Tutorial no. 3: Inferencia en redes Bayesianas

Borja Calvo

Resumen

Este tutorial está dedicado a la inferenica en redes Bayesianas y como esta puede ser realizada usando R. En particular, usaremos el paquete `gRain` que proporciona las herramientas necesarias para propagar evidencias, realizar marinalizaciones, etc.

1 Preparación

Antes de nada, debemos cargar los paquetes a utilizar:

```
> library(gRain)
> library(bnlearn)
> library(Rgraphviz)
> library(ggplot2)
```

1.1 Bases de datos

En este tutorial utilizaremos (además de algunas de las bases de datos incluidas en los paquetes) dos bases de datos clásicas obtenidas del repositorio de la UCI[1]. La primera de ellas, denominada *car*, relaciona ciertas características de los coches con como de aceptables son para los clientes. La segunda (*mushroom*) incluye características físicas y ambientales de ciertas setas junto con información acerca de si son comestibles o no. Para facilitar la obtención de las bases de datos estas están guardadas como ficheros .Rdata. Para cargarlos basta con ejecutar:

```
> load("UCI_car.Rdata")
> load("UCI_mushroom.Rdata")
> summary(car)

##     BUY       MAINTAIN     DOORS     PERSONS     LUG_BOOT    SAFETY
##  high :432   high :432   2    :432   2   :576   big  :576   high:576
##  low  :432   low  :432   3    :432   4   :576   med  :576   low :576
##  med  :432   med  :432   4    :432   more:576   small:576   med :576
##  vhigh:432   vhigh:432   5more:432
##    ACCEPT
##  acc  : 384
##  good :  69
##  unacc:1210
##  vgood:  65
```

[1] http://archive.ics.uci.edu/ml/datasets.html

Año: Master ICSI/KISA
Profesor(a): Borja Calvo, Aritz Pérez
Teléfono: 943 01 50 13
E-mail: [borja.calvo,aritz.perez]@ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

```
> summary(mushroom)

##  EDIBILITY CAP_SHAPE CAP_SURF    CAP_COL     BRUISES       ODOR
##  e:3488    b: 300    f:2160   g      :1696   f:2460   n      :2776
##  p:2156    c:   4    g:   4   n      :1164   t:3184   f      :1584
##            f:2432    s:1260   y      :1056            a      : 400
##            k:  36    y:2220   w      : 880            l      : 400
##            s:  32             e      : 588            p      : 256
##            x:2840             b      : 120            c      : 192
##                               (Other): 140            (Other):  36
##  GILL_ATT GILL_SPACE GILL_SIZE    GILL_COL    STALK_SHAPE STALK_ROOT
##  a:  18   c:4620     b:4940    p      :1384   e:2764      ?:   0
##  f:5626   w:1024     n: 704    n      : 984   t:2880      b:3776
##                               w      : 966               c: 556
##                               h      : 720               e:1120
##                               g      : 656               r: 192
##                               u      : 480
##                               (Other): 454
##  STALK_SRF_ABV_RING STALK_SRF_BLW_RING STALK_COL_ABV_RING
##  f: 552             f: 552             w      :3136
##  k:1332             k:1296             p      :1008
##  s:3736             s:3544             g      : 576
##  y:  24             y: 252             n      : 448
##                                        b      : 432
##                                        c      :  36
##                                        (Other):   8
##  STALK_COL_BLW_RING VEIL_COL RING_NUM RING_TYPE   SPORE_COL    POP
##  w      :3088       n:   0   n:  36   e: 824    n      :1920   a: 384
##  p      :1008       o:   0   o:5488   f:   0    k      :1872   c:  52
##  g      : 576       w:5636   t: 120   l:1296    h      :1584   n: 256
##  n      : 496       y:   8            n:  36    w      : 148   s:1104
##  b      : 432                         p:3488    r      :  72   v:2160
##  c      :  36                                   u      :  48   y:1688
##  (Other):   8                                   (Other):   0
##  HAB
##  d:2492
##  g:1860
##  l:  64
##  m: 292
##  p: 568
##  u: 368
##  w:   0
```

Dado que en la base de datos *mushroom* original hay valores perdidos en una de las columnas mientras que otra de ellas solo tiene un posible valor. En el primer caso, las instancias con valores perdidos han sido filtradas mientras que en el segundo caso la columna a sido completamente eliminada[2].

2 Creación de modelos

En tutoriales anteriores ya vimos como aprender redes Bayesianas a partir de los datos. El paquete que utilizamos es el `bnlearn`, que genera un tipo de objeto de tipo `bn`. Por el contrario, el paquete `gRain` maneja otro tipo de

[2]Para más información sobre las variables y su significado conslutar la web de la UCI

objeto llamado `grain`. La forma de pasar de un tipo de objeto a otro es a través de un tercer, llamado `graphNEL`. Para ello debemos obtener la matriz de adjacencia del modelo aprendido, con ello construir un objeto de tipo `graphNEL` y finalmente crear el objeto de tipo `grain` usando el grafo junto con los datos para aprender los parámetros. Veamos el código:

```
> net_bn<-hc(car)
> net_graphnel<-as(amat(net_bn),"graphNEL")
> net_grain<-grain(net_graphnel,data=car)
> summary(net_grain)

## Independence network: Compiled: FALSE Propagated: FALSE
##  Nodes : chr [1:7] "BUY" "MAINTAIN" "DOORS" "PERSONS" "LUG_BOOT" ...
```

A partir de ahí podemos utilizar el modelo creado para realizar inferencia.

Como alternativa podemos crear la red de forma manual. Para ello basta con definir las tablas de probabilidad condicionada, ya que estas determinan como es el DAG. A continuación se muestra el código necesario para crear la red de ejemplo vista en la teoría:

```
> yn<-c("NO","SI")
> s <- cptable(~S, values=c(50,50),levels=yn)
> j <- cptable(~J, values=c(25,75),levels=yn)
> c.sj <- cptable(~C+S+J, values=c(80,20,50,50,50,50,10,90),levels=yn)
> a.sc <- cptable(~A|S:C, values=c(95,5,50,50,25,75,5,95),levels=yn)
> plist<-compileCPT(list(s,j,c.sj,a.sc))
> net.study<-grain(plist)
```

Como podemos ver, el primer paso es definir, usando la función `cptable` las tablas de probabilidad para cada variable. El ejemplo muestra dos tipos de sintaxis válida para definir la lista de padres de una variable.

A continuación, las tablas se añaden a una lista y se usa la función `compileCPT` para generar un objeto el cual es usado por la función `grain` para generar el objeto final.

3 Inferencia exacta

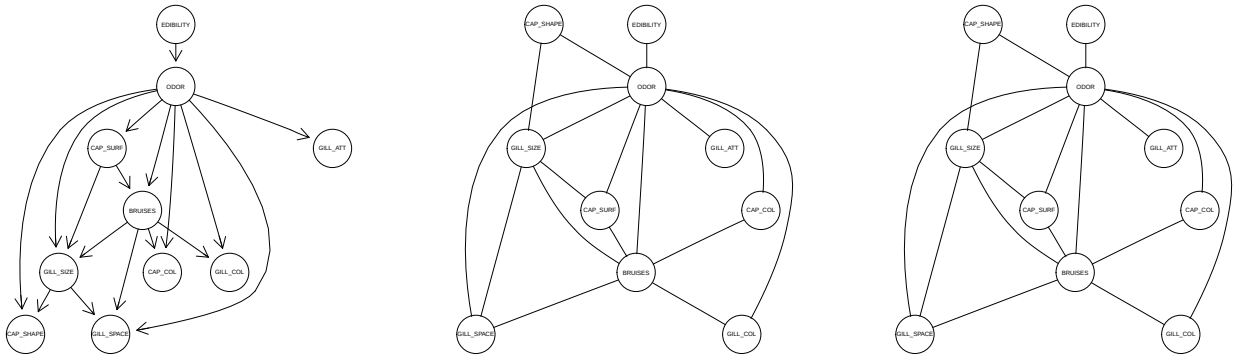
Dada una red Bayesiana, antes de poder realizar inferencia el paquete `gRain` *compila* y *propaga* la red. La compilación no es más que la creación de un *junction tree* que represente la misma distribución que el modelo original. En el segundo paso, la propagación, los potenciales asociados a cada clique son *calibrados* de tal manera que representen la distribución marginal de las variables asociadas.

En caso de que realicemos una consulta, si la red no ha sido compilada y propagada estos pasos se llevan a cabo antes de responder a la consulta. Por ello, una vez creada una red podemos directamente preguntar por, por ejemplo, una distribución marginal:

```
> querygrain(net.study,nodes="A",type="marginal")

## $A
## A
##      NO      SI
## 0.39625 0.60375
```

Como vemos, la función que permite realizar consultas se denomina `querygrain`, y a ella, además del modelo, debemos pasarle uno o varios nodos (sobre los que preguntamos) así como el tipo de consulta, que puede ser `marginal`, para la probabilidad marginal de cada nodo por separado, `joint`, para obtener la distribución conjunta de todas las variables listadas en la opción `nodes` y `conditional` para obtener la distribución de la primera variable condicionada al resto de las variables.



(a) DAG (b) Moralized (c) Triangulated

Figura 1: Original DAG, moralized and triangulated graph obtained from the modeal learened from the first 10 variables in the mushroom dataset

Pese a que no es necesario, dado que la generación y calibrado del *junction tree* solo debe realizarse una vez, es conveniente realizarla de forma explicita, para evitar repetir el proceso en cada consulta. Esto se puede hacer utlizando las funciones `compile` y `propagate`. No obstante, a continuación veremos como realizar la compilación paso a paso.

Para este ejemplo usaremos una versión reducida de la base de datos *mushroom* en la que solo consideraremos las 10 primeras variables. Lo primero que haremos, por tanto, será aprender una red Bayesiana con las 10 primeras columnas de la base de datos. A continuación crearemos el objeto de tipo `grain`

```
> bn_mush<-hc(mushroom[,1:10])

## Warning in check.data(x):  variable CAP_COL has levels that are not observed in the data.
## Warning in check.data(x):  variable ODOR has levels that are not observed in the data.
## Warning in check.data(x):  variable GILL_COL has levels that are not observed in the data.

> net_mush<-grain(as(amat(bn_mush),"graphNEL"),
+                 data=mushroom[,1:10],smooth=1/dim(mushroom)[1])
> plot(net_mush)
```

La Figura 1 muestra (a la izquierda) la red aprendida. En el código puede verse que al crear el objeto `grain` se usa el parámetro `smooth`. Esto es debido a que hay combinaciones de padres que no existen en los datos. El valor que se pasa con este parámetro sirve para dar una probabilidad pequeña en esos casos.

El primer paso de la compilación es la creación del grafo moral, es decir, unir con un arco cualesquiera dos variables que tengan un hijo en común y, posteriormente, eliminar la dirección de los arcos). Una vez moralizado hay que triangularlo, añadiendo arcos hasta asegurar que no hay ciclos de tamaño superior a 3.

```
> net_mush_moral<-moralize(net_mush$dag)
> net_mush_triang<-triangulate(net_mush_moral)
> plot(net_mush_moral)
> plot(net_mush_triang)
```

Ojo, las operación que hemos realizado no son sobre el modelo, sino sobre el DAG. Como se puede ver en la Figura 1, el grafo moral (en el centro) ya está triangulado.

Podemos ver la lista de cliques, así como su ordenación RIP para la factorización, usando la función `rip`.

```
> rip(net_mush_triang)
> plot(rip(net_mush_triang))
```

Esta ordenación es la que posibilita definir un *junction tree* que represente la misma distribución de probabilidad conjunta que el modelo original. Como vemos, la función nos da la lista de cliques, el separador para cada uno de ellos y los paderes de cada clique. También podemos visualizar el arbol realizando un `plot` del objeto creado por la función.

Todos estos pasos, más el cálculo de los potenciales asocidos a cada clique, se puede hacer directamente usando la función `compile` sobre el objeto `grain` original.

```
> net_mush_compiled<-compile(net_mush)
> net_mush$isCompiled

## [1] FALSE

> net_mush_compiled$isCompiled

## [1] TRUE

> summary(net_mush_compiled)

## Independence network: Compiled: TRUE Propagated: FALSE
##  Nodes : chr [1:10] "EDIBILITY" "CAP_SHAPE" "CAP_SURF" "CAP_COL" ...
##  Number of cliques:                 7
##  Maximal clique size:               4
##  Maximal state space in cliques:  216
```

En el resumen del objeto podemos ver cuantos cliques tenemos y el tamaño más grande (en número de variables así como la cardinalidad de la función de potencial).

También podemos ver en el resumen que el modelo no ha sido *propagado*. Esta tarea consiste, como hemos dicho, en una especie de calibración de los potenciales para que representen distribuciones de probabilidad marginales. Esta tarea, que se realiza por medio de un algoritmo de paso de mensajes, se puede realizar con la función `propagate`.

```
> net_mush_propagated<-propagate(net_mush_compiled)
> net_mush$isPropagated

## [1] FALSE

> net_mush_propagated$isPropagated

## [1] TRUE

> summary(net_mush_propagated)

## Independence network: Compiled: TRUE Propagated: TRUE
##  Nodes : chr [1:10] "EDIBILITY" "CAP_SHAPE" "CAP_SURF" "CAP_COL" ...
##  Number of cliques:                 7
##  Maximal clique size:               4
##  Maximal state space in cliques:  216
```

3.1 Absorviendo evidencia

Como hemos visto anteriormente, la función `querygrain` nos permite consultar probabilidades marginales, conjuntas y condicionadas. Respecto a estas últimas, en caso de estar interesados en realizar inferencia en presencia de evidencias (es decir, cuando sabemos o fijamos el valor de ciertas variables), no es necesario computar las probabilidades condicionales para todas las posibles combinaciones de los condicionantes, solo para la combinación correspondiente a la evidencia que disponemos.

Para realizar esto el paquete nos ofrece una serie de herramientas para manipular las evidencias de que disponemos. La primera de ellas es la función `setEvidence` (alternativamente `setFinding`). Al introducir evidencia en el modelo podemos optar por propagarla (actualizar los potenciales) o no. Antes de poder realizar ninguna consulta debemo realizar este paso, pero si vamos a introducir la evidencia en varias llamadas a la función, es más eficiente realizar la propagación al final. Veamos un ejemplo:

```
> net_mush_propagated_ev<-setFinding(net_mush_propagated,nodes="BRUISES",states="t", propagate=F)
> net_mush_propagated_ev<-setEvidence(net_mush_propagated_ev,nodes="GILL_COL",states="n", propagate=F)
> net_mush_propagated_ev<-propagate(net_mush_propagated_ev)
> querygrain(net_mush_propagated,nodes="EDIBILITY",type="marginal")

## $EDIBILITY
## EDIBILITY
##         e         p
## 0.6180014 0.3819986

> querygrain(net_mush_propagated_ev,nodes="EDIBILITY",type="marginal")

## $EDIBILITY
## EDIBILITY
##         e         p
## 0.8920908 0.1079092
```

Primeramente fijamos el valor de dos variables BRUISES (B abreviadamente) y GILL_COL (G_c) a t y n respectivamente. Podemos ver que evidencia hemos fijado usando la función `getEvidence` y la probabilidad de dicha evidencia con la función `pEvidence`. Como se puede ver en las dos primeras líneas, al introducir el evidencia hemos fijado el parámetro `propagate` a `false` y, una vez introducidas las dos evidencias, realizamos la propagación.

Las dos últimas consultas muestran la diferencia entre $P(E)$ y $P(E|B = t, G_c = n)$, siendo E la abreviatura de EDIBILITY. Las evidencias no solo pueden ser introducidas, también pueden ser retiradas del modelo. Vemos otro ejemplo:

```
> net_mush_propagated_ev2<-retractEvidence(net_mush_propagated_ev,nodes=c("BRUISES","GILL_COL"), propag
> net_mush_propagated_ev2<-setEvidence(net_mush_propagated_ev2,nodes=c("CAP_COL","CAP_SHAPE"),states=c(
> querygrain(net_mush_propagated_ev,nodes="EDIBILITY",type="marginal")

## $EDIBILITY
## EDIBILITY
##         e         p
## 0.8920908 0.1079092

> querygrain(net_mush_propagated_ev2,nodes="EDIBILITY",type="marginal")

## $EDIBILITY
## EDIBILITY
##           e           p
## 0.998503831 0.001496169
```

En la primera linea hemos eliminado toda evidencia anteriormente introducida y en la tercera hemos fijado dos nuevas variables, COL_CAP y CAP_SHAPE, fijadas a w y b. Finalmente comparamos la probabilidad de ser comestible ante ambas evidencias.

4 Inferencia aproximada

El paquete `gRain` no incluye algoritmos para realizar inferencia aproximada, por lo que veremos como podemos realizarla de manera manual (al menos en los casos más sencillos). Para ello empezaremos por la más simple, que es el calculo de probabilidades marginales.

Como hemos visto en la parte teórica, una forma simple de calcular una probabilidad marginal consiste en muestrear el modelo y estimar de dicho muestreo la probabilidad marginal que nos interesa. Para realizar el muestreo del modelo podemos utilizar la función `simulate` pasándole dos parámetros, el modelo a muestrear y un parametro `nsim` que indica el número de muestras a obtener.

Supongamos que queremos calcular, como hicimos al principio de la primera sección, la probabilidad marginal de aprobar en el modelo que usamos como ejemplo. En dicha sección vimos que esto se puede hacer de forma exact de la siguiente manera:

```
> querygrain(net.study,nodes="A",type="marginal")

## $A
## A
##      NO      SI
## 0.39625 0.60375
```

Para realizar la estimación de forma aproximada, podemos hacer:

```
> samp<-simulate(net.study,nsim=10)
> summary(samp$A)/10

##  NO  SI
## 0.4 0.6

> querygrain(net.study,nodes="A",type="marginal")

## $A
## A
##      NO      SI
## 0.39625 0.60375
```

Si a la función `summary` le pasamos un vector de factores nos devuelve el número de veces que aparece cada uno de ellos. Por tanto, si dividimos ese conteo por el número de instancias que hemos generado tendremos una estimación de la probabilidad marginal de aprobar.

Como se puede apreciar la estimación es bastante buena pese a solo utilizar 10 muestras. En casos en los que la red es más compleja la estimación puede no ser tan buena:

```
> aprox_marginal<-function(net,node,nsamples){
+   samp<-simulate(net,nsim=nsamples)
+   summary(samp[,node])/nsamples
+ }
> querygrain(net_mush_propagated,nodes="CAP_SURF",type="marginal")

## $CAP_SURF
## CAP_SURF
```

```
##           f           g           s           y
## 0.3827072408 0.0007089679 0.2232459159 0.3933378753

> aprox_marginal(net_mush,"CAP_SURF",10)

##   f   g   s   y
## 0.4 0.0 0.2 0.4

> aprox_marginal(net_mush,"CAP_SURF",10)

##   f   g   s   y
## 0.4 0.0 0.2 0.4

> aprox_marginal(net_mush,"CAP_SURF",10)

##   f   g   s   y
## 0.2 0.0 0.1 0.7

> aprox_marginal(net_mush,"CAP_SURF",1000)

##     f     g     s     y
## 0.375 0.002 0.247 0.376
```

A fin de facilitar los cálculos hemos creado una función que, dada una red y el nombre de un nodo, nos calcula de forma aproximada su marginal. Adicinalmente como parámetro le pasamos el número de muestras a utilizar en el cálculo. A continuación ponemos en práctica la función calculando la distribución de probabilidad marginal de la varaible CAP_SURF usando 10 y 1000 muestreos. Previamente hemos calculado de forma exacta la probabilidad marginal exacta.

El experimento de arriba ponde de manifiesto dos cosas. La primera es que, al ser un proceso estocástico, cada vez que realicemos la estimación obtendremos un valor diferente. Cuanto menor sea el número de muestras utilizadas, mayor será la varainza de la estimación. La otra cosa que podemos ver es que el error cometido decrece cuando usamos un número mayor de muestras. Ambos aspectos se pueden reflejar en la siguiente gráfica:

```
> df<-data.frame()
> node<-"GILL_SIZE"
> marg_exacta<-querygrain(net_mush,nodes=node,type="marginal")[[1]][1]
>
> for (i in seq(10,210,20)){
+   aux<-vector()
+   for (rep in 1:10){
+     aux<-c(aux,aprox_marginal(net_mush,node,i)[1])
+   }
+   df<-rbind(df,data.frame(size=i,min=min(aux),max=max(aux)))
+ }
>
> ggplot(df,aes(x=size,ymin=min,ymax=max)) +
+   geom_ribbon(fill="slategray1") +
+   geom_line(aes(y=min),col="darkgray",size=1.5) +
+   geom_line(aes(y=max),col="darkgray",col="darkgray",size=1.5) +
+   geom_abline(intercept=marg_exacta,slope=0) +
+   labs(x="No. de muestras",y="Envolvente de las estimaciones") +
+   annotate("text", x=25,y=marg_exacta+0.01,label="Valor exacto")

## Warning:  The plyr::rename operation has created duplicates for the following name(s):  (‘colour‘)
```

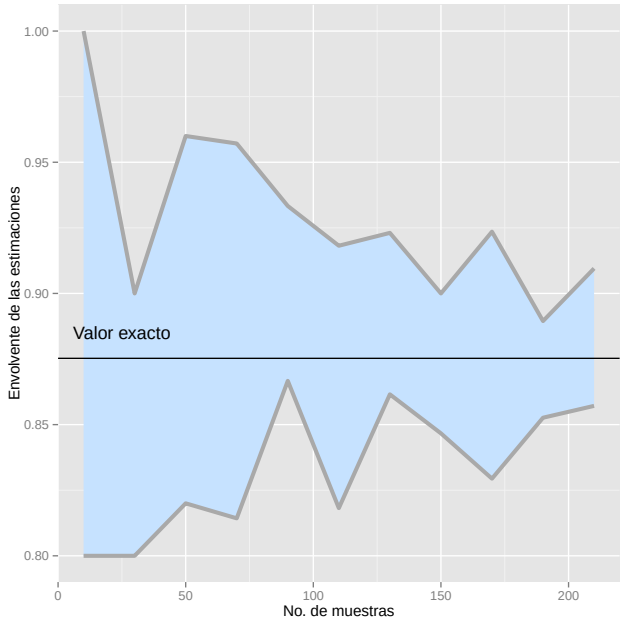


Figura 2: Evolución de la estimación de la probabilidad marginal de la variable GILL_SIZE en función del número de muestras utilizadas

El resultado se muestra en la Figura 2. Para diferentes tamaños de muestra (entre 10 y 210) hemos realizado 10 estimaciones, quedándonos con los valores máximo y mínimo para cada tamaño. La gráfica muestra sombreada la zona entre dichos valores. Como se puede apreciar, a medida que el número de muestras utilizadas aumenta la variabilidad de las estimaciones decrece y se va aproximando al valor exacto.

5 Ejercicios

5.1 Problema. *En la línea de la gráfica mostrada en la Figura 2, crear una gráfica en la que se muestre para nuestro ejemplo de red Bayeiana el error medio (de 5 repeticiones) al estimar de forma aproximada la probabilidad de aprobar frente al número de muestras utilizadas (entre 10 y 5000).*

5.2 Problema. *A la hora de estimar probabilidades marginales disponiendo de evidencia, en algunos casos es evidencia puede ser usada directamente (por ejemplo, cuando sabemos el valor de una variable que no tiene padres). Sin embargo, en el caso general no es posible hacer esto. En nuestro ejemplo de red, si sabemos que* $C = y$ *no podemos simplemente muestrear* S*, fijar la probabilidad condicionada* $P(A|S, C = y)$ *y muestrear* A*. Comprobar que esta manera de calcular* $P(A|C = y)$ *no es correcta.*

Reto no. 2 - Cálculo aproximado de marginales condicionadas

El cálculo aproximado de las probabilidades marginales también puede hacerse en presencia de evidencia. Una de las técnicas para ello es el llamado método de aceptación/rechazo. La idea es muestrear la red de forma iterativa, reteniendo únicamente aquellas muestras que coinciden con la evidencia disponible para, posteriormente, basarse en dichas muestras para obtener las probabilidades marginales. Un ejemplo sería el cálculo de la probabilidad de aprobar (en nuestro modelo de ejemplo) sabiendo que hemos comprendido la asignatura. El proceso sería el siguiente:

---

1 Entradas: Un modelo M, las evidencias $\mathbf{E} = \mathbf{e}$ de evidencias, una variable del modelo X, un número de muestras m
2 Salida: Estimación de $P(X|\mathbf{E} = \mathbf{e})$
3 Inicializar la muestra $\mathcal{S}$ a $\emptyset$
4 **mientras** $|S| < m$
5 Obtener una muestra de $\mathbf{s}$ M
6 **si** en $\mathbf{s}$ se cumple que $\mathbf{E} = \mathbf{e}$
7 Añadir $\mathbf{s}$ a $\mathbf{S}$
8 **is**
9 **sartneim**
10 Estimar de $\mathbf{S}$ la distribución de probabilidad $P(X)$

El reto consiste implementar este algoritmo en una función de R definida como `get_conditional_marginal`. La funciónd deberá tener los siguientes parámetros:

- `net` - Un objeto de tipo `grain` en el que no se haya introducido ninguna evidencia
- `node` - Un string o indice indicando de que varaible hay que estimar la probabilidad marginal
- `evidence_nodes` - Un vector de strings o indices indicando a que variables se debe condicionar la marginal
- `evidences` - Un vector de igual tamaño al anterior que incluya los valores que deben tomar los nodos condicionantes
- `numSamples` - El número de muestras a utilizar en la estimación (m en el pseudocódigo de arriba)