributo Humedad: 4 errores
- Atributo Viento: 5 errores

Los empates se rompen arbitrariamente

1R: Ejemplo

Resultado (para Ambiente):

- If Atributo = soleado
  Then clase = N (cubre 5 y tiene 2 errores)
- If Atributo = nublado
  Then clase = P (cubre 4 y tiene 0 errores)
- If Atributo = lluvioso
  Then clase = P (cubre 5 y tiene 2 errores)

1R: Valores faltantes y contínuos

- Los valores faltantes en 1R se tratan como un nuevo valor
- Para los atributos contínuos se hace una división simple:
  - Primero se ordenan los atributos con respecto a la clase (como lo vimos con árboles de decisión).
  - Se sugieren puntos de partición en cada lugar donde cambia la clase.
  - Si existen dos clases diferentes con el mismo valor, se mueve el punto de partición a un punto intermedio con el siguiente valor hacia arriba o abajo dependiendo de donde está la clase mayoritaria.

1R

- Un problema más serio es que el algoritmo tendería a favorecer contruir reglas para cada una de las particiones, lo cual le da una clasificación perfecta (pero muy poca predicción futura).
- Lo que se hace es que se exige que cada partición tenga un número mínimo de ejemplos de la clase mayoritaria.
- Cuando hay clases adyacentes con la misma clase mayoritaria, éstas se juntan.

1R

Ejemplo:

64	65	68	69	70	71	72	72	75	75	80	81	83	85
P	N	P	P	P	N	N	P	P	P	N	P	P	N

- Tomando los puntos intermedios sería: 64.5, 66.5, 70.5, 72, 77.5, 80.5 y 84.

P ‖ N ‖ P P P ‖ N N ‖ P P P ‖ N ‖ P P ‖ N

1R

P ‖ N ‖ P P P ‖ N N ‖ P P P ‖ N ‖ P P ‖ N

- Si tomamos al menos 3 elementos por partición (en resultados experimentales con varios dominios, este valor se fijo a 6):

P N P P P ‖ N N P P P ‖ N P P N

- Si juntamos clases con la misma clase mayoritaria, nos queda:

P N P P P N N P P P ‖ N P P N

1R

- Lo cual nos daría una regla del tipo:

  If Temperatura $\leq$ 77.5
  Then Clase = P (cubre 10 y tiene 3 errores)

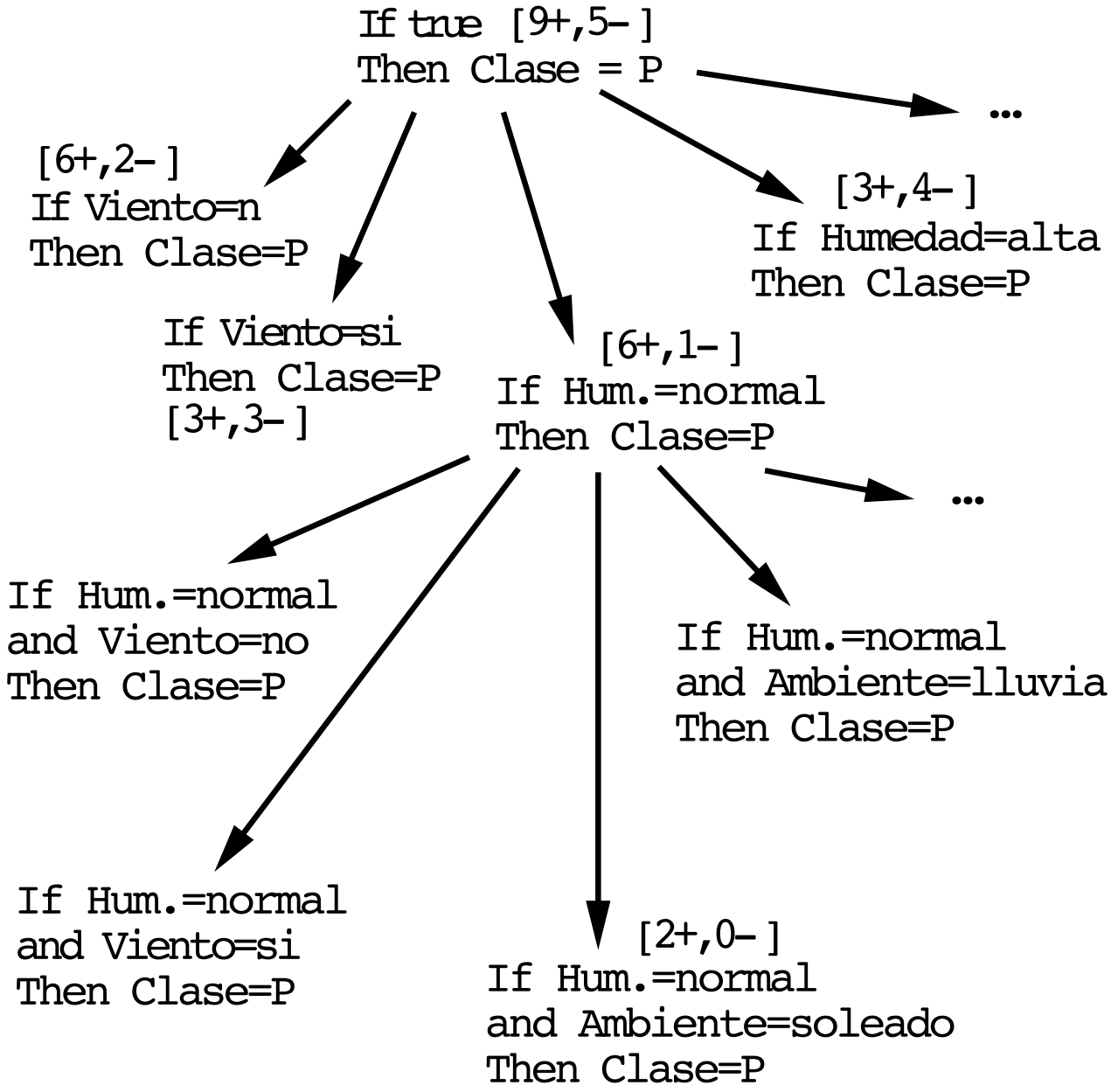
  If Temperatura $>$ 77.5
  Then Clase = N (cubre 4 y tiene 2 errores)

- En la segunda regla se hizo una selección aleatoria, ya que se tenía un empate.

PRISM

- Es un algoritmo básico de aprendizaje de reglas que supone que no hay ruido en los datos
- Sea t el número de ejemplos cubiertos por la regla y p el número de ejemplos positivos cubiertos por la regla
- PRISM añade condiciones a reglas que maximicen la relación p/t (relación entre ejemplos positivos cubiertos y ejemplos cubiertos en total)

PRISM



Algoritmo de PRISM

Para cada clase C
  Sea E = ejemplos de entrenamiento
  Mientras E tenga ejemplos de clase C
    Crea una regla R con LHS vacío y clase C
    Until R es perfecta **do**
      Para cada atributo A no incluido en R
          y cada valor v,
        Considera añadir la condición "$A = v$" a R
        Selecciona el par $A = v$ que maximice p/t
        (en caso de empates, selecciona la que
          tenga p mayor)
    Añade "$A = v$" a R
  Elimina de E los ejemplos cubiertos por R

Reglas Ordenadas vs. No Ordenadas

- Como PRISM va eliminando los ejemplos que va cubriendo cada regla, las reglas que se construyen tienen que interpretarse en orden (las nuevas reglas se diseñan sólo para cubrir los casos que faltan).
- Reglas que dependen del orden para su interpretación se conocen como *decision lists* o listas de decisión.
- Las reglas ordenadas son en general más rápidas de producir ya que van reduciendo el número de ejemplos a considerar
- Reglas que no dependen del orden son más modulares, pero pueden producir varias clasificaciones o no predecir nada.
- Con varias clasificaciones se puede seleccionar la regla que cubra más ejemplos, y cuando no se tiene una clasificación, escoger la clase mayoritaria.

Ejemplo

Consideremos la siguiente tabla:

A_1	A_2	A_3	A_4	Clase
1	x	△	a	P
0	x	◯	a	N
1	y	□	a	P
1	y	△	b	P
1	x	□	b	N
0	y	◯	a	P
0	x	△	b	N
1	y	◯	a	P

Ejemplo

Si empezamos con la clase P construimos todas las posibles combinaciones de atributo valor y evaluamos su predicción sobre la clase P:

If $A_1 = 1$	If $A_1 = 0$	If $A_2 = x$
Then Clase = P	Then Clase = P	Then Clase = P
4/5	1/3	1/4
If $A_2 = y$	If $A_3 = \triangle$	If $A_3 = \bigcirc$
Then Clase = P	Then Clase = P	Then Clase = P
4/4	2/3	2/3
If $A_3 = \square$	If $A_4 = a$	If $A_4 = b$
Then Clase = P	Then Clase = P	Then Clase = P
1/2	4/5	1/3

Ejemplo

En este caso una regla es perfecta: *If* $A_2 = y$ *Then Clase* = P, por lo que seleccionamos ésa y eliminamos todos los ejemplos que cubre:

A_1	A_2	A_3	A_4	Clase
1	x	△	a	P
0	x	◯	a	N
1	x	□	b	N
0	x	△	b	N

Repetimos lo mismo con los ejemplos que quedan.

Ejemplo

If $A_1 = 1$
Then Clase = P
1/2

If $A_1 = 0$
Then Clase = P
0/2

If $A_2 = x$
Then Clase = P
1/4

If $A_3 = \triangle$
Then Clase = P
1/2

If $A_3 = \bigcirc$
Then Clase = P
0/1

If $A_3 = \square$
Then Clase = P
0/1

If $A_4 = a$
Then Clase = P
1/2

If $A_4 = b$
Then Clase = P
0/2

En este caso, tenemos tres empates de 1/2 como valor máximo.

Ejemplo

Tomamos uno al azar y construimos todas las posibles reglas añadiendole posibles pares atributo-valor:

If $A_1 = 1$	If $A_1 = 1$	If $A_1 = 1$
And $A_2 = x$	And $A_3 = \triangle$	And $A_3 = \bigcirc$
Then Clase = P	Then Clase = P	Then Clase = P
1/2	1/1	0/0

If $A_1 = 1$	If $A_1 = 1$	If $A_1 = 1$
And $A_3 = \square$	And $A_4 = a$	And $A_4 = b$
Then Clase = P	Then Clase = P	Then Clase = P
0/1	1/1	0/1

De nuevo tenemos empates y tomamos uno aleatoriamente (el primero).

Ejemplo

- Las reglas entonces para la clase *P* son:

If A_2 = y	If A_1 = 1 and A_3 = △
Then Clase = P	Then Clase = P

- Lo mismo hay que hacer para el resto de las clases
- Para la clase *N* unas posibles reglas son:

If A_2 = x and A_1 = 0	If A_2 = x and A_3 = □
Then Clase = N	Then Clase = N

AQ

- AQ fue uno de los primeros sistemas de reglas
- Desarrollado originalmente por Michalski (79), y reimplementado y mejorado por otros autores (e.g., AQ11, AQ15).
- Su salida es un conjunto de reglas de clasificación del tipo “if...then...”

AQ: principios

- Selecciona aleatoriamente un ejemplo (semilla)
- Identifica el ejemplo de otra clase que sea más *cercano*
- Especializa la regla actual (añade condiciones atributo-valor) para no cubrir ese ejemplo negativo
- Trata de cubrir a la mayor cantidad de ejemplos positivos.

AQ

Se exploran varias alternativas de reglas (*beam-search*).
Heurísticas que se usaron para seleccionar la mejor regla:

- Sumar los ejemplos positivos cubiertos y los negativos excluidos (en caso de empate, preferir la más corta)
- Sumar el número de ejemplos clasificados correctamente dividido por el número total de ejemplos cubiertos
- Maximiza el número de ejemplos positivos cubiertos

Una de sus principales desventajas de AQ es que es sensible a ejemplos con ruido (i.e., si la semilla seleccionada tiene información erronea).

CN2

- CN2 (Clark and Niblett, 1988) se propuso para atacar datos con ruido y evitar el *sobreajuste* que se encontraba en sistemas como AQ
- Su contribución principal es la de quitar la dependencia de un ejemplo específico durante su búsqueda y forma la base de muchos de los algoritmos de reglas actuales.

CN2

- En CN2 se pueden especificar valores *don't-care* "$*$" y valores desconocidos "?"
- Sigue una búsqueda tipo *beam-search*.
- La heurística de búsqueda original que sigue es basada en entropía:

$$Entr = -\sum_i p_i log_2(p_i)$$

donde p_i es la distribución de las clases que cubre cada regla

- Se selecciona la regla de menor entropía, ésto es la regla que cubre muchos ejemplos de una clase y pocos de las demás

CN2

- En una versión posterior usaron *the Laplacian error estimate*:

$$AccuracyA(n, n_c, k) = (n - n_c + k - 1)/(n + k)$$

donde:
n = número total de ejemplos cubiertos por la regla
n_c = número de ejemplos positivos cubiertos por la regla
k = número de clases en el problema.

CN2

- CN2 maneja también una medida de significancia para las reglas
- El usuario proporciona un límite para la medida de significancia, abajo del cual las reglas son rechazadas
- La medida está basada en la razón de verosimilitud (*likelihood ratio statistic*) que mide una distancia entre dos distribuciones, definida por:

$$2\sum_{i=1}^{n} f_i log(\frac{f_i}{e_i})$$

CN2

Donde:

- $F = (f_1, \ldots, f_n)$: Distribución de frecuencias observada de ejemplos dentro de las clases que satisfacen una regla dada:
  - Número de ejemplos que satisfacen la regla entre el número total de ejemplos que satisface la regla
- $E = (e_1, \ldots, e_n)$: Frecuencia esperada en los mismos ejemplos bajo la suposición que la regla selecciona ejemplos aleatoriamente
  - Número de ejemplos cubiertos por la regla siguiendo la distribución de ejemplos del total de los ejemplos
- Entre más baja es la medida es más probable que la aparente regularidad expresada en la regla sea por casualidad

Medidas Alternativas de Selección

- La más simple (que ya vimos), es la frecuencia relativa de ejemplos positivos cubiertos

$$m(R) = \frac{p}{t} = \frac{p}{p+n}$$

t = número total de ejemplos cubiertos por la regla = $p + n$
p = número total de ejemplos positivos cubiertos por la regla
Tiene problemas con muestras pequeñas

- Estimador Laplaciano (CN2). Supone una distribución uniforme de las k clases ($k = 2$)

$$m(R) = \frac{p+1}{p+n+k}$$

Medidas Alternativas de Selección

- Estimador *m*: Considera que las distribuciones *a priori* de las clases ($P_a(C)$), son independientes del número de clases y *m* es dependiente del dominio (entre más ruido, se selecciona una *m* mayor).

$$m(R) = \frac{p + m \cdot P_a(C)}{p + n + m}$$

- Ganancia de Información:

$$log_2 \frac{p}{p+n} - log_2 \frac{P}{P+N}$$

donde *P* y *N* son los ejemplos cubiertos *antes* de que se añadiera la nueva prueba

Medidas Alternativas de Selección

- *Weighted Relative Accuracy*:

$$wla = \frac{p+n}{P+N}\left(\frac{p}{p+n} - \frac{P}{P+N}\right)$$

Valores desconocidos y numéricos

- Con valores desconocidos, en un algoritmo de *covering* lo mejor es hacer como si no aparean ninguna condición (ignorarlos).
- En este sentido, los algoritmos que aprenden *decision lists* tienen cierta ventaja, ya que ejemplos que parecen difíciles al principio, se van quedando y al final se pueden resolver, en general, más fácilmente.
- Los atributos numéricos se pueden tratar igual que con los árboles.

Pruning

- Una forma de evaluar si una regla es buena es considerar la probabilidad de que una regla aleatoria nos de resultados iguales o mejores que la regla a considerar, basados en la mejora en ejemplos positivos cubiertos
- Idea: Generar reglas que cubran puros ejemplos positivos.
- Esta regla es probable que esté sobre-especializada.
- Lo que se hace es que se elimina el último término que se añadió y se verifica si esta regla es mejor a la anterior
- Este proceso se repite hasta que no existan mejoras

Algoritmo de podado de reglas

Inicializa E al conjunto de ejemplos
Until E sea vacío **do**
    Para cada clase C
        Crea una regla perfecta para la clase C
        Calcula la medida de probabilidad $m(R)$ para la regla y para la regla sin la última condición $m(R-)$
        Mientras $m(R-) < m(R)$, elimina la última condición de la regla y repite el proceso
    De las reglas generadas, selecciona aquella con $m(R)$ menor
    Elimina las instancias cubiertas por la regla

Algoritmo de podado de reglas

- Este algoritmo no garantiza encontrar las mejores reglas por 3 razones principales:
  1. El algoritmo para construir las reglas, no necesariamente produce las mejores reglas para ser reducidas
  2. La reducción de reglas empieza con la última condición, y no necesariamente es el mejor orden
  3. La reducción de reglas termina cuando cambia la estimación, lo cual no garantiza que el seguir recortando pueda mejorar de nueva la estimación
- Sin embargo, el procedimiento es bueno y rápido

Medida de Evaluación para reducción de reglas

- ¿Cuál es la probabilidad que una regla seleccionada aleatoriamente con la misma cantidad de ejemplos que cubre R tenga el mismo desempeño?
- Una regla que cubra t casos, de los cuales p sean de la clase C (sin reemplazo):

$$Pr(t_C) = \frac{\binom{P}{p}\binom{T-P}{t-p}}{\binom{T}{t}}$$

Medida de Evaluación para reducción de reglas

Donde:

- p es número de instancias de la clase que la regla selecciona
- t es el número total de instancias que cubre la regla
- P es el número total de instancias de la clase
- T es el número total de instancias

Si queremos ver la probabilidad de que cubra p casos o más, entonces:

$$m(R) = \sum_{i=p}^{min(t,P)} Pr(t_C)$$

Medida de Evaluación para reducción de reglas

- Valores pequeños indican que la regla es buena, ya que es muy poco probable que la regla sea construida por casualidad
- Como este es muy costoso de calcular, se pueden hacer aproximaciones
- Si el número de ejemplos es grande, la probabilidad de que exactamente p ejemplos de los t sean de clase C (con reemplazo) es:

$$Pr(t_C) = \left(\begin{array}{c} t \\ p \end{array} \right) (\frac{P}{T})^p (1 - \frac{P}{T})^{t-p}$$

que corresponde a una distribución binomial

Reduced Error Pruning

- La función acumulada se puede aproximar a una función beta de la siguiente forma:

$$\sum_{i=p}^{t} \binom{t}{i} (\frac{P}{T})^i (1 - \frac{P}{T})^{t-i} = I_\beta(p, t - p + 1)$$

- Todo esto (simplificación de reglas) se puede hacer con un subconjunto del conjunto de ejemplos de entrenamiento (*reduced error pruning*)

Incremental REP

- IREP (*Incremental REP*) simplifica reglas cada vez que se construyen usando:

$$\frac{p + (N - n)}{P + N}$$

- Sin embargo, le da la misma importancia a los ejemplos negativos no cubiertos y los positivos cubiertos
- Por lo que si una regla cubre 2000 (p) de 3,000, esto es, que tiene 1,000 mal (n) es evaluada mejor que una regla que cubre 1,000 (p) de 1,001 ($n = 1$)
- Otra medida popular es:

$$\frac{p - n}{p + n}$$

Pero sufre de problemas parecidos

RIPPER

- Construye un conjunto de reglas usando *covering*
- Reduce las reglas usando alguna heurística como las de arriba con un conjunto de entrenamiento separado
- Luego “optimiza” al mismo tiempo ese conjunto de reglas

RIPPER

RIPPER utiliza varias medidas e ideas al mismo tiempo:

- Un conjunto de ejemplos separados para decidir podar reglas
- Ganancia de información para crecer las reglas
- Medida de IREP para podar reglas
- Medida basada en MDL como criterio de paro para el conjunto global de reglas

RIPPER

Una vez que construye un conjunto inicial de reglas:

- Toma una regla R_i del conjunto total de reglas y la hace crecer (*revision*)
- También hace crecer una nueva regla desde el principio
- Al final se queda con la regla original o alguna de las otras dos (la que hizo crecer o construyo desde cero) tomando en cuenta el error sobre el conjunto total de las reglas

Construir reglas usando árboles

- Es posible crear reglas directamente de un árbol de decisión, sin embargo, las reglas tienden a ser más complejas de lo necesario
- Se pueden utilizar los mecanismos planteados en la sección anterior para ir *podando* las reglas
- Una alternativa es combinar una estrategia de *covering* con una de *splitting*

Construir reglas usando árboles

- Para construir una regla se construye un árbol podado (*splitting*) y la hoja con la mejor covertura se transforma en una regla
- Una vez construida una regla se eliminan todos los ejemplos que cubre (*covering*) y se repite el proceso
- En lugar de construir un árbol completo, se construyen árboles parciales, expandiendo las hojas con mejores medidas de entropía
- Este esquema tiende a producir conjuntos de reglas simples con muy buen desempeño

Ripple-down rules

- La idea es constuir primero las reglas que cubren la mayor cantidad de casos y luego irlas afinando mediante excepciones
- Primero se toma la clase mayoritaria de entrada
- Todo lo que no se cumpla se toma como una excepción a ésta
- Se busca la mejor regla (la que cubra más casos) de otra clase y se añade como una excepción
- Esto divide de nuevo los datos en los que cumplen con esa nueva condición y los que no cumplen
- Dentro de los que no cumplen de nuevo se busca la mejor regla de otra clase y se añade como excepción, y así sucesivamente hasta que se cubran todos los casos

Ejemplo de *Ripple down rules*

Default: reprueba
excepto
    Si estudia=si AND memoriza=no
    Entonces pasa
    excepto
        Si copia=si AND descubren=si
        Entonces reprueba
    Else
        Si estudia=no AND copia=si AND descubren=no
        Entonces pasa
        excepto
            Si vecino-sabe=no
            Entonces reprueba

Una de las ventajas es que la mayoría de los ejemplos se cubren por las reglas de más alto nivel, y las de bajo nivel representan realmente las excepciones

Reglas que incluyen relaciones

- Hasta ahora todas las reglas consideran pruebas sobre atributos (proposicionales)
- En ocasiones es importante expresar relaciones entre atributos
- Por ejemplo, si vemos varias reglas del tipo:
  If ancho $\leq$ 3.5 and alto $\geq$ 4 Then parado
  If ancho $\geq$ 3.5 Then acostado
- Entonces generar reglas en donde lo importante no son los valores sino su relación
  If ancho $<$ alto Then parado
  If ancho $>$ alto Then acostado
- Estas reglas se llaman relacionales. En otro curso veremos ILP y Aprendizaje en Grafos