PEC 3: Secuencias promotoras de E.coli. Predicción con algoritmos de Machine Learning

Pablo Iriso Soret

Contents

13 de enero, 2021

Fundamento teórico. Promotores

En genética, un promotor es una secuencia de ADN necesaria para convertir un gen en activado o desactivado. Esta región controla la iniciación de la transcripción de una determinada porción del ADN a ARN, pudiendo este ARN posteriormente codificar una proteína, o tener una función en sí mismo, como ARNt, ARNm o ARNr. Los promotores se encuentran cerca de los sitios de inicio de la transcripción de los genes, corriente arriba en el ADN (hacia la región 5‘). (Sharan, Karni, and Felder 2007)

Carga y Exploración de los datos

```
file1 <- "promoters.txt"
file2 <-"promoters_onehot.txt"

promoters <- read.table(file1, sep = ",")
colnames(promoters) <- c("class", "name", "seq")
promoters_onehot<- read.table(file2, header = TRUE)
str(promoters)
```

```
'data.frame':   105 obs. of  3 variables:
 $ class: chr  "+" "+" "+" "+" ...
 $ name : chr  "S10" "AMPC" "AROH" "DEOP2" ...
 $ seq  : chr  "tactagcaatacgcttgcgttcggtggttaagtatgtataatgcgcgggcttgtcgt" "tgctatcctgacagttgtcacgctgat
```

Los atributos del fichero de datos son:

1. Un símbolo de {+/-}, indicando la clase (“+” = promotor).
2. El nombre de la secuencia promotora. Las instancias que corresponden a no promotores se denominan por la posición genómica.
3. Las restantes 57 posiciones corresponden a la secuencia.

One-hot encoding

A la hora de implementar algoritmos de machine learning es fundamental elegir una representación de los datos que sea fidedigna y permita una ejecución precisa del algoritmo. En este caso, en el que trabajamos con secuencias, hemos escogido la representación one-hot encoding. En circuitos digitales y machine learning se denomina “one-hot” al grupo de bits en el cual la combinación de valores está definida por un valor alto (1) y el resto de valores bajo (0) (Harris and Harris 2010). Una implementación similar en la cual todos los bits son un “1” excepto un valor que es un “0” es llamada en ocasiones “one-cold” (Harrag and Gueliani 2020).

En esta transformación representaremos cada nucleótido por una combinación de ceros y unos, más concretamente un uno y tres ceros, en el que la posición del 1 definirá el nucleótido. Pongamos por ejemplo, el nucleótido T se representa por (1,0,0,0), el nucleótido C por (0,1,0,0), el nucleótido G por (0,0,1,0) y el nucleótido A por (0,0,0,1). A continuación vamos a explicar detalladamente la función de transformación.

```
#Para crear esta función, definiremos una combinación de bits para cada base nucleotídica,
#y haremos una asociación automática para la secuencia introduica
t <- c(1,0,0,0)
c <- c(0,1,0,0)
```

```
g <- c(0,0,1,0)
a <- c(0,0,0,1)
sigla <- c("t","c","g","a")

#Creamos una matriz con la información de las bases y la convertimos en un dataframe
bases <- rbind(t,c,g,a)
bases <- cbind(sigla, bases)
bases <- as.data.frame(bases)


#Creamos una función que nos proporciona una secuencia en bits
#a partir de una secuencia de caracteres con las bases
cod_o <- function(sequence){
  unlist(asplit(bases[na.omit(match(strsplit(sequence, "")[[1]], bases$sigla)), -1], 1),
         use.names = FALSE)
}

#Unimos esta función a una nueva función que es capaz de iterar dicha función,
#creando un data frame con la secuencia nucleotidica y su correspondiente secuencia de bits
generador <- function(datos){
  bits <- t(as.data.frame(lapply(datos$seq, cod_o)))
  row.names(bits) <- datos$seq
  bits <- as.data.frame(bits)
  return(bits)
}
```

A continuación aplicamos esta función sobre la columna de valores. Por comodidad llamaremos `datos` al dataframe `promoters_onehot` generado por nuestra función.

```
seq <- promoters$seq
seq <- as.data.frame(seq)
promoters_onehotr <-generador(seq)

# A continuación unimos las secuencias bits con las columnas nombre y clase
datos <- promoters %>%
  subset(select = c("class","name")) %>%
  cbind(promoters_onehotr)
```

Agrupación en `train` y `test`

Vamos a dividir los datos en dos partes. Una primera parte compuesta por 2/3 de los datos que utilizaremos como entrenamiento y que llamaremos `train`, y una segunda compuesta por 1/3 de los datos que llamaremos `test` y utilizaremos para cuantificar la calidad de nuestro algoritmo. Esta partición vendrá precedida por un reordenamiento aletorio de los datos (`set.seed(123)`). Convertiremos además la variable a predecir en codificación binaria.

```
set.seed(123)
#Para ordenarlo de forma aleatoria creamos una distribución aleatoria de todas las columnas
# Eliminamos la columna con los nombres
datos <- datos[-2]
# Convertimos en codificacion binaria el etiquetado
datos$class[datos$class == "+"] <- 1
```

```r
datos$class[datos$class == "-"] <- 0
datos$class <- as.factor(datos$class)

rows <- sample(nrow(datos))
datos_n <- datos[rows, ]

#La expresión round(0.67*nrow(ort)), nos proporciona el elemento situado en la
# posición correspondiente al 67%
index <- round(0.67*nrow(datos_n))
train <- datos_n[1:index, ]
test <- datos_n[index:length(datos_n$class),]
#Las etiquetas se asocian con el tipo de promotor
train_labels <- datos_n[1:index, 1]
test_labels <- datos_n[index:length(datos_n$class), 1]
```

K-Nearest Neighbors

El algoritmo K-nearest neighbors (kNN - k vecinos más cercanos) se encuentra entre las diez técnicas más empleadas en el *data mining*. Este método utiliza el principio de Ciceron *"pares cum paribus facillime congregantur"* (pájaros del mismo plumaje vuelan juntos o, literalmente, iguales con iguales se asocian fácilmente), y pertenece al conjunto de algoritmos de clasificación.

Un algoritmo de clasificación permite la identificación de la categoría a la que pertenece un objeto concreto. De tal forma que, a partir de un conjunto de observaciones $(x_1, y_1), ...(x_n, y_n)$, donde y es la categoría a la que pertenece la muestra, y x, un vector de características correspondientes a dicha muestra, los algoritmos de clasificación nos van a permitir determinar dicha categoría para un objeto del que poseamos únicamente sus caracterítsicas. (Kramer 2013)

Para este método concreto, dicha asignación se basará en la cercanía de nuestro nuevo objeto, al resto de observaciones más cercanas, y le otorga una clase basado en la mayoría de los datos que le rodean (y las clases que presentan dichos datos). Será por lo tanto necesario precisar, en primer lugar, que valor de k vamos a escoger, es decir, en base a que número de vecinos próximos estableceremos la clasificación, y en segundo lugar, como vamos a representar computacionalmente dichas distancias.

El algoritmo k-NN se denomina habitualmente como clasificador "vago", dado que técnicamente no genera un clasificador a partir de los datos de entrenamiento, si no que cada vez que quiere asignar una clase a un nuevo objeto, calcula las distancias para dicho objeto a partir de los datos de entrenamiento. Esto provoca que sea costoso computacionalmente. (Mucherino, Papajorgji, and Pardalos 2009)

Algoritmo kNN basico:

- **Input** : Presenta los siguientes componentes, D, el set de entrenamiento, formado por un conjunto de objetos; z, que es el objeto al que queremos asignar una clase y viene definido por un vector de valores; y L, el conjunto de clases para los objetos.
- **Output**: $c_z \in L$, la clase de z.
  (Mucherino, Papajorgji, and Pardalos 2009)

Entrenando el modelo

A continuación vamos a implementar el algoritmo knn con diferentes valores de k (k = 3, 5, 7, 11) para predecir que secuencias presentan acción promotora.

K = 1

```
knn_1 <- knn(train=train, test=test, cl=train_labels, k=1)
p_knn1 <- table(knn_1)
prop_nc1 <- p_knn1[1]
prop_c1 <- p_knn1[2]
```

K = 3

```
knn_3 <- knn(train=train, test=test, cl=train_labels, k=3)
p_knn3 <- table(knn_3)
prop_nc3 <- p_knn3[1]
prop_c3 <- p_knn3[2]
```

K = 5

```r
knn_5 <- knn(train=train, test=test, cl=train_labels, k=5)
p_knn5 <- table(knn_5)
prop_nc5 <- p_knn5[1]
prop_c5 <- p_knn5[2]
```

K = 7

```r
knn_7 <- knn(train=train, test=test, cl=train_labels, k=7)
p_knn7 <- table(knn_7)
prop_nc7 <- p_knn7[1]
prop_c7 <- p_knn7[2]
```

Predicción y evaluación

Evaluación del modelo

```r
confusionMatrix(reference =  test_labels, data = knn_1)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 11  1
         1  7 17

               Accuracy : 0.7778
                 95% CI : (0.6085, 0.8988)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 0.0005966

                  Kappa : 0.5556

 Mcnemar's Test P-Value : 0.0770999

            Sensitivity : 0.6111
            Specificity : 0.9444
         Pos Pred Value : 0.9167
         Neg Pred Value : 0.7083
             Prevalence : 0.5000
         Detection Rate : 0.3056
   Detection Prevalence : 0.3333
      Balanced Accuracy : 0.7778

       'Positive' Class : 0
```

```r
confusionMatrix(reference =  test_labels, data = knn_3)
```

```
Confusion Matrix and Statistics

          Reference
```

```
Prediction  0  1
         0 13  1
         1  5 17

               Accuracy : 0.8333
                 95% CI : (0.6719, 0.9363)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 3.48e-05

                  Kappa : 0.6667

 Mcnemar's Test P-Value : 0.2207

            Sensitivity : 0.7222
            Specificity : 0.9444
         Pos Pred Value : 0.9286
         Neg Pred Value : 0.7727
             Prevalence : 0.5000
         Detection Rate : 0.3611
   Detection Prevalence : 0.3889
      Balanced Accuracy : 0.8333

       'Positive' Class : 0
```

```
confusionMatrix(reference =  test_labels, data = knn_5)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 13  2
         1  5 16

               Accuracy : 0.8056
                 95% CI : (0.6398, 0.9181)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 0.0001563

                  Kappa : 0.6111

 Mcnemar's Test P-Value : 0.4496918

            Sensitivity : 0.7222
            Specificity : 0.8889
         Pos Pred Value : 0.8667
         Neg Pred Value : 0.7619
             Prevalence : 0.5000
         Detection Rate : 0.3611
   Detection Prevalence : 0.4167
      Balanced Accuracy : 0.8056

       'Positive' Class : 0
```

```r
confusionMatrix(reference =  test_labels, data = knn_7)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 13  1
         1  5 17

               Accuracy : 0.8333
                 95% CI : (0.6719, 0.9363)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 3.48e-05

                  Kappa : 0.6667

 Mcnemar's Test P-Value : 0.2207

            Sensitivity : 0.7222
            Specificity : 0.9444
         Pos Pred Value : 0.9286
         Neg Pred Value : 0.7727
             Prevalence : 0.5000
         Detection Rate : 0.3611
   Detection Prevalence : 0.3889
      Balanced Accuracy : 0.8333

       'Positive' Class : 0
```

```r
kappaknn1 <- Kappa(table(test_labels, knn_1))
kappaknn3 <- Kappa(table(test_labels, knn_3))
kappaknn5 <- Kappa(table(test_labels, knn_5))
kappaknn7 <- Kappa(table(test_labels, knn_7))
kappaknn7
```

```
            value    ASE     z  Pr(>|z|)
Unweighted 0.6667 0.1211 5.504 3.709e-08
Weighted   0.6667 0.1211 5.504 3.709e-08
```

K escogida	Precisión	Coeficiente Kappa de Cohen
K=1	0.7778	Value = 0.5556, ASE=0.1307
K=3	0.8333	Value = 0.6667, ASE=0.1211
K=5	0.8056	Value = 0.6111, ASE=0.1301
K=7	0.8333	Value = 0.6667, ASE=0.1211

Los modelos con $K = 3$ y $K = 7$ presentan los valores más elevadas en precisión, con 0.8333 y coeficiente de Kappa de Cohen 0.6667.

Naive Bayes

Un clasificador Naive Bayes es un clasificador probabilístico fundamentado en el teorema de Bayes y algunas hipótesis simplificadoras adicionales. Es a causa de estas simplificaciones, que se suelen resumir en la hipótesis de independencia entre las variables predictoras, que recibe el apelativo de naive, es decir, ingenuo.

(Wu et al. 2008)

El algoritmo Naive Bayes se basa en los principios fundamentales de la teoría de probabilidad condicional establecidos por Thomas Bayes en el siglo 18^{th}. Este teorema establece una relación entre eventos dependientes definida por la siguiente fórmula:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Entrenando el modelo

Creamos modelos de predicción con diferentes valores para Laplace.

```
#Creamos el modelo clasificador
nv0 <- naiveBayes(class~., data=train, laplace=0)
#Creamos predicciones a partir de dicho clasificador
nv0_p <- predict(nv0, test, level="class")
```

```
#Para Laplace=1
nv1 <- naiveBayes(class~., data=train, laplace=1)
nv1_p <- predict(nv0, test, level="class")
```

Predicción y evaluación

```
confusionMatrix(test_labels,nv0_p, positive="1")
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  3 15

               Accuracy : 0.8889
                 95% CI : (0.7394, 0.9689)
    No Information Rate : 0.5556
    P-Value [Acc > NIR] : 1.823e-05

                  Kappa : 0.7778

 Mcnemar's Test P-Value : 0.6171

            Sensitivity : 0.9375
            Specificity : 0.8500
```

```
         Pos Pred Value : 0.8333
         Neg Pred Value : 0.9444
             Prevalence : 0.4444
         Detection Rate : 0.4167
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.8938

       'Positive' Class : 1
```

```r
confusionMatrix(test_labels,nv1_p, positive="1")
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  3 15

               Accuracy : 0.8889
                 95% CI : (0.7394, 0.9689)
    No Information Rate : 0.5556
    P-Value [Acc > NIR] : 1.823e-05

                  Kappa : 0.7778

 Mcnemar's Test P-Value : 0.6171

            Sensitivity : 0.9375
            Specificity : 0.8500
         Pos Pred Value : 0.8333
         Neg Pred Value : 0.9444
             Prevalence : 0.4444
         Detection Rate : 0.4167
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.8938

       'Positive' Class : 1
```

Modelo	Precisión	Coeficiente Kappa de Cohen
Laplace = 0	0.8889	Value = 0.7778
Laplace = 1	0.8889	Value = 0.7778

Artificial Neural Network

El Machine Learning consiste en asignar entradas (por ejemplo, características de una célula), a objetivos (diagnóstico), lo cual se hace mediante la observación de muchos ejemplos de entradas y objetivos. Las redes neuronales artificiales realizan esta asignación de entrada-objetivo mediante una secuencia profunda de transformaciones de datos sencillas (capas), y estas transformaciones de datos se aprenden por la exposición a ejemplos.

La especificación de lo que hace una capa a los datos de entrada se almacena en los "pesos", que en esencia son un montón de números. En términos técnicos diríamos que la transformación implementada por la capa está parametrizada por sus pesos (los pesos también se denominan parámetros de una capa). En este contexto, "aprender" significa encontrar un conjunto de valores para los pesos de todas las capas de una red, de manera que la red asigne correctamente a nuevas entradas sus objetivos asociados.

Necesitamos evidentemente, un sistema que nos permita cuantificar cuanto nos estamos acercando a los valores deseados. Para controlar algo, primero tenemos que ser capaces de observarlo. Para controlar el resultado de una red neuronal, tenemos que ser capaces de cuantificar cuánto se aleja ese resultado de lo que esperábamos. Esa es la tarea de la **función de pérdida** de la red, también llamada función ojetivo. La función de pérdida coge las predicciones de la red y el objetivo verdadero y computa una puntuación de distancia.

El truco fundamental del deep learning es utilizar esta puntuación como señal de retroalimentación para ajustar un poco el valor de los pesos en una dirección que reducirá la puntuación de la pérdida. Este ajuste es tarea del optimizador que implementa lo que se denomina **algoritmo de retropropación**, el algoritmo central del deep learning. [cholletdeep]

Entrenando el modelo

Para las redes neuronales artificiales vamos a utilizar el paquete `caret`. Crearemos dos modelos con 4 y 5 nodos en la capa intermedia respectivamente, y 3 fold-Crossvalidation.

La función trainControl controla los matices computacionales de la función de entrenamiento. En este caso, el método "cv" indica que queremos llevar a cabo una validación cruzada, y a continuación especificamos el número de partes en las que queremos dividirlo, en este caso, 3.

```
# Escogemos crossvalidation y 3 plieuges
# Para no modificar nuestras variables especificamos classProbs=FALSE
ctrl <- trainControl(method= "cv", number = 3,
                     classProbs = FALSE, verboseIter = FALSE,
                     preProcOptions = list(thresh = 0.75, ICAcomp = 3, k = 5))


ann4 <- train(class~., data=train,
              method="nnet",
              preProcess = c('center', 'scale'),
              tuneGrid=expand.grid(size=c(4), decay=c(0.1)),
              trControl = ctrl)
```

```
# weights:  921
initial  value 47.157196
iter  10 value 10.928573
iter  20 value 3.762775
iter  30 value 3.293260
final  value 3.293008
converged
```

```
# weights:  921
initial  value 46.921915
iter  10 value 5.358833
iter  20 value 3.505639
iter  30 value 3.488851
final  value 3.488823
converged
# weights:  921
initial  value 52.332856
iter  10 value 8.917812
iter  20 value 3.580017
iter  30 value 3.525213
iter  40 value 3.524528
iter  50 value 3.524516
final  value 3.524515
converged
# weights:  921
initial  value 70.810343
iter  10 value 16.841335
iter  20 value 6.528610
iter  30 value 5.504767
iter  40 value 5.415949
final  value 5.415939
converged
```

```
ann5 <- train(class~., data=train,
              method="nnet",
              preProcess = c('center', 'scale'),
              tuneGrid=expand.grid(size=c(5), decay=c(0.1)), MaxNWts=2000,
              trControl = ctrl)
```

```
# weights:  1151
initial  value 51.434395
iter  10 value 13.446508
iter  20 value 3.273076
iter  30 value 3.057394
iter  40 value 3.048999
iter  50 value 3.048388
final  value 3.048375
converged
# weights:  1151
initial  value 55.648225
iter  10 value 4.630066
iter  20 value 3.059036
iter  30 value 3.055939
final  value 3.055921
converged
# weights:  1151
initial  value 51.211871
iter  10 value 6.154806
iter  20 value 3.085755
iter  30 value 3.050769
iter  40 value 3.050389
final  value 3.050387
```

```
converged
# weights:  1151
initial  value 66.966735
iter  10 value 5.736959
iter  20 value 3.586983
iter  30 value 3.564824
iter  40 value 3.564722
final  value 3.564721
converged
```

Predicción y evaluación

Podemos observar los valores asignados para nuestros valores de test.

```
plsClasses4 <- predict(ann4, newdata = test)
confusionMatrix(test_labels, plsClasses4)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 16  2
         1  1 17

               Accuracy : 0.9167
                 95% CI : (0.7753, 0.9825)
    No Information Rate : 0.5278
    P-Value [Acc > NIR] : 5.76e-07

                  Kappa : 0.8333

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9412
            Specificity : 0.8947
         Pos Pred Value : 0.8889
         Neg Pred Value : 0.9444
             Prevalence : 0.4722
         Detection Rate : 0.4444
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.9180

       'Positive' Class : 0
```

```
plsClasses5 <- predict(ann5, newdata = test)
confusionMatrix(test_labels, plsClasses5)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
```

```
        0 17  1
        1  1 17

               Accuracy : 0.9444
                 95% CI : (0.8134, 0.9932)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 9.706e-09

                  Kappa : 0.8889

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9444
            Specificity : 0.9444
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.9444
             Prevalence : 0.5000
         Detection Rate : 0.4722
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.9444

       'Positive' Class : 0
```

Modelo	Precisión	Coeficiente Kappa de Cohen
Nodos = 4	0.9167	Value = 0.8333
Nodos = 5	0.9444	Value = 0.8889

Support Vector Machine

Las máquinas de vector soporte (SVM) son un conjunto de algoritmos de aprendizaje supervisado desarrollados por Vladimir Vapnik y su equipo en los laboratorios AT&T. Las SVM utilizan hiperplanos para separar las observaciones en grupos de individuos semejentes (que comparten características). Cuando estas observaciones se pueden separar perfectamente mediante una línea recta se les llama linealmente separables. En ocasiones esta condición no se da, pero SVM puede operar igualmente bien en condiciones en las que no hay linealidad.

En la inmensa mayoría de las ocasiones se pueden trazar multitud de hiperplanos separando las dos clases de objetos. El criterio para seleccionar el mejor hiperplano será aquel que maximice la separación entre los grupos, también llamada Hiperplano de Margen Máximo (MMH).

Entrenando el modelo

Para la clasificación de los promotores mediante SVM utilizaremos el paquete `caret`, con dos métodos: el método de función lineal `svmLinear` y el método de función radial `svmRadialSigma`, para hiperplanos no lineales.

```
svm_lineal <- train(class~., data=train,
                    method="svmLinear",
                    preProcess = c('center', 'scale'),
                    trControl = ctrl)

svm_rbf <- train(class~., data=train,
                    method="svmRadialSigma",
                    preProcess = c('center', 'scale'),
                    trControl = ctrl)
```

Predicción y evaluación

Obtenemos las predicciones para los datos de test.

```
plsClassesLin <- predict(svm_lineal, newdata = test)
confusionMatrix(test_labels, plsClassesLin)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 15  3
         1  1 17

               Accuracy : 0.8889
                 95% CI : (0.7394, 0.9689)
    No Information Rate : 0.5556
    P-Value [Acc > NIR] : 1.823e-05

                  Kappa : 0.7778

 Mcnemar's Test P-Value : 0.6171
```

```
            Sensitivity : 0.9375
            Specificity : 0.8500
         Pos Pred Value : 0.8333
         Neg Pred Value : 0.9444
             Prevalence : 0.4444
         Detection Rate : 0.4167
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.8938

       'Positive' Class : 0
```

```r
plsClassesRbf <- predict(svm_rbf, newdata = test)
confusionMatrix(test_labels, plsClassesRbf)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  1 17

               Accuracy : 0.9444
                 95% CI : (0.8134, 0.9932)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 9.706e-09

                  Kappa : 0.8889

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.9444
            Specificity : 0.9444
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.9444
             Prevalence : 0.5000
         Detection Rate : 0.4722
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.9444

       'Positive' Class : 0
```

Modelo	Precisión	Coeficiente Kappa de Cohen
Lineal	0.8889	Value = 0.7778
RBF	0.9444	Value = 0.8889

Arbol de decisión

El árbol de decision es un algoritmo de aprendizaje supervisado que se puede utilizar tanto para problemas de clasificación como para problemas de regresión. Es un clasificador estructurado en árbol, donde los nodos internos representan las características de un conjunto de datos, las ramas representan las reglas de decisión y cada nodo hoja representa el resultado.

Entrenando el modelo

Para los modelos obtenidos a partir del algoritmo arbol de decisión utilizaremos `caret` y 3-fold cross validation.

```
#Emplearemos la lista generada anteriormente en ctrl para especificar el tipo de remuestreo
# Especificamos el paquete rpart como arbol de decisión sin boosting
dt <- train(class~., data=train,
            method="rpart",
            trControl =ctrl)

# El paquete gbm proporciona un arbol de decisión con boosting
dtb <- train(class~., data=train,
            method="gbm",
            trControl =ctrl,
            verbose = FALSE)
```

Predicción y evaluación

```
dtClasses <- predict(dt, newdata = test)
confusionMatrix(test_labels, dtClasses)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 16  2
         1 10  8

               Accuracy : 0.6667
                 95% CI : (0.4903, 0.8144)
    No Information Rate : 0.7222
    P-Value [Acc > NIR] : 0.82500

                  Kappa : 0.3333

 Mcnemar's Test P-Value : 0.04331

            Sensitivity : 0.6154
            Specificity : 0.8000
         Pos Pred Value : 0.8889
         Neg Pred Value : 0.4444
             Prevalence : 0.7222
```

```
         Detection Rate : 0.4444
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.7077

       'Positive' Class : 0
```

```r
dcClassesB <- predict(dtb, newdata = test)
confusionMatrix(test_labels, dcClassesB)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  3 15

               Accuracy : 0.8889
                 95% CI : (0.7394, 0.9689)
    No Information Rate : 0.5556
    P-Value [Acc > NIR] : 1.823e-05

                  Kappa : 0.7778

 Mcnemar's Test P-Value : 0.6171

            Sensitivity : 0.8500
            Specificity : 0.9375
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.8333
             Prevalence : 0.5556
         Detection Rate : 0.4722
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.8938

       'Positive' Class : 0
```

Modelo	Precisión	Coeficiente Kappa de Cohen
No Boosting	0.75	Value = 0.5
Boosting	0.9167	Value = 0.8333

Random Forest

El random forest, como su nombre indica, consta de una gran cantidad de árboles de decisión individuales que operan como un conjunto. Cada árbol individual en el bosque aleatorio proporciona una predicción de clase y la clase que ha sido predicha un mayor número de veces se convierte en la predicción de nuestro modelo.

Entrenando el modelo

Entrenamos el modelo para 50 y para 100 árboles de decisión

```
rf50 <- randomForest(class~., data=train, importance = TRUE, ntree=50)

rf100 <- randomForest(class~., data=train, importance = TRUE, ntree=100)
```

Predicción y evaluación

```
predTrain50 <- predict(rf50, test, type = "class")
confusionMatrix(test_labels, predTrain50)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  2 16

               Accuracy : 0.9167
                 95% CI : (0.7753, 0.9825)
    No Information Rate : 0.5278
    P-Value [Acc > NIR] : 5.76e-07

                  Kappa : 0.8333

 Mcnemar's Test P-Value : 1

            Sensitivity : 0.8947
            Specificity : 0.9412
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.8889
             Prevalence : 0.5278
         Detection Rate : 0.4722
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.9180

       'Positive' Class : 0
```

```r
predTrain100 <- predict(rf100, test, type = "class")
confusionMatrix(test_labels,predTrain100)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 17  1
         1  3 15

               Accuracy : 0.8889
                 95% CI : (0.7394, 0.9689)
    No Information Rate : 0.5556
    P-Value [Acc > NIR] : 1.823e-05

                  Kappa : 0.7778

 Mcnemar's Test P-Value : 0.6171

            Sensitivity : 0.8500
            Specificity : 0.9375
         Pos Pred Value : 0.9444
         Neg Pred Value : 0.8333
             Prevalence : 0.5556
         Detection Rate : 0.4722
   Detection Prevalence : 0.5000
      Balanced Accuracy : 0.8938

       'Positive' Class : 0
```

Modelo	Precisión	Coeficiente Kappa de Cohen
n = 50	0.9167	Value = 0.8333
n = 100	0.9444	Value = 0.8889

Conclusión. Elección del modelo

En la elección del modelo tendremos en cuenta los estimadores de calidad, los recursos que implica la implementación de estos modelos sobre muestras mayores, la fiabilidad que tienen los algoritmos sobre el tipo de dato con el que estamos trabajando (para lo cual será necesario consultar la bibliografía), y por último, las posibles modificaciones previas al modelo que se hayan realizado sobre los datos.

Atendiendo únicamente a la precisión, destacan tres modelos que coinciden con un valor de 94.44% de precisión y 88.89% de coeficiente Kappa de Cohen. Estos son la red neuronal artificial de cinco nodos, el algoritmo SVM con función radial, y el random forest de 100 árboles. Sería interesante generar modelos más potentes con la red neuronal y el random forest, dada la mejora sustancial que se produce al aumentar a cinco nodos y cien árboles respectivamente.

En lo que respecta a los datos, los tres algoritmos emplearon la codificación one-hot y la misma divisón en train y test. La única diferencia radica en el paquete empleado: randomForest para randomForest, y caret para redes neuronales y SVM. El paquete caret permite un control computacional del algoritmo que nos ha permitido obtener las predicciones mediante cross-validation, lo que supone una mejora teórica y práctica sobre su calidad y capacidad predictiva. Sería interesante replicar el análisis con condiciones idénticas para los tres algoritmos.

Dado que la capacidad de predicción es semejante, podemos reducir la elección del modelo a las características que nos interesen. Por un lado, tanto redes neuronales como SVM utilizan modelos black box, generados por el propio algoritmo; sin embargo, de acuerdo con (Lantz 2019) los modelos generados por random forest también son difíciles de interpretar. Tanto SVM como random forest presentan una ventaja importante con respecto a las redes neuronales, y es su ligereza y portabilidad, por lo que, la gran capacidad de adaptación de las redes neuronales quizá deje de ser un factor decisivo en nuestra elección. SVM y random forest son ambos bastante robustos, y se ven poco afectados por observaciones ruidosas.

Una vez alcanzados este punto, considero que tanto SVM como random forest son una decisión acertada para la predicción de secuencias promotoras. Sin embargo, es necesario hacer una puntualización, y es que la teoría y la experiencia nos dicen que los modelos cross-validated son más confiables, por lo que nuevamente, sería interesante replicar el modelo random forest con cross-validation. Con nuestro conocimiento actual, y dado que los resultados son idénticos, es conveniente escoger el modelo menos arriesgado, SVM con función radial en este caso.

Bibliografía

Harrag, Fouzi, and Selmene Gueliani. 2020. "Event Extraction Based on Deep Learning in Food Hazard Arabic Texts." *arXiv Preprint arXiv:2008.05014*.

Harris, David, and Sarah Harris. 2010. *Digital Design and Computer Architecture*. Morgan Kaufmann.

Kramer, Oliver. 2013. *Dimensionality Reduction with Unsupervised Nearest Neighbors*. Springer.

Lantz, Brett. 2019. *Machine Learning with R: Expert Techniques for Predictive Modeling*. Packt Publishing Ltd.

Mucherino, Antonio, Petraq Papajorgji, and Panos M Pardalos. 2009. *Data Mining in Agriculture*. Vol. 34. Springer Science & Business Media.

Sharan, Roded, Shaul Karni, and Yifat Felder. 2007. "Analysis of Biological Networks: Transcriptional Networks–Promoter Sequence Analysis." *Tel Aviv University*, 1–5.

Wu, Xindong, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, et al. 2008. "Top 10 Algorithms in Data Mining." *Knowledge and Information Systems* 14 (1): 1–37.