PEC1: Predicción de los splice junctions

Virginio Cepas

noviembre 04, 2021

Contents

1 Algoritmo k-NN

El algoritmo k-NN (Nearest Neighbours algorithm) se basa en la clasificación de un individuo tomando como referencia a sus **k** vecino más cercanos. En un primera fase, se entrena al algoritmo con un conjunto de datos de entrenamiento ya clasificados previamente. A continuación, se utiliza un set de datos no clasificados para obtener una clasificacion según la similitud que tienen estos datos con sus vecinos. Es decir, que se les asigna la clase mayoritaria de los **k** vecinos más próximos. En este algoritmo la elección de la **k** es importante ya que puede variar en el resultado dado que la distancia entre el dato a clasificar y sus **k** vecinos juegan un papel crucial en su clasificacion.

La siguiente tabla muestra las fortalezas y debilidades del algoritmo **k-NN**

Fortalezas	Debilidades
· Simple y efectivo	· No produce un modelo, limitando la capacidad de entender cómo están relacionadas las características con la clase
· No hace suposiciones subyacentes sobre la distribución de los datos	· Requiere la selección de una k apropiada
· Fase de entrenamiento rápida	· Fase de clasificación lenta
	· Las características nominales y datos perdidos requieren de un procesamiento adicional

2 Funcion para implementar la codificación “One-hot”

```
# Creamos una función donde se aplicará a cada elemento de la lista (lapply) y
# substituiremos con gsub las letras del DNA por su codificacion en one-hot
onehot_function<-function(x){
  x <- lapply(x,as.character)
  x <- gsub("A", "10000000", x)
  x <- gsub("G", "01000000", x)
  x <- gsub("T", "00100000", x)
  x <- gsub("C", "00010000", x)
  x <- gsub("D", "00001000", x)
  x <- gsub("N", "00000100", x)
  x <- gsub("S", "00000010", x)
  x <- gsub("R", "00000001", x)
  dna <- lapply(strsplit(x, ""), as.numeric)# separamos los caracteres con ""
  # se pasa la lista a vectores para formar una matriz que se rellena de fila en fila
  dna <- matrix(unlist(dna), nrow = filas , byrow = TRUE)
  return(dna)
}
```

Para mostrar la función, se utiliza la secuencia del enunciado ya que la podemos utilizar como referencia para comprobar que ha funcionado correctamente.

```
ejemplo <- as.data.frame("CCAGCTGCATCACAGGAGGCCAGCGAGCAGGTCTGTTCCAAGGGCCTTCGAGCCAGTCTG")
filas <- nrow(ejemplo) # Indicamos el parametro fila que se utiliza en la funcion.
onehot_function(ejemplo)[1:100]
```

```
##   [1] 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
##  [38] 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0
##  [75] 1 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1
```

Se observa que coinciden las primeras 100 entradas con las del enunciado.

3 Desarrollar un script en R que implemente un clasificador k-NN

3.1 Leer los datos del fichero splice.txt y hacer una breve descripción de ellos

```
splice <- read.table("splice.txt", header = FALSE, sep = ",")
names(splice) = c("class", "seq_name", "secuencia")
# Quitamos los espacios en blanco en el df
splice <- splice %>%
  mutate(across(where(is.character), str_trim))
str(splice)
```

```
## 'data.frame':    3190 obs. of  3 variables:
##  $ class    : chr  "EI" "EI" "EI" "EI" ...
##  $ seq_name : chr  "ATRINS-DONOR-521" "ATRINS-DONOR-905" "BABAPOE-DONOR-30" "BABAPOE-DONOR-867" ...
##  $ secuencia: chr  "CCAGCTGCATCACAGGAGGCCAGCGAGCAGGTCTGTTCCAAGGGCCTTCGAGCCAGTCTG" "AGACCCGCCGGGAGGCG
```

```
table(splice[,1])
```

```
##
##   EI   IE    N
##  767  768 1655
```

La tabala contiene 3 columnas con las clases, el nombre y la secuencia. También se muestra el total de clases dentro de la tabla.

3.2 Transformación de las secuencias de nucleótidos en vectores numéricos

Para realizar esta transformación, utilizaremos la funcion creada anteriormente

```
filas <- nrow(splice)
onehot_dna <- onehot_function(splice[, 3])
#Se crea la tabla que contiene la nueva columna
splice_oh <- splice %>%
  select (-c("secuencia"))
splice_oh <- cbind(splice_oh,onehot_dna)
head(splice_oh)[1:20]
```

```
##   class           seq_name 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
## 1    EI   ATRINS-DONOR-521 0 0 0 1 0 0 0 0 0  0  0  1  0  0  0  0  1  0
## 2    EI   ATRINS-DONOR-905 1 0 0 0 0 0 0 0 0  1  0  0  0  0  0  0  1  0
## 3    EI   BABAPOE-DONOR-30 0 1 0 0 0 0 0 0 1  0  0  0  0  0  0  0  0  1
## 4    EI  BABAPOE-DONOR-867 0 1 0 0 0 0 0 0 0  1  0  0  0  0  0  0  0  1
## 5    EI BABAPOE-DONOR-2817 0 1 0 0 0 0 0 0 0  0  0  1  0  0  0  0  0  0
## 6    EI CHPIGECA-DONOR-378 0 0 0 1 0 0 0 0 1  0  0  0  0  0  0  0  0  1
```

Se observa que la función se ha aplicado correctamente y ha transformado la secuencia en el formato "One-hot".

3.3 Implementación del algoritmo k-NN

3.3.1 Preparación de los datos

```
#Se divide el df segun si son acceptors o donors junto con las secuencias sin splicing.
acceptors_n_df <- filter(splice_oh, class == "IE"  | class == "N")
donors_n_df <- filter(splice_oh, class == "EI"  | class == "N")

#Se crean los dos grupos al azar sin reemplazo
set.seed(123) #fijar la semilla para el generador pseudoaleatorio
acceptors_n <- sample(1:nrow(acceptors_n_df),round(0.67 * nrow(acceptors_n_df))
                      , replace = F)
set.seed(123)
donors_n <- sample(1:nrow(donors_n_df),round(0.67 * nrow(donors_n_df))
                   , replace = F)

#Creación de los datos de entrenamiento y test
```

```
training_acceptors_n <-acceptors_n_df[3:480][acceptors_n,]
test_acceptor_n <-acceptors_n_df[3:480][-acceptors_n,]
training_donors_n <-donors_n_df[3:480][donors_n,]
test_donors_n <-donors_n_df[3:480][-donors_n,]

#Creación de las etiquetas para cada grupo y transformación a factor para poder
#utilizarlos en confusionMatrix()
training_label_acceptors_n <- as.factor(acceptors_n_df[acceptors_n,1])
test_label_acceptors_n <- as.factor(acceptors_n_df[-acceptors_n,1])
training_label_donors_n <- as.factor(donors_n_df[donors_n,1])
test_label_donors_n <- as.factor(donors_n_df[-donors_n,1])

head(training_acceptors_n)[1:28] # Representación de la muestra
```

```
##      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
## 2227 0 1 0 0 0 0 0 0 0  0  0  1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  1
## 526  0 0 0 1 0 0 0 0 1  0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  0  1  0
## 195  0 0 0 1 0 0 0 0 0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0  1
## 1842 0 0 0 1 0 0 0 0 1  0  0  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0  0
## 1142 0 0 0 1 0 0 0 0 0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  0  1  0  0
## 1253 0 0 1 0 0 0 0 0 0  0  0  1  0  0  0  0  0  0  1  0  0  0  0  0  1  0  0  0
```

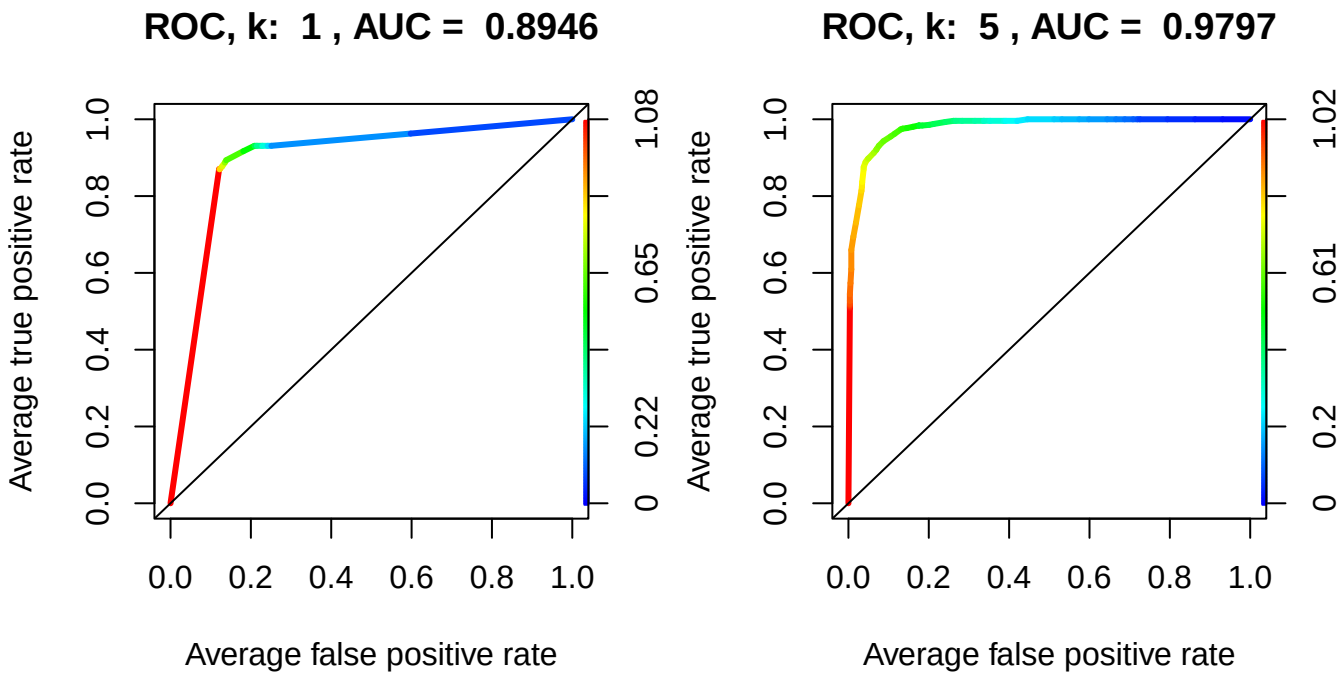
3.3.2 Aplicar el k-NN, realizar curva ROC para cada K y mostrar el valor AUC

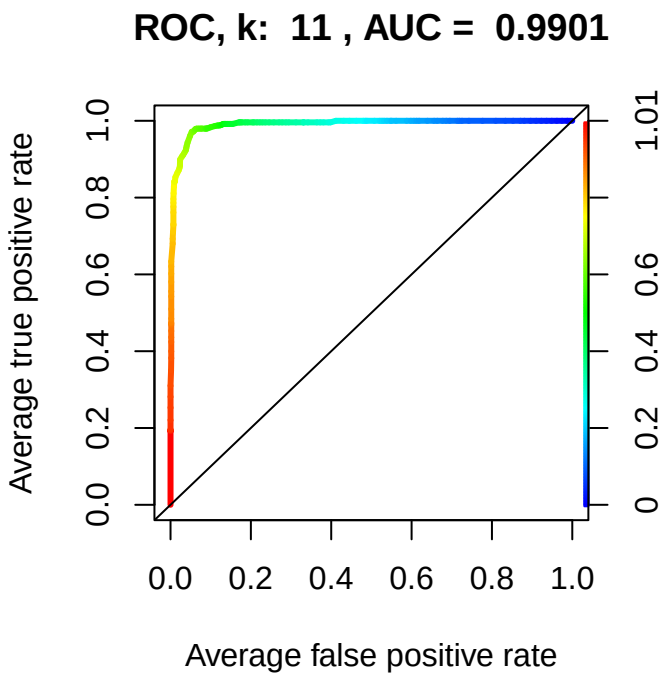
Acceptors

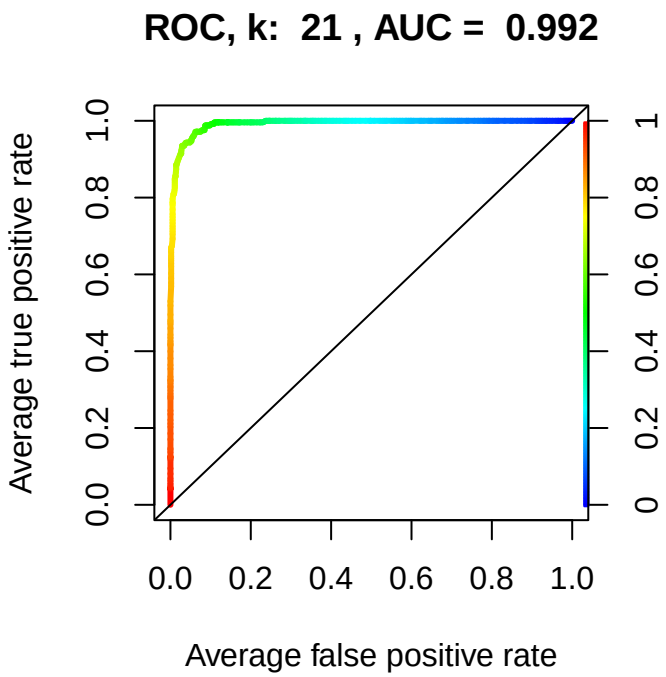
Para hacer el informe más dinamico, creamos un bucle donde se calculará el algoritmo **k-NN** para cada una de las **k**'s.
Para calcular la probabilidad de la clase positiva, las que representan secuencias con puntos de *splicing* (IE,EI), se extrae de la funcion *knn()* asignando a la probabilidad el valor TRUE para posteriormente restarle 1 y obtenir así la probabilidad de ser clase ganadora. Una vez calculada la probabilidad de ser clase ganadora se representa la curva ROC con su valor AUC.

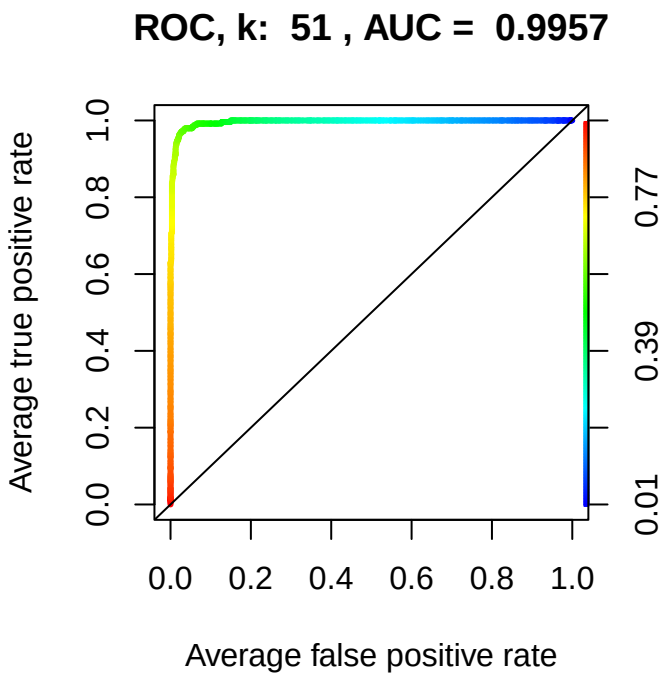
```
k <- params$k_value # Indicamos los valores de k
res_auc_acc <- c()# Creamos este vector para utilizarlo más tarde en una tabla resumen
for (i in k){
  test_pred_acc <- knn(train = training_acceptors_n, test = test_acceptor_n
                       , cl = training_label_acceptors_n, k = i, prob = TRUE)
  # Obtenemos la probabilidad ganadora utilizando el comando attr para poder
  # extraerlo del objeto
  prob <- attr(test_pred_acc, "prob")
  # Calculamos la posibilidad de ser clase ganadora para aquellos que sean N
  prob_all <- ifelse(test_pred_acc == "IE", prob, 1-prob)
  pred_knn <- prediction(predictions = prob_all, labels = test_label_acceptors_n
              # Ordenamos de menor a mayor para obtener el gráfico de lo contrario,
              # el eje de las x se invierte
                         ,label.ordering = c("N","IE"))
  # Cálculo true y false prediction rates para la curva ROC
  perf_tf <- performance(pred_knn, "tpr", "fpr")
  perf_auc <- performance(pred_knn, "auc")# Cálculo AUC
  # Extracción de AUC utilizando el @ ya que se trata de un objeto S4
  perf_auc <- unlist(perf_auc@y.values)
  res_auc_acc <- c(res_auc_acc,perf_auc)# Guardamos los AUC
```

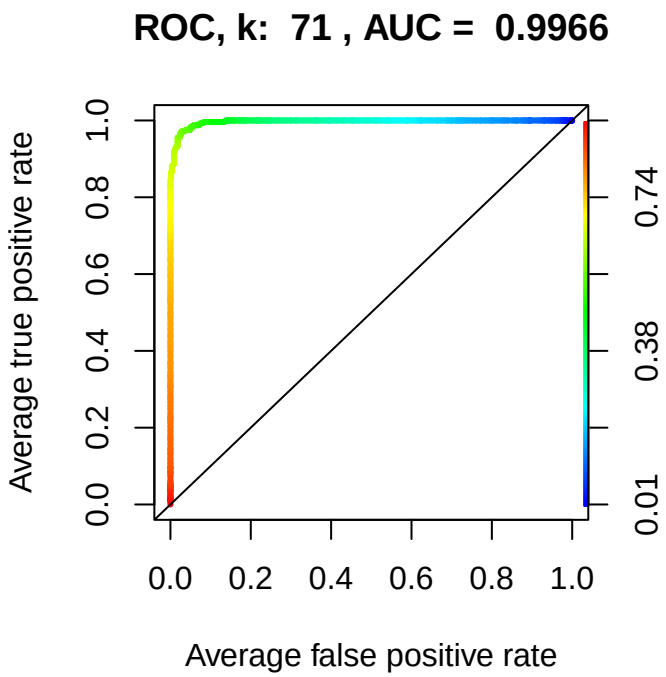
```
  res_auc_acc <- round(res_auc_acc,4)
  # El formato sale de https://ipa-tys.github.io/ROCR/articles/ROCR.html
  plot(perf_tf, avg = "threshold", colorize = T, lwd = 3
       , main = paste("ROC, k: ", i, ", AUC = ", round(perf_auc,4)))
  abline(a = 0, b = 1, lwd = 1, lty = 1)
}
```







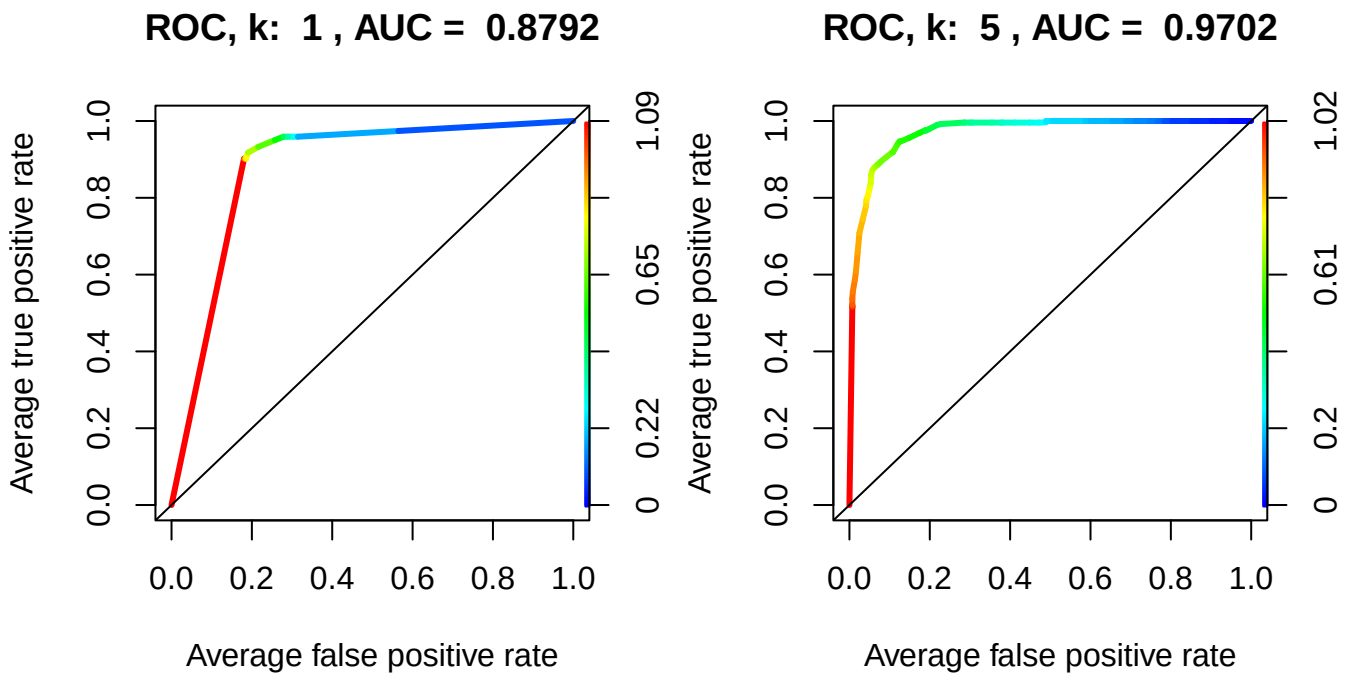


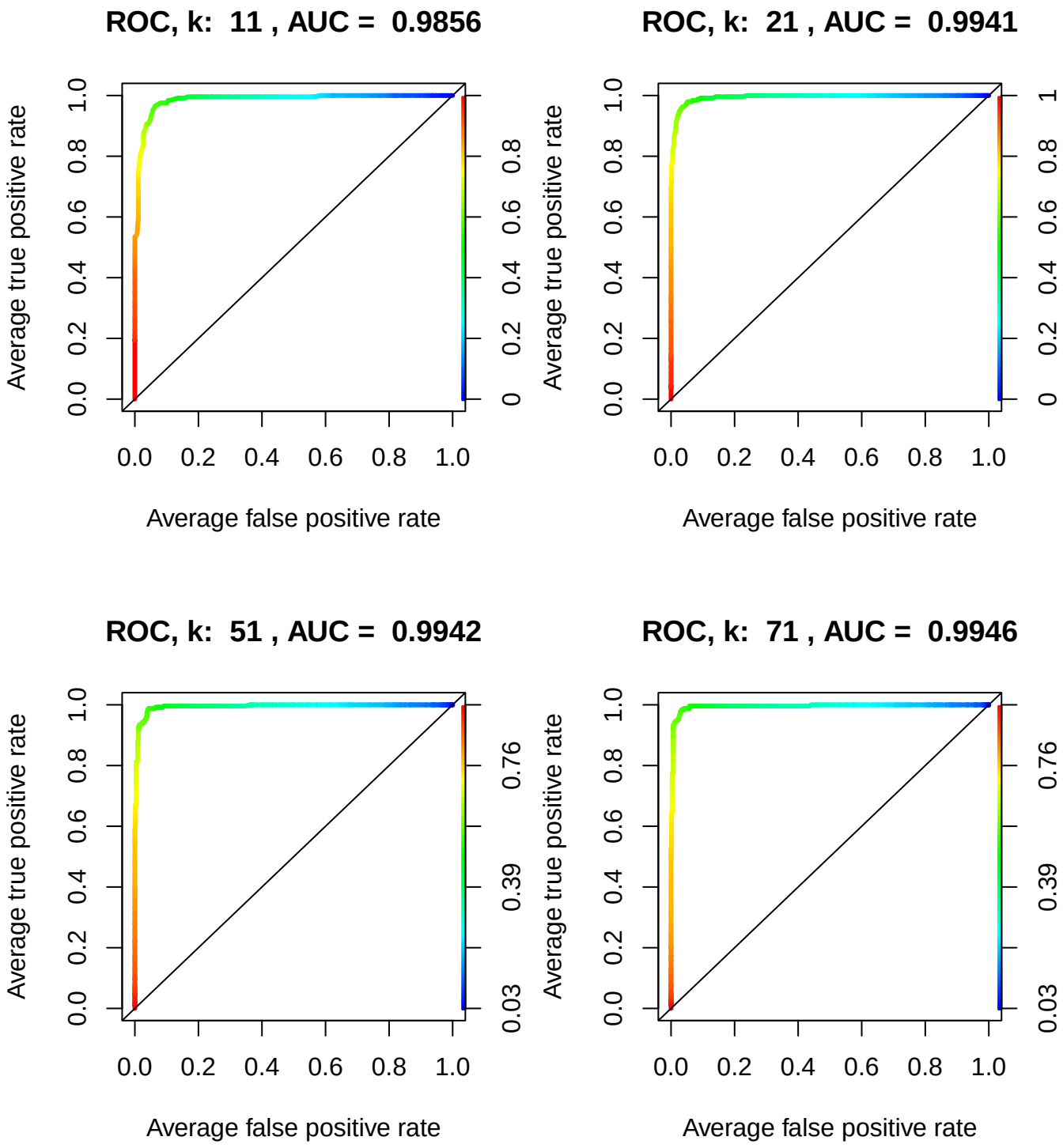


Donors

Para los donors, aplicamos el mismo código que el apartado anterior.

```r
res_auc_don <- c()# Creamos este vector para utilizarlo más tarde en una tabla resumen
for (i in k){
  test_pred_don <- knn(train = training_donors_n, test = test_donors_n
                       , cl = training_label_donors_n, k=i, prob= TRUE)
  # Obtenemos la probabilidad ganadora utilizando el comando attr para poder
  # extraerlo del objeto
  prob <- attr(test_pred_don, "prob")
  # Calculamos la posibilidad de ser clase ganadora para aquellos que sean N
  prob_all <- ifelse(test_pred_don == "EI", prob, 1-prob)
  pred_knn <- prediction(predictions = prob_all, labels = test_label_donors_n
            # Ordenamos de menos a mayor para obtener el gráfico de lo contrario,
            # el eje de las x se invierte
                         ,label.ordering = c("N","EI"))
  # Cálculo true y false prediction rates para la curva ROC
  perf_tf <- performance(pred_knn, "tpr", "fpr")
  perf_auc <- performance(pred_knn, "auc")# Cálculo AUC
  # Extracción de AUC utilizando el @ ya que se trata de un objeto S4
  perf_auc <- unlist(perf_auc@y.values)
  res_auc_don <- c(res_auc_don,perf_auc)# Guardamos los AUC
  res_auc_don <- round(res_auc_don,4)
  #El formato sale de https://ipa-tys.github.io/ROCR/articles/ROCR.html
  plot(perf_tf, avg = "threshold", colorize = T, lwd = 3
       , main = paste("ROC, k: ", i, ", AUC = ", round(perf_auc,4)))
  abline(a = 0, b = 1, lwd = 1, lty = 1)
}
```





3.3.3 Comentar los resultados de la clasificación

Acceptors

Para obtener los falsos positivos, falsos negativos y error de clasificación obtenidos para los diferentes valores

de **k** se pueden hacer de diferentes formas. Uno de ellos sería hacer una *CrossTable* y mediante la formula (TP + TN) / (TP + TN + FP + FN) obtener la *accuracy* y restarle a 1 este valor para obtener el error de clasificación. La otra alternativa a realizar los cálculos de forma individual es crear una *ConfussionMatrix* y extraer los da tos de ahí.

```
resum_acc <- data.frame(k, FP=NA, FN=NA, mal_clas=NA)
j <- 0
set.seed(123)
for (i in k) {
  j <- j +1
  test_pred_acc <- knn(train =training_acceptors_n, test = test_acceptor_n
                       , cl = training_label_acceptors_n, k=i, prob= T)
  cfM <-confusionMatrix(test_pred_acc,test_label_acceptors_n, positive = "IE")
  err_rate_acc <- (1-cfM$overall["Accuracy"])*100#calculo del error
  # Se extrae de la matriz de confusion los FP y los FN. Debemos tener en cuenta que
  # en esta tabla los valores se transponen
  resum_acc[j,2:4] <- c(cfM$table[2,1], cfM$table[1,2], (err_rate_acc))
}
tabla_acc <- cbind(resum_acc, res_auc_acc)
```

La siguiente tabla muestra los datos que pide el enunciado para las secuencias acceptors.

Table 2: Acceptors

K	Falso positivo	Falso negativo	Error de clasificación (%)	AUC
1	20	99	14.875	0.8946
5	5	90	11.875	0.9797
11	3	69	9.000	0.9901
21	1	63	8.000	0.9920
51	2	52	6.750	0.9957
71	1	43	5.500	0.9966

En este modelo, a medida que aumenta la **k**, tanto los falsos positivos y falsos negativos disminuyen. No obstante, se observa que en la **k** = 21 la precisión del modelo es mayor ya que contiene unicamente 1 falso negativo en su predicción. En cuanto al % de mal clasificados se observa que al aumentar la **k** el número de falsos negativos decrece y por lo tanto es de esperar que el % en error de clasificación también se vea reducido (hasta casi un 6%). El modelo mejora substancialmente en cuanto al error de clasificación si comparamos la **k** = 1 y la **k** = 11. En cuanto al valor de AUC se observa que con la **k** = 11 hasta **k** = 71 la curva ROC tiene una pendiente más elevada y su AUC bajo la curva ROC es del 99% demostrando su alto poder de predicción con una precisión elevada.

Finalmente, el modelo para la detección de regiones de *splicing*, en el limite de introns-exones o *acceptors*, funciona mejor a medida que aumenta la **k**. No obstante hay que evitar modelos con *overfiting* seleccionando **k**'s muy elevadas. En este caso en concreto, quizás la **k**=21 sería una buena opción para implementar.

Donors

Para los *donors* implementamos el mismo código que en el apartado anterior

```
resum_don <- data.frame(k, FP=NA, FN=NA, mal_clas=NA)
j <- 0
set.seed(123)
for (i in k) {
  j <- j +1
```

```
  test_pred_don <- knn(train = training_donors_n, test = test_donors_n
                       , cl = training_label_donors_n, k=i, prob= TRUE)
  cfM <-confusionMatrix(test_pred_don,test_label_donors_n, positive = "EI")
  err_rate_don <- (1-cfM$overall["Accuracy"])*100#calculo del error
  # Se extrae de la matriz de confusion los FP y los FN. Debemos tener en cuenta que
  # en esta tabla los valores se transponen
  resum_don[j,2:4] <- c(cfM$table[2,1], cfM$table[1,2], (round(err_rate_don,3)))
}
tabla_don <- cbind(resum_don, res_auc_don)
```

La siguiente tabla muestra los datos que pide el enunciado para las secuencias donors.

Table 3: Donors

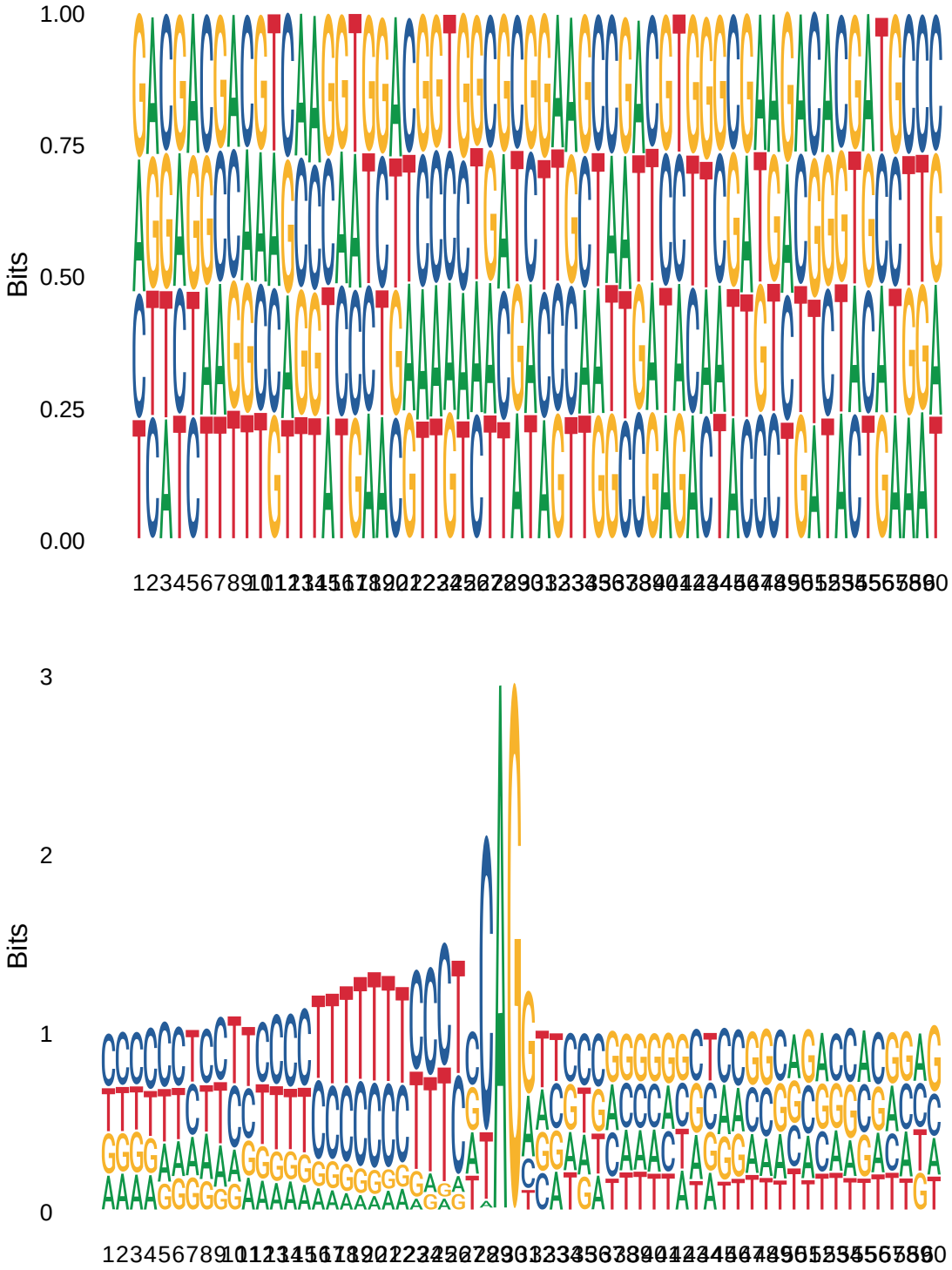
K	Falso positivo	Falso negativo	Error de clasificación (%)	AUC
1	14	134	18.523	0.8792
5	9	90	12.390	0.9702
11	3	69	9.011	0.9856
21	2	54	7.009	0.9941
51	2	39	5.131	0.9942
71	1	37	4.756	0.9946

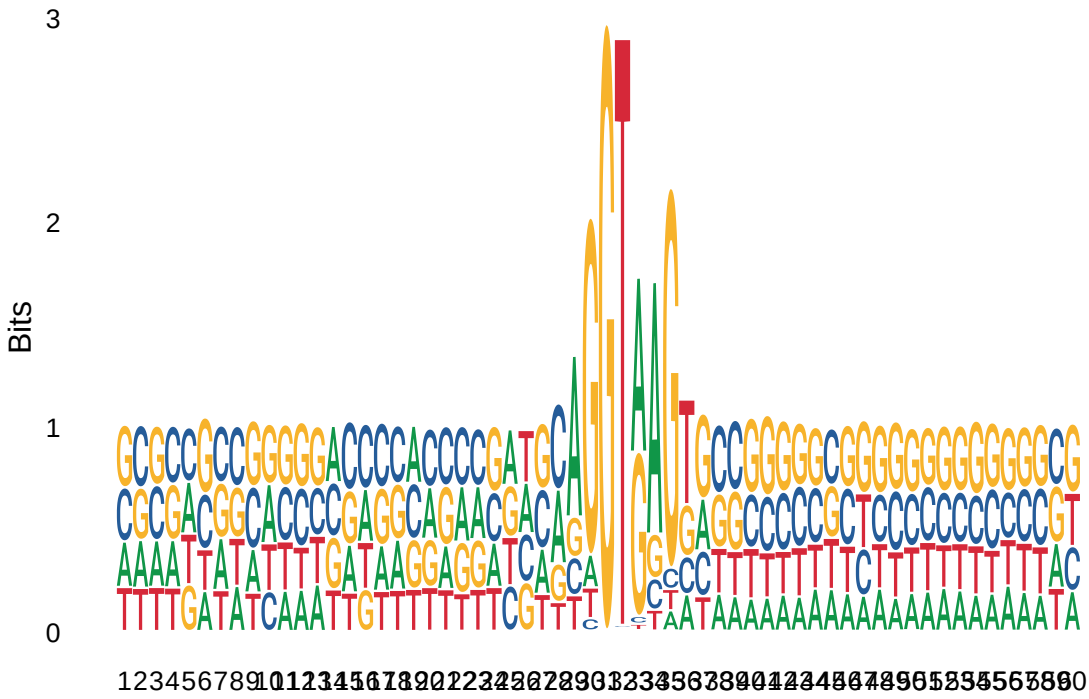
Igual que pasaba anteriormente, a medida que aumenta la **k** tanto los falsos positivos como los falsos negativos disminuyen. La sensibilidad de este modelo varia del 99,4 para **k**=11 a 99,7 para **k**=21, la diferencia es pequeña. También se observa que hay un cambio substantial en el error de clasificación si comparamos **k**=1 y **k**= 21 donde disminuye hasta un 11,5%. En cuanto al AUC observamos que la diferencia de **k**=21 hasta **k** = 71 es de 0.0005. En este caso para seleccionar una **k** adecuada para el modelo y predecir los lugares de *splicing* para los *donors*, elegiria **k**=21 ya que la diferencia en la sensibilidad y AUC con **k**'s mayores es pequeña y así evitariamos problemas de *overfiting*.

4 Secuencias logo

Para poder presentar la secuencia logo se crea previamente una variable con las letras que queremos que contenga. Debemos de crearla porque el que viene por defecto (DNA) no incluye los carácteres D, N, S y R. Si no los incluimos, las secuencias logo no se generan correctamente.

```
# Creación del abecedario
alphabet <- c("A", "G", "T", "C", "D", "N", "S", "R")
tipos <- c("N","IE", "EI")
for (j in tipos) {
  plot(ggseqlogo(splice[splice$class==j,3], seq_type='other', namespace = alphabet))
}
```


Bits
1.00
0.75
0.50
0.25
0.00
Bits
3
2
1
0



La primera secuencia logo corresponde a las secuencias que no incluyen un sitio de *splicing*. Se observa que no predomina ningún nucleotido por encima de otro ya que se pueden encontrar de forma homogenea los 4 nucleótidos en toda la secuencia.

Para el segundo gráfico, las IE o límites intron/exon, en las posiciones 29-30 se observa claramente como dominan los nucleótidos A y G. Seguramente este patrón de nucleótidos sea una señal celular por donde realizar el *splicing* cuando se sintetizan las proteinas.

El último gráfico, EI o límites exón/intrón, en las posiciones 31-32 predominan los nucleótidos G y T. Igual que en el caso anterior, seguramente se trate de un patrón genético que la máquinaria celular reconozca como un lugar de *splicing*.

Finalmente se puede concluir que los carácteres D, N, S y R son minoritarios en la secuencia ya que no hay representación en ninguna de las secuencias logo.