BAGGING - RANDOM FORESTS

(ENSEMBLE LEARNING I)

PRÁCTICA

Paquetes, funciones y datos

Los paquetes

`ada, adabag, ipred, randomForest`

son los específicos del tema. Además, usaremos de temas anteriores los:

`tree, rpart, e1071, class, MASS.`

Para conjuntos de datos, además de los anteriores, usaremos:

`ElemStatLearn,mlbench`

Finalmente, alguno de los anteriores usa como auxiliares los:

`prodlim, KernSmooth, survival, splines, caret, lattice, ggplot2.`

Otras funciones

Del paquete `stats`, para comparación, la función: `glm`

Del paquete `MASS`, para comparación, el discriminador lineal: `lda`

Documentación de métodos y paquetes

Además de los help de las funciones, hay las vignettes de los paquetes, y los artículos:

- adabag: Alfaro, E., Gómez, M., García, N. (2013), adabag: An R Package for Classification with Boosting and Bagging, Journal of Statistical Software, 54.

- Random Forests: Breiman, L . , Cutler, A ., Random Forests (notas de uso) y de hecho toda la página Leo Breiman - Random Forests, en la que hay enlace a varios documentos y tutoriales.

Datos

Datos de ficheros externos (del Campus Virtual):

- DNA.splice.1.txt

En el paquete datasets, de la instalación básica de R:

- Datos de las flores iris: `data(iris)`.
- Datos de supervivientes del naufragio del Titanic: `data(Titanic)`.
- Datos de calidad del aire: data(airquality)
- Datos de fertilidad e indicadores socioeconómicos en Suiza: data(swiss)

En el paquete MASS. Instalar con: `library(MASS)`

- Datos sintéticos: `data(synth.tr), data(synth.te)`.

  Convertir la variable `yc` en un factor (viene codificada con los valores numéricos 0/1) para que el programa la pueda tratar como una etiqueta para clasificar:

  ```
  synth.tr$yc<-factor(synth.tr$yc)

  synth.te$yc<-factor(synth.te$yc)
  ```

- Características y comportamiento de CPU's de ordenadores: `data(cpus)`.

  Eliminar la primera variable, que es la etiqueta de la marca. Si se desea, puede conservarse como nombre de las filas. Eliminar también la última variable, que es una predicción de la variable respuesta hecha por los autores del artículo original:

  ```
  labels<-cpus[,1]
  cpus<-cpus[,2:8]
  rownames(cpus)<-as.character(labels)
  ```

  Una transformación logarítmica de estos datos:

  ```
  logcpus<-cpus[,1:7]
  logcpus$logperf<-log10(cpus$perf)
  ```

- Fragmentos de vidrio recogidos en trabajos forenses: `data(fgl)`.

- Datos clásicos de color de ojos y del pelo en 5387 niños de Caithness (Escocia): `data(caith)`.

En el paquete `ElemStatLearn`. Instalar con: `library(ElemStatLearn).`

- Datos de enfermedad coronaria: `data(SAheart)`

  Convertir la variable chd en un factor (viene codificada con los valores numéricos 0/1):

  ```
  SAheart$chd<-factor(SAheart$chd)
  ```

  Convertir la variable `famhist` a numérica (viene como un factor). Esto no es estrictamente necesario para árboles de predicción, pero sí para comparar resultados con otras técnicas.

  ```
  SAheart$famhist<-as.numeric(SAheart$famhist).
  ```

- Datos de email (distinguir spam): `data(spam).`

- Datos de OCR: `data(zip.train), data(zip.test)`.

- Datos vowel: `data(vowel.train), data(vowel.test)`

  Convertir en factor la respuesta:

  ```
  vowel.train$y<-factor(vowel.train$y)
  vowel.test$y<-factor(vowel.test$y)
  ```

En el paquete `mlbench`. Instalar con: `library(mlbench)`

- Datos de cáncer de mama: `data(BreastCancer)`

En el paquete `ipred`.

- Datos de diagnóstico de glaucoma: `data(GlaucomaM)`

En el paquete `rpart:`

- Datos de cancer de próstata en fase C: `data(stagec).`

  Estos datos `stagec` son, en realidad, datos para el análisis de supervivencia. La primera variable, `pgtime` es el tiempo de supervivencia para los pacientes que progresan. La segunda variable, `pgstat`, es el indicador de progreso del paciente (1 = "progresa", 0 = "censurado"). Además, hay observaciones con alguna variable missing, Ver en el script `stagec.r` la preparación de estos datos.

Bagging

Con el paquete adabag

El paquete adabag usa solamente CART, mediante `rpart`, como predictor elemental:

```
require(adabag)
require(rpart)
source("prob.err.r")
```

```
data(iris)
names(iris)<-c("LS","AS","LP","AP","Species")
Itrain<-
c(sample(1:50,25),sample(51:100,25),sample(101:150,25))
iris.bagging<-
bagging(Species~.,data=iris[Itrain,],mfinal=10)
```

El parámetro `mfinal` es el número de remuestras bootstrap.

Probad con distintos valores, observando las matrices de confusión.

Atención a los valores demasiado grandes, que pueden llenar la memoria y/o emplear tiempo excesivo.

```
iris.bagging.pred<-predict.bagging(iris.bagging,
                                   newdata=iris[-Itrain,])
iris.bagging.conf<-iris.bagging.pred$confusion
prob.err(iris.bagging.conf)
```

Los datos `iris` son de clasificación fácil, pero ¿Qué ocurre con unos datos notoriamente difíciles, como los `vowel`?

```
require(ElemStatLearn)
data(vowel.train)
data(vowel.test)
vowel.train$y<-factor(vowel.train$y)
vowel.test$y<-factor(vowel.test$y)
vowel.bagging <- bagging(y~ ., data=vowel.train,
mfinal=10)

vowel.bagging.pred<-predict.bagging(vowel.bagging,
                                    newdata=vowel.test)
vowel.bagging.conf<-vowel.bagging.pred$confusion
```

Atención! la matriz de confusión producida por `predict.bagging()` es transpuesta de la que se suele emplear (los valores conocidos aparecen aquí en las columnas).

Además, con estos datos `vowel`, que tienen como respuesta un factor con 11 niveles, la función `predict.bagging()` tiene un error en el formato de presentación de la matriz de confusión, cuyas filas aparecen en el orden: 1,10,11,2,3,4,5,6,7,8,9.

Esto se puede compensar haciendo:

```
Icorrect<-c(1,4,5,6,7,8,9,10,11,2,3)
vowel.bagging.conf<-vowel.bagging.conf[Icorrect,]
prob.err(vowel.bagging.conf)
```

Una vez hecha esta corrección, la estimación de la probabilidad de error, cociente entre la suma de los elementos no diagonales de la matriz de confusión y el total, coincide con el valor producido (correctamente) por `predict.bagging()`

```
vowel.bagging.pred$error
```

En este paquete hay una función errorev(), que calcula el error para cada valor del número de árboles generados. Así se puede evitar un modelo innecesariamente grande:

```
vowel.bagging <- bagging(y~ ., data=vowel.train,
mfinal=100)
vowel.bagging.errev<-errorevol(vowel.bagging,vowel.test)
plot(vowel.bagging.errev$error, type="l",
        main="Bagging error Vs number of trees",
        col="blue")
```

En vez de subdividir los datos en dos partes, muestra de entrenamiento y muestra de test, se pueden juntar las dos partes y usar la función `bagging.cv()` para hacer validación cruzada con varios subconjuntos.

Con el paquete ipred

Atención! Este paquete no debe emplearse en una misma sesión a continuación del `adabag`, pues al tener también una función `bagging()`,

interfiere con la de igual nombre que ya tenemos activa. Se puede reiniciar R o hacer limpieza de memoria con:

```
rm(list = ls())
```

Otra observación es que la operación de cargar `ipred` requiere cargar también los paquetes `MASS, survival, splines, nnet, class, prodlim, KernSmooth`, que debemos instalar, aunque no los necesitemos todos. Estos paquetes suministran un repertorio de máquinas de predicción elementales a los que se puede someter al mecanismo bagging.

```
require(ipred)
require(rpart)
data(GlaucomaM)
GlaucomaM.bagging.1<-bagging(Class~.,data=GlaucomaM,
         nbagg=25,coob=TRUE)
print(GlaucomaM.bagging.1)
```

Aquí el parámetro `nbagg` es el número de remuestras bootstrap.

Con la opción `coob=TRUE` obtenemos la estimación OOB *(Out-of-Bag)* de la probabilidad de mala clasificación. No es necesario dividir la muestra en una porción de entrenamiento y otra de prueba para tener esta estimación.

Random forests

Hay varias implementaciones de Random Forests para el entorno R, algunas de ellas especializadas a clases concretas de datos, pero la de referencia es en el package `randomForest`.

Aplicando a los datos iris:

```
require(randomForest)
require(rpart)
data(iris)

iris.rf <- randomForest(Species ~ ., data=iris,
                 importance=TRUE, proximity=TRUE)
```

Haciendo `print(iris.rf)` se obtiene información básica de los árboles generados, la estimación OOB de la probabilidad de clasificación errónea y la matriz de confusión.

El resultado de `plot(iris.rf)` es un gráfico de la estimación OOB del error de clasificación, con cuatro componentes, la primera es el error promedio y las otras tres, individualmente para cada clase.

La función `margin()` retorna el margen para cada observación de los datos, una medida de la firmeza de la asignación, correcta o errónea, de la observación. Formalmente se define como la proporción de los votos procedentes de todos los árboles generados que han ido a la clase correcta menos el máximo de las proporciones de votos que han ido a las otras clases. Puede variar entre -1 (observación mal clasificada, con todos los votos a una clase errónea) y 1 (observación bien clasificada, con todos los votos a la clase correcta). El método `plot()` de esta clase, `plot(margin(iris.rf))`, produce un gráfico de los márgenes.

La función `importance()` retorna los valores de importancia relativa de cada variable predictora para la predicción. En una clasificación se desglosa para cada clase. La función `varImpPlot()` produce un diagrama de estos valores.

```
varImpPlot(iris.rf,col=c(1,2,3,4))
```

Random Forest permite obtener una matriz de proximidades entre las observaciones. Su elemento (i,j) es la proporción de los árboles en los que las observaciones i y j quedan en un mismo nodo terminal. Esta matriz puede usarse como input a un MDS para visualizar alguna estructura de los datos. El gráfico se obtiene directamente con:

```
MDSplot(iris.rf,iris$Species,palette=rep(1,3))
```

El fichero de datos `DNA.splice.1.txt` contiene los datos de DNA cuyo estudio se describe en la referencia Breiman, Cutler dada más arriba. Hay 3175 observaciones, en vez de los 3186 que se mencionan en este texto pues en la preparación he eliminado los casos con algún valor missing en los datos. Podéis repetir con este problema las operaciones hechas con los datos iris.