Tutorial no. 2: Aprendizaje de Redes Bayesianas (I)

Borja Calvo, Aritz Pérez

Resumen

En este tutorial veremos como se pueden aprender redes Bayesianas a partir de datos utilizando `R`. Pese a que existen varios paquetes que permiten realizar esta tarea, en este tutorial nos centraremos en el paquete `bnlearn`, ya que es el más completo.

1 Preparación

En el primer tutorial de la serie ya instalamos los paquetes que necesitaremos en este, por lo que es suficiente con cargarlos. Además de los paquetes, en este tutorial usaremos una base de datos que viene con el paquete `bnlearn` , por lo que también tendremos que cargarla.

```
> library(bnlearn)
> library(Rgraphviz)
> library(ggplot2)
> data(learning.test)
```

2 Algoritmos de aprendizaje

El paquete `bnlearn` ofrece algoritmos de aprendizaje para redes Bayesianas tanto con variables discretas como continuas. Nosotros nos centraremos exclusivamente en el caso de variables discretes, ya que el caso continuo (también denominadas redes Gaussianas) no lo hemos tratado en la teoría.

2.1 Aprendizaje estructural

Dentro de los algoritmos disponibles para el aprendizaje estructural hay tanto algoritmos basados en restricciones como algoritmos de tipo *score + search*. En lo que respecta a los algoritmos basados en restricciones, el resultado no es necesariamente una estructura de red Bayesiana, sino un grafo que represente la clase de equivalencia. Por este motivo, en la red obtenida con este tipo de algoritmos podemos encontrar tanto arcos dirigidos como no dirigidos.

Los algoritmos basados en restricciones implementados en el paquete son los siguientes:

- Grow-shrink, implementado en la función `gs`
- Incremental association, implementado en la función `iamb`
- Una variante computacionalmente menos costosa del Incremental association, implementado en la función `fast.iamb`
- Interleaved incremental association, otra versión del Incremental Association implementada en la función `inter.iamb`

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)

- Max-min Parents and Children, implementado en la función `mmpc`

En todos los casos hay tres tipos de implementación. La primera de ellas, la opción por defecto, es una implementación optimizada. Si el parámetro `optimized` se fija a `FALSE`, la versión ejecutada es la original. Por último, utilizando el paquete `snow` es posible ejecutar los algoritmos en un cluster una versión paralelizada de los algoritmos. Para ello es necesario inicializar el cluster y pasar el objeto a través del parámetro `cluster`. Supongamos que queremos ejecutar el algoritmo grow-shrink, tenemos las siguientes opciones:

```
> net<-gs(learning.test) #Equivalente a ejecutar gs(data,optimized=T)
> net<-gs(learning.test,optimized=F)
> #cl debe ser un objeto de tipo cluster creado e inicializado usando snow
> net<-gs(learning.test,cluster=cl)
```

En lo que respecta al aprendizaje basado en una estrategia de tipo *score+search*, el paquete implementa tanto una busqueda local *greedy* de tipo *hill-climbing*, la cual es accesible a través de la función `hc`, como una búsqueda tabú, accesible a través de la función `tabu`.

En ambos casos podemos determinar cual es la red de partida sobre la que se aplicará la búsqueda, usando para ello el parámetro `start`. En el caso de la búsqueda tabú podemos determinar el tamaño de la 'memoria' utlizada a través del parametro `tabu`, que determina el tamaño de la lista tabú. Respecto a la búsqueda *greedy*, podemos evitar optimos locales utilizando reinicializaciones aletorias. Estas se controlan con dos parámetros, `restart`, que indica el número de reinicializaciones y `perturb`, que nos dice cuantos arcos se intentarán cambiar (aleatoriamente) en cada reinicialización. Los valores son, respectivamente, 0 y 1. Si los queremos cambiar por, por ejemplo 10 reinicializaciónes en las que se intenten cambiar 5 arcos, usaríamos la siguiente llamada:

```
> net<-hc(learning.test,restart=10,perturb=5)
```

En lo que se refiere a *scores*, el paquete `bnlearn` implementa las siguientes funciones (para el caso de datos discretos):

- Bayesianos: K2 (`k2`) y BDeu (`bde`)
- Basados en teoría de la información: Logverosimilitud (`loglik`), Akaike Information Criterion (`aic`) y Bayesian Information Criterion/Minimum Description Length (`bic`)

Tanto la función `hc` como la función `tabu` tienen un parametro (`score`) para determinar la función de evaluación a utilizar. Una vez aprendida la red, también es posible obtener su evaluación usando la función `score`, tal y como se muestra a continuación.

```
> net<-hc(learning.test)
> score(x=net,data=learning.test,type="loglik")

## [1] -23832.13

> score(x=net,data=learning.test,type="bic")

## [1] -24006.73

> score(x=net,data=learning.test,type="k2")

## [1] -23958.7

> score(x=net,data=learning.test,type="bde")

## [1] -23967.65
```

Para finalizar, cabe destacar que el paquete cuenta con un mecanismo para la definición de restricciones estructurales. Este mecanismo consiste en definir dos listas, denominadas *whitelist* y *blacklist*, las cuales se pueden pasar al algoritmo utilizando los parámetros de igual nombre.

El resultado (es decir, el grafo obtenido) puede ser visualizado usando la función `graphviz.plot`. Esta función permite realizar ciertos cambios en el grafo a dibujar. Particularmente interesantes son las opciones `layout`, que determina el algoritmo que se utilizará para colocar los nodos en el espacio, y la opción `shape`, que permite usar circulos o elipses para representar los nodos (estas últimas especialmente adecuadas si los nombres de las variables son largos). Una vez dibujado el grafo en pantalla, este se puede guardar usando la función `dev.print`. A continuación se muestra un ejemplo:

```
> graphviz.plot(x=net,layout="dot",shape="ellipse")
> dev.print(device=pdf,file="grafo.pdf")
> #En el caso de imagenes png, hay que determinar la dimensión en pixeles
> dev.print(device=png,width=1000,height=1000,file="grafo.png")
> dev.off()
```

2.2 Aprendizaje paramétrico

Las funciones presentadas anteriormente solamente generan la estructura, pero no los parámetros del modelo. Para completar la red Bayesiana con los parámteros tenemos que realizar el aprendizaje parametrico de forma explicita utilizando la función `bn.fit`:

```
> net.fitted<-bn.fit(net,data=learning.test)
> net.fitted[2]
```

Para la estimación paramétrica podemos optar por usar tanto estimadores maximoverosimiles como estimadores Bayesianos (`method="mle"` y `method="bayes"` respectivamente).

El objeto creado por esta función se corresponde a las tablas de probabilidad. Si queremos recuperar la red definida por dichas tablas debemos usar la función `bn.net`.

```
> graphviz.plot(bn.net(net.fitted))
```

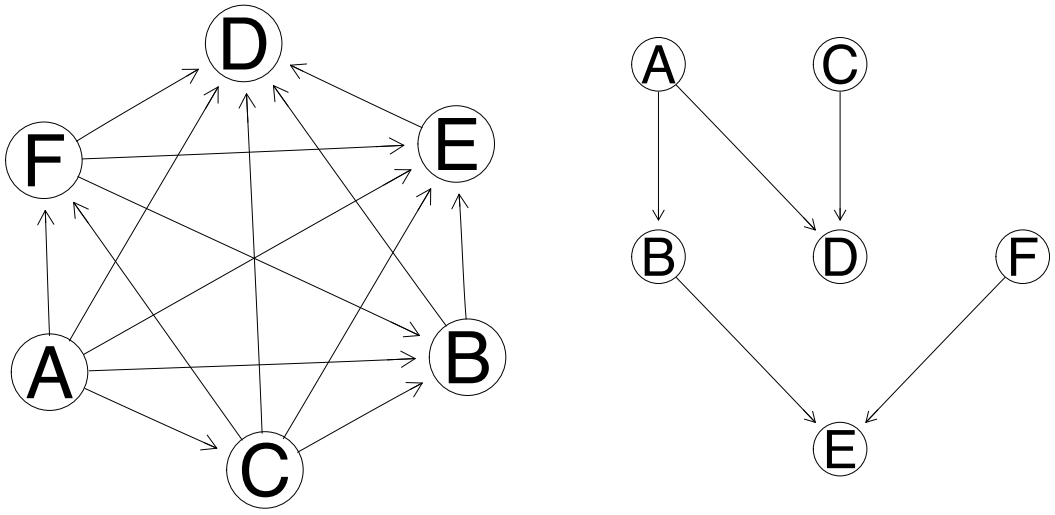
3 Inconvenientes del aprendizaje maximoverosimil

Como hemos visto en la teoría, guiar la búsqueda de la estructura por medio de la verosimilitud lleva a una estructura completa si no imponemos restricciones estructurales. Esto lo podemos ver facilmente con un ejemplo sencillo en el que compararemos la red aprendida con la log-verosimilitud y la aprendida con el score BIC.

```
> net_loglik<-hc(learning.test,score="loglik")
> net_bic<-hc(learning.test,score="bic")
> graphviz.plot(net_loglik,layout="neato")
> graphviz.plot(net_bic)
```

El resultado se muestra en la Figura 1. Como se puede apreciar, el número de arcos es mayor en el caso de la verosimilitud (15) que en el caso del score BIC (5). Esto se debe a que la verosimilitud es monótona creciente con respecto al número de arcos del modelo.

Esta comparación la podemos llevar al extremo incrementando el peso de penalización. Para ello, en lugar de usar los valores por defecto (1 para BIC), podemos pasar a la función el valor que queremos utilizar. A medida que aumentamos ese valor veremos que el número de arcos aprendidos disminuye.



(a) Logverosimilitud (b) BIC

Figura 1: Comparación entre el grafo generado con la log-verosimilitud y con el score BIC

```
> net<-hc(learning.test,score="bic",k=1)
> dim(arcs(net))[1]

## [1] 5

> net<-hc(learning.test,score="bic",k=100)
> dim(arcs(net))[1]

## [1] 4

> net<-hc(learning.test,score="bic",k=200)
> dim(arcs(net))[1]

## [1] 3

> net<-hc(learning.test,score="bic",k=1000)
> dim(arcs(net))[1]

## [1] 0
```

4 Capacidad de ajuste, de generalización y sobreajuste

La capacidad de ajuste de un modelo esta diréctamente relacionada con su complejidad, es decir, con su número de parámetros. El ajuste es el logaritmo de la verosimilitud que otorga un modelo al conjunto de datos empleado durante su entrenamiento, $LL(M = A(D); D)$. Esta cantidad se reduce lineálmente con el número de casos N y el número de variables n. Por tanto, para que la cantidad sea comparable para diferentes valores de N y n debe ser normalizada, $\frac{LL(A(D);D)}{n \cdot N}$.

Por otra parte la generalización es la verosimilitud (normalizada) que asigna un modelo a un conjunto de datos que no se ha empleado durante el entrenamiento, $\frac{LL(A(D);T)}{n \cdot N}$. La capacidad de generalización depende del número de parámetros y el número de casos del conjunto de entrenamiento. Cuando hay una gran discrepancia entre ajuste y la generalización se dice que el modelo sobreajusta. El sobreajuste se debe a un desequilibrio

entre el número de parámetros del modelo y el número de casos disponibles durante el entrenamiento, siendo el número de parámetros excesivamente alto para el número de casos disponibles.

A continuación analizaremos la capacidad de ajuste y de generalización de modelos de difernte complejidad. La sección incluye dos experimentos. En el primero estudiamos el ajuste y la generalización de un modelo en función de su número de parámetros y del número de casos empleados para realizar el aprendizaje paramétrico. Para ello se han empleado tres grafos asociados a modelos de diferentes complejidades: el grafo vacio, el grafo de cadena y el grafo completo. Los resultados obtenidos se muestran en la Figura 2. Las conclusiones que podemos extraer del estudio son las siguientes:

- El ajuste es mayor cuanto mayor es la complejidad del modelo. El ajuste disminuye con el número de casos empleados para realizar el aprendizaje paramétrico. En otras palabras, al modelo le resulta más dificil aprender (de memoria) el conjunto de entrenamiento conforme este último aumenta de tamaño.
- La generalización mejora con el número de casos empleados en el entrenamiento. Sin embargo, hay un límite a partir del cual deja de mejorar. Dicho limite esta dado por la complejidad del modelo, siendo los menos complejos los que antes se saturan.
- El sobreajuste se reduce conforme aumenta el número de casos debido a que la generalización y el ajuste tienden a tomar el mismo valor. El fenómeno del sobreajuste es mayor cuantos menos casos se disponen para realizar el aprendizaje paramétrico y cuanta mayor sea la complejidad del modelo.

```
> N<-dim(learning.test)[1]
> n<-dim(learning.test)[2]
> sizes<-round(exp(seq(1,log(2560),(log(2560)-1)/50)))
> train<- learning.test[1:2560,]
> test<- learning.test[2561:N,]
>
> # Definimos un modelo con el grafo vacio, con una cadena y con el grafo completo
> MEmpty<-empty.graph(c("A","B","C","D","E","F"))
> MChain<-empty.graph(c("A","B","C","D","E","F"))
> chain <- matrix(c("A", "B", "B", "C", "C", "D","D", "E", "E", "F"),
+                 ncol = 2, byrow = TRUE, dimnames = list(NULL, c("from", "to")))
> arcs(MChain) <- chain
> MComplete<-hc(train,score="loglik")
>
> # Funcion que evalua el ajuste y la generalización (normalizado) de una estructura
> # Los parametros se aprenden empleando los primeros "size" casos de "train"
> eval_struct<-function(size, BN, train, test){
+   D<-train[1:size,]
+   BN<- bn.fit(x= BN, data= D,method = "bayes")
+   gener=stats::logLik(BN,test)/(n*dim(test)[1])
+   fit=stats::logLik(BN,D)/(n*dim(D)[1])
+   c(gener,fit)
+ }
>
> # Para cada tamanio, obtenemos los valores
> resultados_Empty<-sapply(sizes, FUN=eval_struct, BN=MEmpty, train=train, test=test)
> resultados_Chain<-sapply(sizes, FUN=eval_struct, BN=MChain, train=train, test=test)
> resultados_Complete<-sapply(sizes, FUN=eval_struct, BN=MComplete, train=train, test=test)
>
> # Representacion grafica del resultado
> lims<- c(min(resultados_Empty,resultados_Chain),max(resultados_Empty,resultados_Chain))
> layout(matrix(c(1,2,3),ncol=3))
```

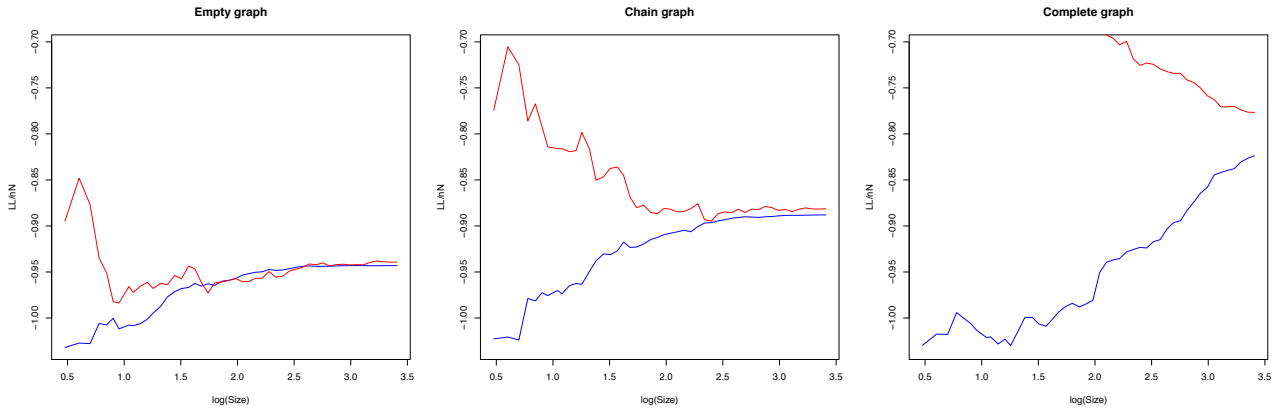


Figura 2: Evoluciones del ajuste (rojo) y generalización de modelos de creciente complejidad con respecto al número de casos empleados en el aprendizaje paramétrico.

```
> plot(log10(sizes), resultados_Empty[1,], main="Empty graph",
+      ylab= "LL/nN",xlab="log(Size)",type="l",ylim= lims, col= "blue")
> lines(log10(sizes),  resultados_Empty[2,],col="red")
>
> plot(log10(sizes),  resultados_Chain[1,], main="Chain graph",
+      ylab= "LL/nN",xlab="log(Size)",type="l",ylim= lims, col= "blue")
> lines(log10(sizes),  resultados_Chain[2,],col="red")
> plot(log10(sizes),  resultados_Complete[1,], main="Complete graph",
+      ylab= "LL/nN",xlab="log(Size)",type="l",ylim= lims, col= "blue")
> lines(log10(sizes),  resultados_Complete[2,],col="red")
```

En el segundo experimento analizamos las ventajas de emplear una función de evaluacion que penaliza los parametros con respecto a la log. verosimilitud (LL). Para ello emplearemos el algoritmo hill-climbing como heurístico de optimización y Bayesian infomation criteria (BIC) como función penalizada. Los resultados obtenidos se muestran el la Figura 3. Tal y como se ha ilustrado en la sección anterior la busqueda hill-climbing guiada por LL tiende a obtener grafos completos miestras que guiada por BIC limita el número de arcos añadidos debido a que se trata de una versión de la verosimilitud penalizada por la complejidad del modelo considerado, i.e. su número de parámetros. BIC regula la complejidad del modelo aprendido en función de los casos disponibles y, en consecuencia, muestra un mejor equilibrio entre la capacidad de ajuste y de generalización. De esta manera consigue reducir el efecto del sobreajuste que sufre el aprendizaje guiado por la verosimilitud, sin necesidad de imponer restricciones estructurales al modelo.

```
> # Funcion para, dado un tamanio de entrenamiento, obtener la verosimilitud en el train
> # y el test
> eval_net<-function(size, test=test,score="loglik"){
+   train<-learning.test[1:size,]
+   BN<- hc(train,score=score)
+   BN<- bn.fit(x= BN, data= train,method = "bayes")
+   gener=stats::logLik(BN,test)/(n*dim(test)[1])
+   fit=stats::logLik(BN,train)/(n*dim(train)[1])
+   c(gener,fit)
+ }
> # Para cada tamanio, obtenemos los valores
> resultados_loglik<-sapply(sizes, FUN=eval_net, test=test, score="loglik")
```

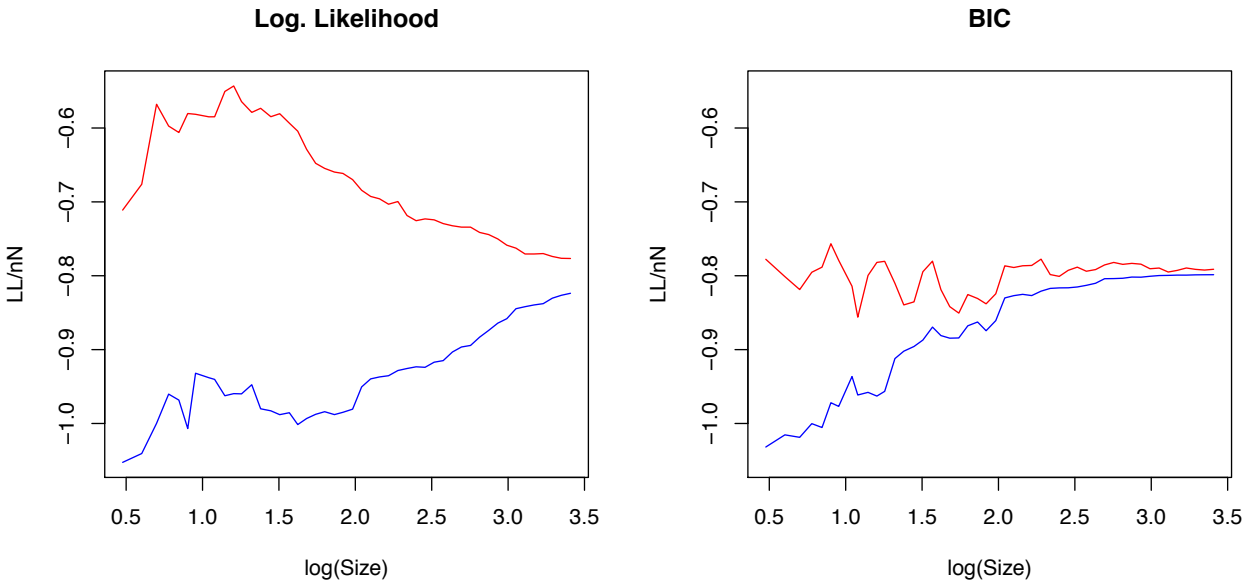


Figura 3: Comparación de la verosimilitud del conjunto de entrenamiento (en rojo) y el de test (en azul) para redes apredidas con la verosimilitud y el score BIC

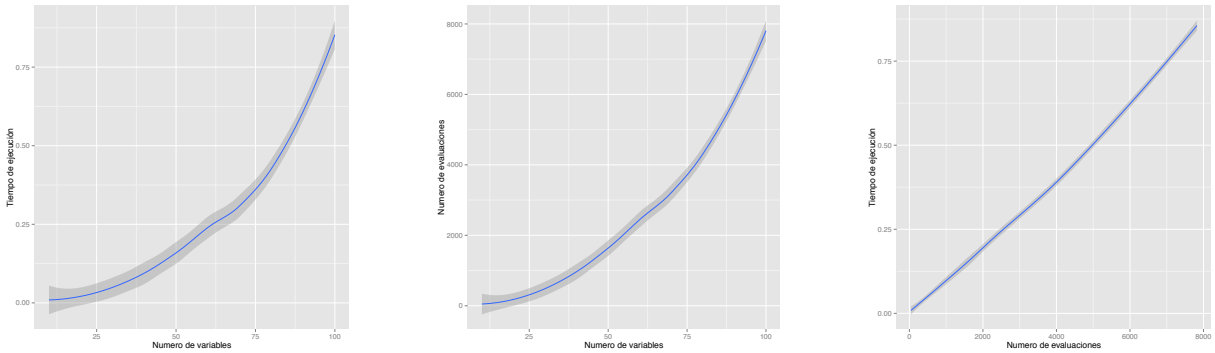
```
> resultados_bic<-sapply(sizes, FUN=eval_net, test=test, score="bic")
>
> # Representacion grafica del resultado
> lims<- c(min(resultados_bic,resultados_loglik),max(resultados_bic,resultados_loglik))
> layout(matrix(c(1,2),ncol=2))
> plot(log10(sizes), resultados_loglik[1,], main="Log. Likelihood",
+      ylab= "LL/nN",xlab="log(Size)",type="l",ylim= lims, col= "blue")
> lines(log10(sizes),  resultados_loglik[2,],col="red")
>
> plot(log10(sizes),  resultados_bic[1,], main="BIC", ylab= "LL/nN",xlab="log(Size)",
+      type="l",ylim= lims, col= "blue")
> lines(log10(sizes),  resultados_bic[2,],col="red")
```

5 Complejidad computacional del aprendizaje

Uno de los problemas del aprendizaje de redes Bayesianas es la complejidad computacional. Por ello, es fundamental entender el coste computacional de cada algoritmo y ver en que afecta cada parámetro.

Comenzaremos analizando el impacto del número de variables en el aprendizaje de una red por medio de técnicas de score + search. Para ello usaremos el algoritmo *greedy* junto con el score BDe. A fin de generar un problema complejo, crearemos una base de datos totalmente aleatoria (de esta manera no hay nada que aprender, más allá de las relaciones aleatorias que se generen en la muestra).

```
> numInst=100
> varSizeVector=seq(10,100,10)
> times<-vector()
> evaluations<-vector()
```



(a) Tiempo vs. no. de variables (b) Evaluaciones vs. no. de variables (c) Tiempo vs. Evaluaciones

Figura 4: Análisis empírico de la complejidad de las estrategias score+search en función del número de variables

```
> for (numVar in varSizeVector){
+   random_matrix<-data.frame(matrix(runif(numVar*numInst),ncol=numVar))
+   data<-discretize(random_matrix,method="interval",breaks=4)
+   t0<-proc.time()
+   net<-hc(data,score="bde",optimized=T)
+   t1<-proc.time()
+   times<-c(times,(t1-t0)[3])
+   evaluations<-c(evaluations,net$learning$ntests)
+ }
>
> df<-data.frame(vars=varSizeVector,time=times,eval=evaluations)
> ggplot(df,aes(x=vars,y=times)) + geom_smooth() +
+   labs(x="Numero de variables",y="Tiempo de ejecución")
> ggplot(df,aes(x=vars,y=eval)) + geom_smooth() +
+   labs(x="Numero de variables",y="Numero de evaluaciones")
> ggplot(df,aes(x=eval,y=times)) + geom_smooth() +
+   labs(x="Numero de evaluaciones",y="Tiempo de ejecución")
```

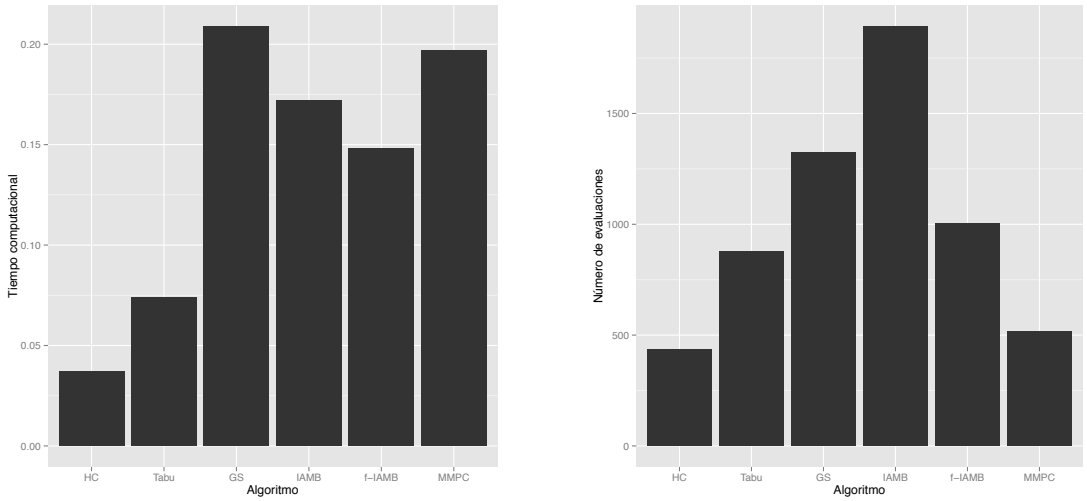
Definida una secuencia de número de variables, generamos datos al azar usando una distribución uniforme y los discretizamos. Para cada tamaño, obtenemos el tiempo necesario para aprender la red así como el número de evaluaciones requeridas. La figura 4 muestra el resultado.

Como puede apreciarse, el crecimiento tanto en tiempo como en número de evaluaciones es exponencial. De hecho, existe una relación lineal entre el número de evaluaciones y el tiempo, ya que el coste de evaluar la función no se ve afectado por el número de variables[1]. Como ejercicio queda probar que ocurre con otros algoritmos de aprendizaje.

A continuación analizaremos la complejidad de cada uno de los algoritmos disponibles.

```
> numInst=100
> numVar=30
> random_matrix<-data.frame(matrix(runif(numVar*numInst),ncol=numVar))
> data<-discretize(random_matrix,method="interval",breaks=4)
>
> get.cost<-function(data,alg,label){
+   t0<-proc.time()
```

[1]Esto tiene que ver con el hecho de que en la versión optimizada del algoritmo se usa la descomponibilidad del score, por lo que, independientemente del número de variables, actualizar un score implica el cambio de un solo arco



(a) Tiempo computacional

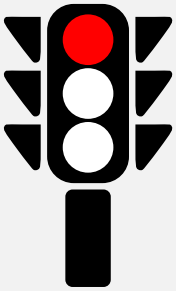
(b) Número de evaluaciones

Figura 5: Análisis empírico de la complejidad de diferentes algoritmos de aprendizaje de redes Bayesianas

```
+   net<-alg(data)
+   t1<-proc.time()
+   data.frame(alg=label,time=(t1-t0)[3],eval=net$learning$ntests)
+ }
>
> df<-rbind(get.cost(data,hc,"HC"),
+           get.cost(data,tabu,"Tabu"),
+           get.cost(data,gs,"GS"),
+           get.cost(data,iamb,"IAMB"),
+           get.cost(data,fast.iamb,"f-IAMB"),
+           get.cost(data,mmpc,"MMPC"))
>
> ggplot(df,aes(x=alg,y=time)) + geom_bar(stat="identity") +
+   labs(x="Algoritmo",y="Tiempo computacional")
> ggplot(df,aes(x=alg,y=eval)) + geom_bar(stat="identity") +
+   labs(x="Algoritmo",y="Número de evaluaciones")
```

El resultado de la comparación puede verse en la Figura 5. Como ejercicio queda la comparación con diferentes números de variables.

Ejercicio (opcional) no. 1 - Promediado de redes



Cuando tenemos una problema en el que hay muchas más variables que muestras pequeñas modificaciones de la base de datos pueden llevar a redes muy diferentes. Una forma de obtener resultados más robustos consiste en utilizar *bootstrapping* y *model averaging*.

El remuestreo boostrap consiste en, dada una base de datos formada por n variables y N instancias (casos), crear nuevas bases de datos de igual dimensión muestreando las instancias **con reemplazamiento**. El resultado son bases de datos en las cuales ciertas instancias aparecerán más de una vez y otras no apareceran.

Si utilizamos remuestreos bootstrap para apreder la estructura de la red, el resultado en cada remuestreo será, en general, diferente. Todas esas redes pueden ser combinadas para crear un nuevo grafo en el que solo se recojan los arcos que más veces se repiten. Es necesario controlar que no se produzcan ciclos cuando añadimos los arcos y para ello tendremos que fijar un orden ancestral o bien comprobar si el grafo construido es acíclico mediante la función `aciclic` del paquete `bnlearn` . La página web `http://www.bnlearn.com/examples/dag/` contiene información adicional para la edición de grafos.

Este reto consiste en crear una función en R que, dada una matriz de datos y un número de remuestreos (y tal vez otros parámetros), genere una red consenso (no necesariamente un DAG). Para llevar a cabo este reto se dispone de una función con la que generar remuestreos bootstrap.

```
remuestreo<-function(mat){
  num_samples<-dim(mat)[1]
  id<-sample(num_samples,replace=TRUE)
  mat[id,]
}
```

También conviene tener en cuenta que la función `amat` del paquete `bnlearn` permite obtener la matriz de adjacencia de una red. Así mismo, esta misma función puede ser utilizada para crear una red dada la matriz de adjacencia, tal y como se muestra en el ejemplo.

```
> names<-c("A","B","C","D")
> adj_mat<-matrix(c(0,1,1,0,0,0,0,0,0,0,0,1,0,0,0,0),byrow=T,ncol=4)
> colnames(adj_mat)<-names
> rownames(adj_mat)<-names
> graph<-empty.graph(names)
> amat(graph)<-adj_mat
> graphviz.plot(graph)
```