# Tutorial no. 4: Clasificadores basados en redes Bayesianas

Borja Calvo

**Resumen**

En este último tutorial de la series abordaremos un tipo de inferencia particularmente interesante en el caso de las redes Bayesianas: la predicción. Como veremos, el paquete `bnlearn` ya analizado en el tutorial no. 2 nos servirá para construir redes Bayesianas con una estructura especialmente pensada para la clasificación. Además de eso veremos de que manera otras funciones pueden ser utlizadas para construir clasificadores basados en redes Bayesianas.

## 1 Preparación

En este tutorial utilizaremos los dos paquetes básicos vistos hasta ahora, `bnlearn` y `gRain` :

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

Para el aprendizaje utilizaremos las bases de datos usadas en el tutorial no. 3, por lo que debermos cargarlas:

```
> car<-read.arff("car.arff")
> #Eliminamos las variables que solo tiene un estado
> mushroom<-read.arff("mushroom.arff")
> constantes<-which(lapply(mushroom,FUN=function(x){length(levels(x))})<2)
> mushroom<-mushroom[,-constantes]
```

## 2 Clasificadores básicos

En esta sección veremos como podemos aprender dos de los clasificadores basados en redes Bayesianas más básicos, el naive Bayes y el tree augmented naive Bayes (TAN). El paquete `bnlearn` incluye dos metodos, `naive.bayes` y `tree.bayes`, que nos permite construir este tipo de modelos de manera sencilla. Para ello basta con determinar que variable actuará como clase y que variables actuarán como variables predictoras. A continuación veremos como aprender un naive Bayes a partir de los datos `mushroom` y un TAN a partir de la base de datos `car`. En ambos casos la variable que actuará como supervisora (la clase) será la ultima (`class` tanto en `mushroom` como en `car`).

```
> nb_mush<-naive.bayes(training=names(mushroom)[dim(mushroom)[2]],
+                      explanatory=names(mushroom)[-dim(mushroom)[2]],x=mushroom)
> nb_mush_fitted<-bn.fit(x=nb_mush,data=mushroom,method="mle")
> nb_mush
```

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)

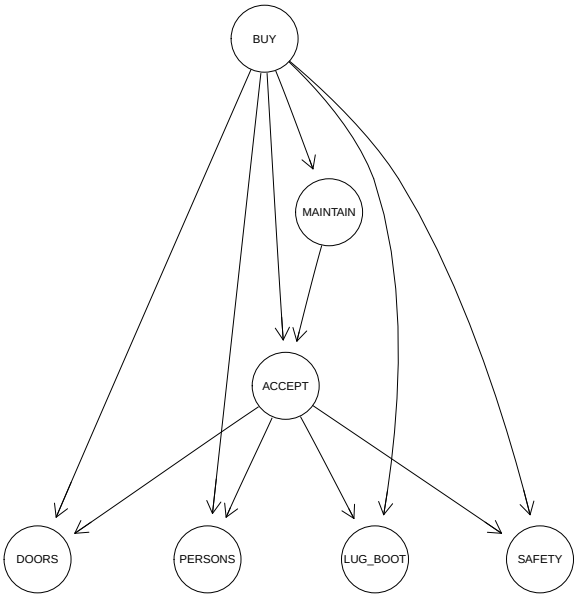



Figura 1: TAN aprendido con la base de datos `car`

```
> tan_car<-tree.bayes(x=car,training=names(car)[dim(car)[2]],explanatory=names(car)[-dim(car)[2]])
> tan_car_fitted<-bn.fit(x=tan_car,data=car,method="mle")
> tan_car
> graphviz.plot(tan_car)
```

Como se puede ver en el código, las funciones descritas realizan el aprendizaje estructural pero no el paramétrico. Este último paso lo realizamos con la función `bn.fit`.

La Figura 1 muestra el árbol aprendido en el caso de la base de datos `car`.

Una vez aprendido un clasificador podemos utilizar la función `predict` para predecir la clase de una nueva instancia. Veremos esta función en el siguiente apartado, dedicado a la evaluación de funciones de clasificación.

# 3 Evaluación de los clasificadores aprendidos

Existen diferentes formas de estimar el error de clasificación. La primera, más sencilla, es simplemente clasificar las instancias usadas en el aprendizaje. Este estimador, conocido como estimación por resubstitución, tiene como pega que no es honesto, ya que el clasificador es evaluado con las mismas instancias que se han usado en el aprendizaje y, como resultado, el error estimado es menor al error real (debido al sobreajuste). Para estimar el error por resubstitución podemos usar la función `predict`:

```
> estimacion_error<- function(clasificador, indice_clase, test){
+   clase_predicha<-predict(clasificador, test)
+   clase_real<-test[,indice_clase]
+   sum(clase_predicha!=clase_real) / length(clase_predicha)
+ }
>
> estimacion_error(clasificador= nb_mush_fitted, indice_clase=dim(mushroom)[2],
+                  test=mushroom)
```

```
## [1] 0

> estimacion_error(clasificador= tan_car_fitted, indice_clase=dim(car)[2],
+                  test=car)

## [1] 0.6550926
```

Como podemos ver, el error estimado en ambos casos es muy diferente. Esto es debido a que, por un lado, el en caso de `mushroom` solo hay dos posibles valores para la clase frente a los cuatro de `car` y, por otro lado, a que cada problema de clasificación es distinto y algunos son más sencillos que otros.

Una forma sencilla de realizar una estimación honesta es usando un esquema de tipo holdout. Para ello basta con dividir la base de datos en dos partes, una usada para entrenar y otra para evaluar.

```
> training<-0.66 #2/3 de base de datos para aprender, 1/3 para evaluar
> num_instancias<-dim(mushroom)[1]
> ##Barajeamos las instancias para quedarnos con los dos primeros tercios para prender
> permutation<-order(runif(num_instancias))
> id.train<-permutation[1:(training*num_instancias)]
> id.test<-permutation[(training*num_instancias+1):num_instancias]
>
> nb_mush_2<-naive.bayes(training=names(mushroom)[dim(mushroom)[2]],
+                        explanatory=names(mushroom)[-dim(mushroom)[2]],
+                        x=mushroom[id.train,])
> nb_mush_2_fitted<-bn.fit(nb_mush_2,mushroom[id.train,],"mle")
>
> estimacion_error(clasificador = nb_mush_2_fitted, indice_clase = dim(mushroom)[2],
+                  test = mushroom[id.test,])

## [1] 0.005681818

> num_instancias<-dim(car)[1]
> permutation<-order(runif(num_instancias))
> id.train<-permutation[1:(training*num_instancias)]
> id.test<-permutation[(training*num_instancias+1):num_instancias]
>
> tan_car_2<-tree.bayes(x=car[id.train,],
+                       training=names(car)[dim(car)[2]],
+                       explanatory=names(car)[-dim(car)[2]])
> tan_car_2_fitted<-bn.fit(x=tan_car_2,data=car[id.train,],method="mle")
> estimacion_error(clasificador = tan_car_2_fitted,indice_clase = dim(car)[2],
+                  test = car[id.test,])

## [1] 0.6678024
```

Por último, podemos evaluar los clasificadores usando para ello un esquema de tipo validación cruzada ($k$-fold cross validation). Este esquema está integrado en el paquete a través de la función `bn.cv`.

```
> bn.cv(data=car,bn=tan_car,loss="pred",k=5)

##
##   k-fold cross-validation for Bayesian networks
##
##   target network structure:
```

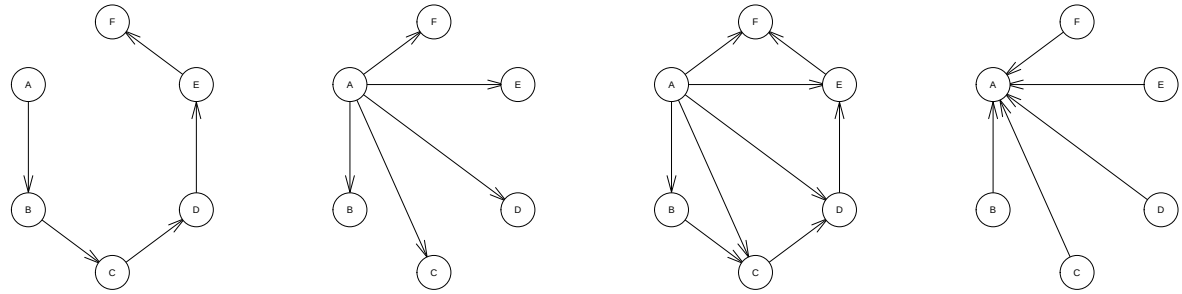



Figura 2: Estructuras consideradas en el experimento

```
##    [class][buying|class][maint|class:buying][safety|class:buying]
##    [persons|class:safety][lug_boot|class:safety][doors|class:lug_boot]
##   number of subsets:                    5
##   loss function:                        Classification Error
##   training node:                        class
##   expected loss:                        0.6666667
```

Como podemos ver, a la función le tenemos que pasar los datos, el modelo ya aprendido (aunque internamente se reaprenda en cada fold), el tipo de evaluación (`pred` para el error de clasificación) y la $k$ a utilizar, es decir, el número de folds en los que se dividirán los datos.

Una vez vistos y entendidos las formas en las que podemos evluar (de forma honest y no honesta) el error, veremos ahora como afecta la dimensionalidad de los datos al sobreajuste del modelo.

# 4 Estructuras sesgadas para la clasificación

Esta seccion recoge una serie de experimentos que ilustran la importancia de seleccionar una buena estructura que este orientada a los problemas de clasificacion. En la Figura 2 se muestran las estructuras seleccionadas. Las Figuras 4 y 3 resumen los resultados obtenidos en la experimentación.

```
> data<- learning.test
> vars<-names(data)
> N<- dim(data)[1]
> N_test<- dim(data)[1]/2
> N_train<-round(exp(seq(1,log(N-N_test),
+                        (log(N_test)-1)/50)))
>
> #Estructuras FIJAS:
> #Cadena
> chain.struc<-empty.graph(c("A","B","C","D","E","F"))
> arcs(chain.struc) <-  matrix(c("A","B","B", "C", "C","D","D", "E",
+                   "E", "F"), ncol = 2, byrow = TRUE,
+                 dimnames = list(NULL, c("from", "to")))
>
> #naive Bayes
> naive.struc<-empty.graph(c("A","B","C","D","E","F"))
```

```
> arcs(naive.struc) <-  matrix(c("A", "B", "A", "C", "A", "D","A", "E",
+                   "A", "F"), ncol = 2, byrow = TRUE,
+                 dimnames = list(NULL, c("from", "to")))
>
> #sabiondo Bayes
> sabiondo.struc<- empty.graph(vars)
> arcs(sabiondo.struc) <-  matrix(c("B", "A", "C", "A", "D", "A","E", "A",
+                   "F", "A"), ncol = 2, byrow = TRUE, dimnames =
+                     list(NULL, c("from", "to")))
>
>
> #tree augmented naive Bayes
> tree.struc<-empty.graph(vars)
> arcs(tree.struc) <-  matrix(c("A", "B", "A", "C", "A", "D","A", "E",
+                   "A", "F","B", "C","C","D","D", "E",
+                   "E", "F"), ncol = 2, byrow = TRUE, dimnames =
+                     list(NULL, c("from", "to")))
> #Regla de Bayes (implementada con la verosimilitud por eficiencia)
> Bayes_rule<- function(x,model,id_clase=1){
+   options(warn=-1)
+   omega_C<-levels(x[[id_clase]])
+   log_p= matrix(rep(0,dim(x)[1]*length(omega_C)),ncol=length(omega_C))
+   for(i in 1:length(omega_C)){
+     cl<-omega_C[i]
+     x[[id_clase]]<-factor(rep(cl,dim(x)[1]),levels=omega_C)
+     log_p[,i]= logLik(object = model, data= x, by.sample = T)
+   }
+   options(warn=0)
+   apply(log_p,MARGIN=1,FUN=function(x){id<-which(x==max(x));omega_C[id]})
+ }
>
> error<-function(data, model, id_clase=1){
+   sum(Bayes_rule(data,model)!=data[,id_clase])/dim(data)[1]
+ }
>
> res<-data.frame()
> num_rep<-20
> for(r in 1:num_rep){

+   for(s in unique(N_train)){
+       #Generar train y test
+       permutation<-order(runif(N))
+       id.train<-permutation[1:s]
+       id.test<-permutation[(N-N_test+1):N]
+       train<-data[id.train,]
+       test<-data[id.test,]

+       chain.model<- bn.fit(x= chain.struc, data= train, method = "bayes")
+       res<- rbind(res,data.frame("size_train"=s,
+                                  "error"=error(data= test,model = chain.model,id_clase = 1),
+                                  "data"="test","rep"=r, "estructura"="chain"))
+       res<- rbind(res,data.frame("size_train"=s,
```

```
+                                   "error"=error(data= train,model = chain.model,id_clase = 1),
+                                   "data"="train","rep"=r, "estructura"="chain"))

+       naive.model<- bn.fit(x= naive.struc, data= train, method = "bayes")
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= test,model = naive.model,id_clase = 1),
+                                   "data"="test","rep"=r, "estructura"="naive"))
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= train,model = naive.model,id_clase = 1),
+                                   "data"="train","rep"=r, "estructura"="naive"))

+       tree.model<- bn.fit(x= tree.struc, data= train, method = "bayes")
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= test,model = tree.model,id_clase = 1),
+                                   "data"="test","rep"=r, "estructura"="tree"))
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= train,model = tree.model,id_clase = 1),
+                                   "data"="train","rep"=r, "estructura"="tree"))

+       sabiondo.model<- bn.fit(x= sabiondo.struc, data= train, method = "bayes")
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= test,model = sabiondo.model,id_clase = 1),
+                                   "data"="test","rep"=r, "estructura"="sabiondo"))
+       res<- rbind(res,data.frame("size_train"=s,
+                                   "error"=error(data= train,model = sabiondo.model,id_clase = 1),
+                                   "data"="train","rep"=r, "estructura"="sabiondo"))
+   }
+ }
>
> ggplot(data = res,aes(x=size_train,y = error,
+                       col=data))+geom_line(stat="summary",
+                                            fun.y = "mean",size=1.1
+                                            )+scale_x_log10()+facet_wrap(~estructura)
> ggplot(data = res,aes(x=size_train,y = error,
+                       col=estructura))+geom_line(stat="summary",
+                                                  fun.y = "mean",size=1.1
+                                                  )+scale_x_log10()+facet_wrap(~data)
> layout(matrix(1:4,ncol=4,byrow=TRUE))
> plot(chain.struc)
> plot(naive.struc)
> plot(tree.struc)
> plot(sabiondo.struc)
```

## 5 Extendiendo los clasificadores básicos

En la teoría hemos visto que la idea básica del naive Bayes se puede extender de diferentes maneras. Una forma especialmente interesante de extender este tipo de clasificador es el llamado *selected naive Bayes*, cosistente en limitar las variables utilizadas a un subconjunto de ellas.

Una forma de seleccionar las variables es ver que ocurre con la variable clase en una red bayesiana genérica aprendida de los datos. En particular, una forma de seleccionar las variables relevantes para la clasificación es el llamado `Markov blanket` de la variable clase, ya que, conocidas esas varaibles, todas las demás son

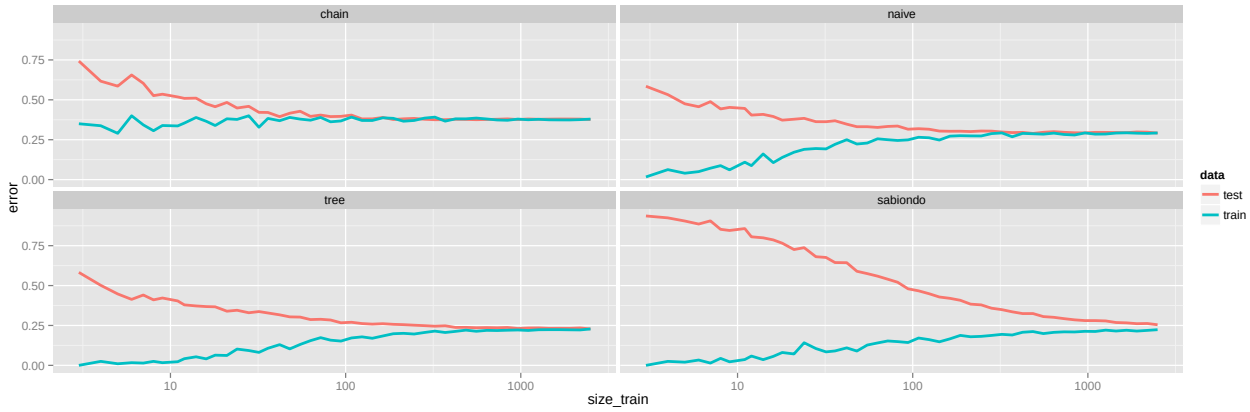



Figura 3: Resultados agrupados por estructura. En rojo se muestra el error cometido en el test que al ser de un tamaño sufientemente grande (2500 casos) se puede considerar como un buen estimador del error de cada modelo. En azul se muestra el error en el conjunto de entrenamiento y esta relacionado con el sobreajuste del modelo en terminos de error.

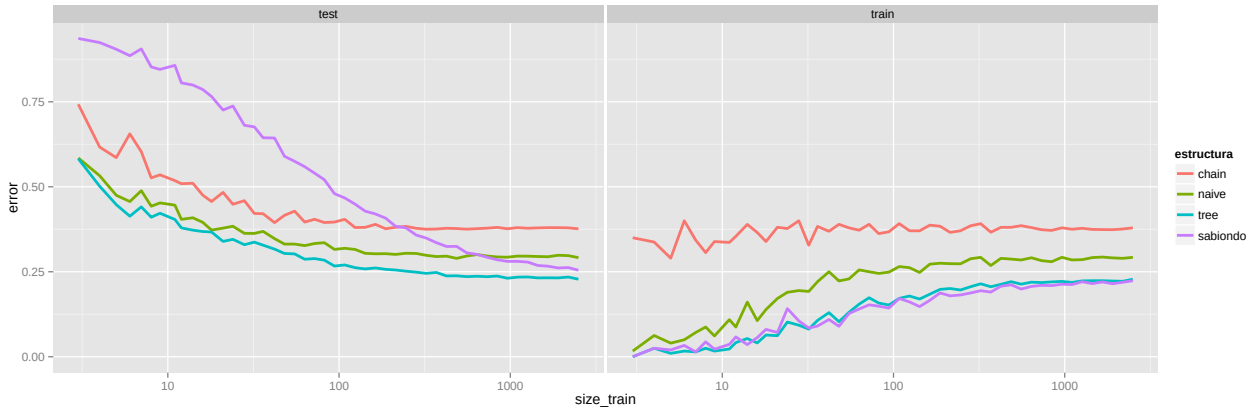



Figura 4: Resultados agrupados por el tipo de estimacion del error.

independientes de la varaible clase.

```
> selective_mb<-function (data,training){
+   net<-hc(data)
+   selected<-mb(net,training)
+   data_filtered<-data[,c(training,selected)]
+   model<-naive.bayes(training=training,explanatory=selected,x=data_filtered)
+   model_fitted<-bn.fit(model,data_filtered,"mle")
+   list(model=model_fitted,
+        features=selected,
+        training=training)
+ }
>
> predict_selective_mb<-function (model,data)
+ {
+   data_filtered<-data[,c(model$training,model$features)]
```

```
+   predict(model$model,data_filtered)
+ }
>
> selective_nb<-selective_mb(data=mushroom,training="class")
> selective_nb$features

## [1] "odor"      "gill-size" "habitat"

> #Holdout
> ratio<-0.66
> num_instancias<-dim(mushroom)[1]
> permutation<-order(runif(num_instancias))
> id.train<-permutation[1:(training*num_instancias)]
> id.test<-permutation[(training*num_instancias+1):num_instancias]
> snb<-selective_mb(data=mushroom[id.train,],training="class")
> sum(predict_selective_mb(snb,data=mushroom[id.test,])!=
+       mushroom[id.test,dim(mushroom)[2]])/length(mushroom[id.test,1])

## [1] 0
```

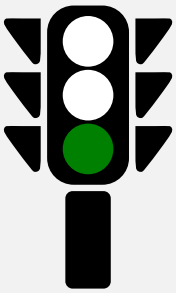
Como podemos ver, al aplicar esté clasificador a la base de datos `mushroom` únicamente 12 de las 21 variables predictoras son utilizadas para realizar la clasificación.

Para concluir con este tutorial veremos otra forma de clasificar, consistente en utlizar una red Bayesiana genérica. En las estructuras típicas usadas para clasificación la variable clase no tiene padres y es a su vez padre de todas las demás variables. Esto permite realizar la predicción de forma muy eficiente.

```
> general_bn<-function (data,training,explanatory,net=NULL){
+   if(is.null(net)) net<-hc(data)
+   graph<-as(amat(net),"graphNEL")
+   model<-grain(graph,data=data,smooth=1/dim(data)[1])
+   list(model=model,explanatory=explanatory,training=training)
+ }
>
> predict_general_instance<-function (x,model){
+   setEvidence(model$model,nodes=model$explanatory,states=x[model$explanatory])
+   prob<-querygrain(model$model,nodes=model$training,type="marginal")
+   m<-max(prob[[1]])
+   id<-prob[[1]]==m
+   names(prob[[1]])[id]
+ }
>
> predict_general<-function (model,data){
+   apply(data,MARGIN=1,FUN=predict_general_instance,model=model)
+ }
>
> ratio<-0.66
> num_instancias<-dim(mushroom)[1]
> permutation<-order(runif(num_instancias))
> id.train<-permutation[1:(ratio*num_instancias)]
> id.test<-permutation[(ratio*num_instancias+1):num_instancias]
> gbn<-general_bn(data=mushroom[id.train,],
+                 training=names(mushroom)[dim(mushroom)[2]],
+                 explanatory=names(mushroom)[-dim(mushroom)[2]])
```

```
## extractCPT - data.frame

> sum(predict_general(gbn,data=mushroom[id.test,])!=
+       mushroom[id.test,dim(mushroom)[2]])/length(mushroom[id.test,1])

## [1] 0.1079545

> sum(predict_selective_mb(snb,data=mushroom[id.test,])!=
+       mushroom[id.test,dim(mushroom)[2]])/length(mushroom[id.test,1])

## [1] 0
```

## Ejercicio (optional) no. 3 - Selective naive Bayes basado en test de independencia



En el tutorial hemos visto como construir un selective naive Bayes usando el Markov blanket para seleccionar las variables de interés. Otra alternativa a la selección de variables es utilizar test estadísticos.
Estos test se pueden aplicar usando la función `ci.test` del paquete `bnlearn` . Por ejemplo, si queremos ver si las dos primeras variables en la base de datos `car` son o no independientes podemos ejecutar:

```
> test<-ci.test(x=names(car)[1],y=names(car)[2],data=car)
> test

##
##  Mutual Information (disc.)
##
## data:  buying ~ maint
## mi = 0, df = 9, p-value = 1
## alternative hypothesis: true value is greater than 0

> test$p.value

## [1] 1
```

El objetivo de este reto es construir una función, similar a la función `selective_mb` implementada en la Sección 5, pero en la cual las variables se seleccionen verificando si son o no independientes de la variable clase utilizando para ello un test de independencia. La función deberá incluir un parámetro extra, el nivel de confianza a la hora de determinar si una variable es o no independiente de la clase.
Para ilustrar el uso de esta función, escribir el código para usarla con la base de datos `mushroom` y comprobar con un holdout cual es el error cometido.