Predicción 'in-silico' de sitios de escisión reconocidos por la proteasa del HIV-1

Amelia Martínez Sequera

6/11/2020

Índice de contenidos:

1. Algoritmo k-NN.

El algoritmo de clasificación k-NN se basa en calcular la distancia desde un punto

$$x_0$$

que queremos clasificar a los *k training points* (puntos de entrenamiento) más cercanos. Una vez tengamos identificados los k puntos más cercanos o voters del conjunto de entrenamiento, clasificaremos nuestro punto

$$x_0$$

a partir de la clasificación mayoritaria de los voters.

Tabla de fortaleza y debilidades

2. Codificación ortogonal (*orthogonal encoding*) de los octameros.

El primer paso es importar los archivos, y convertir la segunda columna en factor (inicialmente es numérica):

```
setwd("~/UOC/ML/PEC1")
library(readr)
impens <- read.csv("impensData.txt", stringsAsFactors = FALSE, header = FALSE)
schilling<- read.csv("schillingData.txt", stringsAsFactors = FALSE, header = FALSE)
ort <- rbind(impens, schilling)
colnames(ort) <- c("octamer", "cleavage")
ort$cleavage <- factor(ort$cleavage, levels = c(-1, 1),
                           labels = c("Negative", "Positive"))
```

Para crear la función encodeAAStringChain que codifique los octámeros a partir del formato de los ficheros, se define primero una función encodeAA que codifica el aminoácido a partir de un alfabeto arbitrario. La posición del AA en el alfabeto será la que indique qué bit va a 1. La función encodeAACharChain es una función recursiva que codifica el vector de caracteres generado en encodeAAStringChain encadenando la codificación del primer AA con el resto de la cadena:

```
# Devuelve el vector que codifica el aminoácido a partir del vector aminoacids
encodeAA <- function(aminoacid) {

  retVal <- rep(0,20) # 20 ceros
  aminoacids <- c('G', 'A', 'L', 'M', 'F', 'W', 'K', 'Q', 'E', 'S', 'P', 'V',
                  'I', 'C', 'Y', 'H', 'R', 'N', 'D', 'T')
  # cambio posición para el i-ésimo aminoácido
  retVal[which(aminoacids == aminoacid)] = 1
  return(retVal)

}

# Función recursiva a partir del vector de caracteres de una cadena (octámero)
encodeAACharChain <- function(characterChain) {

  if (length(characterChain)>1) {
    # codificación ortogonal del primer AA + codificacion ortogonal
    # del resto de la cadena
    retVal <- c(encodeAA(characterChain[1]),
                encodeAACharChain(characterChain[2:length(characterChain)]))
  }
  else {
    ## último AA
    retVal <- encodeAA(characterChain[1])
  }

}

# Devuelve la codificación ortogonal a partir de un octámero (la cadena de AA
# puede tener una longitud arbitraria)
encodeAAStringChain <- function(octamero) {

  # convierte a vector de caracteres y devuelve la codificación
  encodeAACharChain(strsplit(octamero,"")[[1]])

}
```

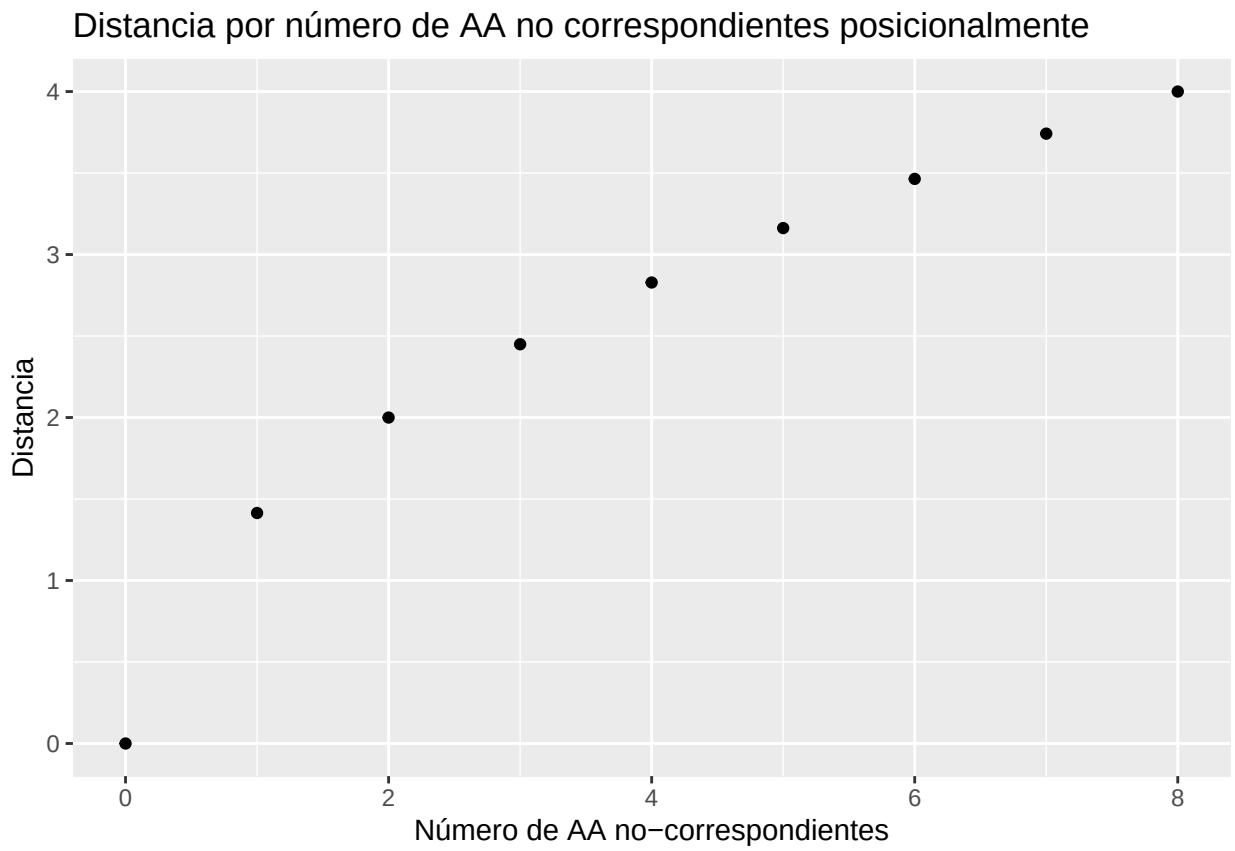
La función encodeAAStringChain codificará ortogonalmente un número arbitrario de AA (no se limita a octámeros).

Conceptualmente la distancia entre octámeros será menor cuantos más AA comunes tengan en las mismas posiciones. La distancia entre octámeros se puede calcular en función del número de AA comunes en las mismas posiciones teniendo en cuenta que la distancia al cuadrado entre dos octámeros diferentes será 2 por cada octámero diferente en la posición i

```
# número de AA diferentes para las posicione del octámero
distancia <- function(n) {
  distancia <- sqrt(2*n)
}


distancias <- data.frame(0:8,distancia(0:8))
colnames(distancias) <- c("difAA","dist")

p <- ggplot(distancias, aes(difAA, dist))+geom_point()+
ggtitle("Distancia por número de AA no correspondientes posicionalmente")+
  xlab("Número de AA no-correspondientes")+ylab("Distancia")
p
```



A partir de la definición de distancia y de la codificación ortogonal se observa que: - El cálculo de la distancia no depende del alfabeto: no se obtendrán diferentes resultados al cambiar la codificación. - La distancia entre octámeros no crece linealmente. - Cualquier posición i contribuye de la misma manera al cálculo de la distancia: el significado biológico es clasificar la adherencia (cleavage) de los octámeros entre la cuarta y la quinta posición, pero la el cálculo de la distancia no tiene en cuenta la posición de los AA más allá de si son diferentes o no: no se privilegian posiciones. - No requiere normalización: todas las dimensiones del vector ortogonal son iguales. No hay cambios de magnitud.

Tratamiento de los datos

Se renombran los 160 componentes del vector ortogonal como C.XXX. Se crea un dataframe df a partir de la codificación y se van añadiendo los vectores a medida que codificamos los octámeros:

```
# nombre de los componentes (columnas en el dataframe)
compVals <- paste0("C.", 1:160)

# creación del dataframe a partir de la codificación de los octámeros
df <- data.frame(matrix(ncol = 160, nrow = 0))
for (w in ort$octamer) {
  v <- encodeAAStringChain(w)
  df <- rbind(df, v)
}
# nombre de columnas C.XXX (C-componente; XXX-[1:160])
colnames(df) <- compVals
```

Se unifica el resultado en el dataframe *ort* y se elimina la cadena de octámeros:

```
# se añade el dataframe al antiguo para la invocación al clasificador
# y se elimina la columna del octamero
ort <- cbind(ort, df)
ort$octamer <- NULL
```

3. Desarrollo de un script en R que implemente un clasificador knn.

Separación de datos en training (67%) y test (33%)

```
set.seed(123)
# buscamos el 67% de los índices (rows)
v <- c(1:nrow(ort))
percentage <- 67
numMuestras <- round(nrow(ort) * percentage / 100)
# indices de las muestras
index_train <- sample(v, replace = FALSE, size = numMuestras)
```

Definimos train y test y creamos un vector de etiquetas con la clasificación real para cada uno de los conjuntos. Eliminamos las etiquetas de los conjuntos de training y test:

```
library(class)

train_set <- ort[index_train,]
test_set <- ort[-index_train,]

# renumeramos los row de los nuevos dataframes
rownames(train_set) <- 1:nrow(train_set)
rownames(test_set) <- 1:nrow(test_set)
```

```
# etiquetas
test_set_labels <- test_set[1]
train_set_labels <- train_set[1]

# borramos las etiquetas de los conjuntos de test
train_set[,1] <- NULL
test_set[,1] <- NULL
```

Entrenamiento del modelo

La documentación de knn {class} indica que el resultado de generar un modelo a partir del algoritmo knn es la clasificación del juego de test. Utilizando las etiquetas de clasificación "real" obtendremos las métricas de evaluación del modelo.

Implementamos primero una función evaluateModel que genera las métricas de evaluación a partir de los datos de la matriz de confusión para dos clases:

```
evaluateModel <- function (model, test_set_labels) {

  TP <- 0
  FP <- 0
  TN <- 0
  FN <- 0
  TOT <- 0

  # ----------------------------------------
  # NUMEROS DEL DATASET
  # ----------------------------------------

  # Calculamos el número total de casos, los falsos positivos,
  # falsos negativos, verdaderos positivos y verdaderos negativos
  for (i in 1:length(model)) {
    TOT <- TOT + 1

    # positivo
    if (model[i] == TRUE) {
      # true positive
      if (test_set_labels[i,1] == TRUE) {
        TP <- TP + 1
      }
      # false positive
      else {
        # valor real falso, clasificado como positivo
        FP <- FP + 1
      }
    }
    # negativo
    else {
      # true negative
      if (test_set_labels[i,1] == FALSE) {
        TN <- TN + 1
      }
```

```
    # false negative
    else {
      FN <- FN + 1
    }
  }
}

# ----------------------------------------
# CALCULOS
# ----------------------------------------

# accuracy y tasa de error
accuracy <- (TP + TN) / TOT
errorRate <- 1 - accuracy

# matriz de confusion: eje x el modelo, eje y los datos reales
m <- matrix(nrow = 2, ncol = 2, data = c(TP, FP, FN, TN))
f <- data.frame(m)
row.names(f) <- c("TRUE", "FALSE")
colnames(f) <- c("TRUE", "FALSE")

# kappa
# the hypothetical probability of chance agreement,
# using the observed data to calculate the probabilities
# of each observer randomly seeing each category
pe <- 0
for (i in row.names(f)) {
  pe <- pe + ((sum(f[i,])/TOT) * (sum(f[,i])/TOT))
}
kappa <- 1- ((1-accuracy) / (1-pe))

# sensibilidad, especificidad, precisión, recall y
# f-Score
sensitivity <- TP / (TP + FN)
especificity <- TN / (TN + FP)
precission <- TP / (TP + FP)
recall <- TP / (TP + FN)
fScore <-  2 * (precission * recall / (precission + recall))

# valores de salida
text <- c("total", "accuracy", "error rate", "kappa",
          "sensitivity", "specificity", "precission",
          "F-score", "TP", "FP", "TN", "FN")
nums <- c(TOT, accuracy, errorRate, kappa,
          sensitivity, especificity, precission,
          fScore, TP, FP, TN, FN)

# summary
summary <- data.frame(text, nums)

# devolvemos una lista con el summary y la matriz de
# confusión
return(list(summary, f))
```

```
}
```

La función doClassification servirá para devolver en una sola respuesta los modelos para las k propuestas. Invocará a la función evaluateModel para devolver la evaluación del modelo:

```
doClassification <- function (train, test,
  train_labels, test_labels, ki) {

    # entrenamos
    trained_model <- knn(train = train, test = test,
                cl = train_labels$cleavage, k = ki)

    # matriz de confusión
    evaluation <- evaluateModel(trained_model, test_labels)

    # compactamos y devolvemos
    newList <- list("k" = ki, "model" = trained_model,
                    "evaluation" = evaluation)
}
```

la función doNClassifications devuelve directamente los datos de clasificación a partir del vector de k de manera compacta:

```
doNClassifications <- function(train_set, train_set_labels,
                               test_set, test_set_labels) {

    # vector de Ks
  c <- c(3,5,7,11)
  result <- list(list())

  # indice de lista
  j <- 1
  for (ki in c) {
    classification <- doClassification(train = train_set,
                                       train_labels = train_set_labels,
                                       test = test_set,
                                       test_labels = test_set_labels, k = ki)
    result[[j]] <- classification
    j <- j + 1
  }
  return (result)
}

result <- doNClassifications(train_set, train_set_labels,
                             test_set, test_set_labels)
```

En función de el valor del seed tendremos unas matrices de confusión diferentes. La documentación de knn condiciona la clasificación al azar. Es decir: para ciertos puntos podemos tener a la misma distancia euclídea puntos clasificados de manera diferente. El algoritmo elige aleatoriamente los k voters cuando a la misma distancia tenemos n puntos con k<n.

Evaluación del modelo

Evaluamos el modelo para k=3 a partir de result utilizando confusionMatrix de caret:

```
library(caret)
```

```
## Loading required package: lattice
```

```
cmat1 <- confusionMatrix(result[[1]]$model, as.factor(test_set_labels$cleavage),
                positive = "Positive")

cmat1
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Negative Positive
##   Negative     1180      139
##   Positive       19       54
##
##                Accuracy : 0.8865
##                  95% CI : (0.8686, 0.9027)
##     No Information Rate : 0.8614
##     P-Value [Acc > NIR] : 0.003083
##
##                   Kappa : 0.3571
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.27979
##             Specificity : 0.98415
##          Pos Pred Value : 0.73973
##          Neg Pred Value : 0.89462
##              Prevalence : 0.13865
##          Detection Rate : 0.03879
##    Detection Prevalence : 0.05244
##       Balanced Accuracy : 0.63197
##
##        'Positive' Class : Positive
##
```

```
# Para k= 5
cmat5 <- confusionMatrix(result[[2]]$model, as.factor(test_set_labels$cleavage),
                positive = "Positive")

cmat5
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Negative Positive
##   Negative     1189      156
```

```
##   Positive       10       37
##
##                Accuracy : 0.8807
##                  95% CI : (0.8626, 0.8973)
##     No Information Rate : 0.8614
##     P-Value [Acc > NIR] : 0.01841
##
##                   Kappa : 0.2686
##
##  Mcnemar's Test P-Value : < 2e-16
##
##             Sensitivity : 0.19171
##             Specificity : 0.99166
##          Pos Pred Value : 0.78723
##          Neg Pred Value : 0.88401
##              Prevalence : 0.13865
##          Detection Rate : 0.02658
##    Detection Prevalence : 0.03376
##       Balanced Accuracy : 0.59168
##
##        'Positive' Class : Positive
##
```

```
# Para k= 7
cmat7 <- confusionMatrix(result[[3]]$model, as.factor(test_set_labels$cleavage),
                positive = "Positive")

cmat7
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Negative Positive
##   Negative     1193      167
##   Positive        6       26
##
##                Accuracy : 0.8757
##                  95% CI : (0.8572, 0.8926)
##     No Information Rate : 0.8614
##     P-Value [Acc > NIR] : 0.0636
##
##                   Kappa : 0.1995
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.13472
##             Specificity : 0.99500
##          Pos Pred Value : 0.81250
##          Neg Pred Value : 0.87721
##              Prevalence : 0.13865
##          Detection Rate : 0.01868
##    Detection Prevalence : 0.02299
##       Balanced Accuracy : 0.56486
##
```

```
##        'Positive' Class : Positive
##
```

```
# Para k= 11
cmat11 <- confusionMatrix(result[[4]]$model, as.factor(test_set_labels$cleavage),
               positive = "Positive")

cmat11
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Negative Positive
##   Negative     1197      178
##   Positive        2       15
##
##                Accuracy : 0.8707
##                  95% CI : (0.8519, 0.8879)
##     No Information Rate : 0.8614
##     P-Value [Acc > NIR] : 0.1663
##
##                   Kappa : 0.1232
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.07772
##             Specificity : 0.99833
##          Pos Pred Value : 0.88235
##          Neg Pred Value : 0.87055
##              Prevalence : 0.13865
##          Detection Rate : 0.01078
##    Detection Prevalence : 0.01221
##       Balanced Accuracy : 0.53803
##
##        'Positive' Class : Positive
##
```

```
# Proporción de clasificaciones correctas:

100 * sum(test.sites_labels == knn.3)/100  # Para knn = 3
```

```
## [1] 1342
```

```
100 * sum(test.sites_labels == knn.5)/100  # Para knn = 5
```

```
## [1] 1341
```

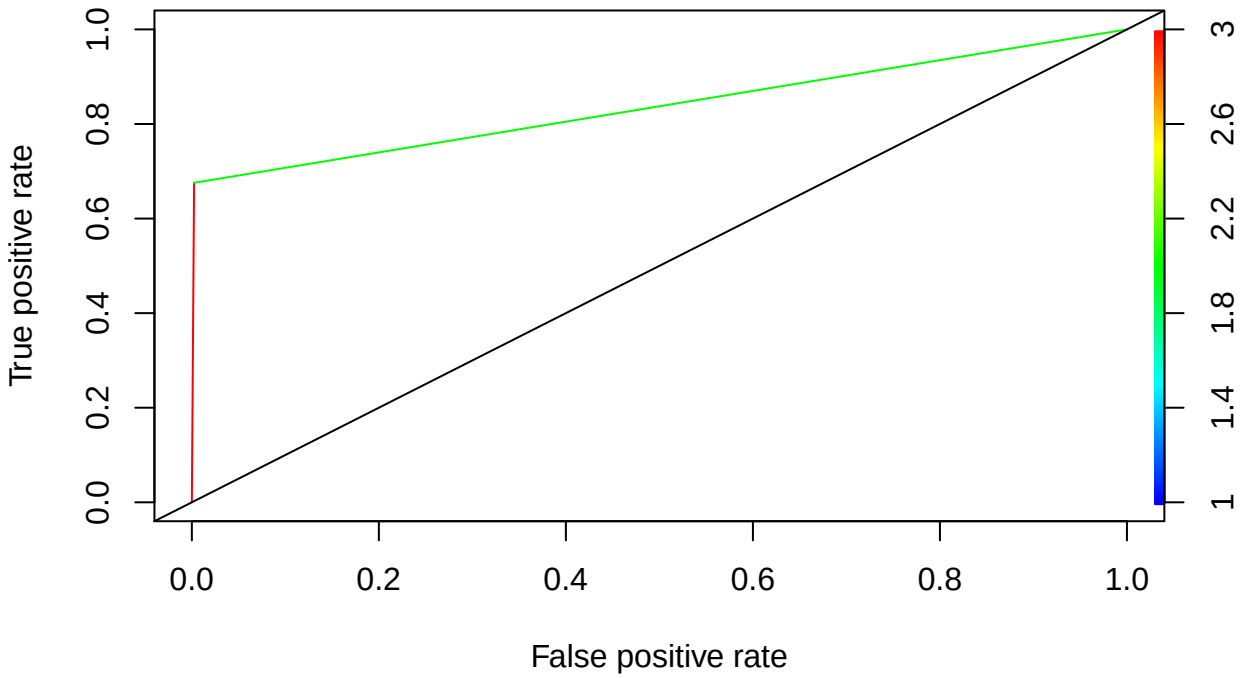
```
100 * sum(test.sites_labels == knn.7)/100  # Para knn = 7
```

```
## [1] 1325
```

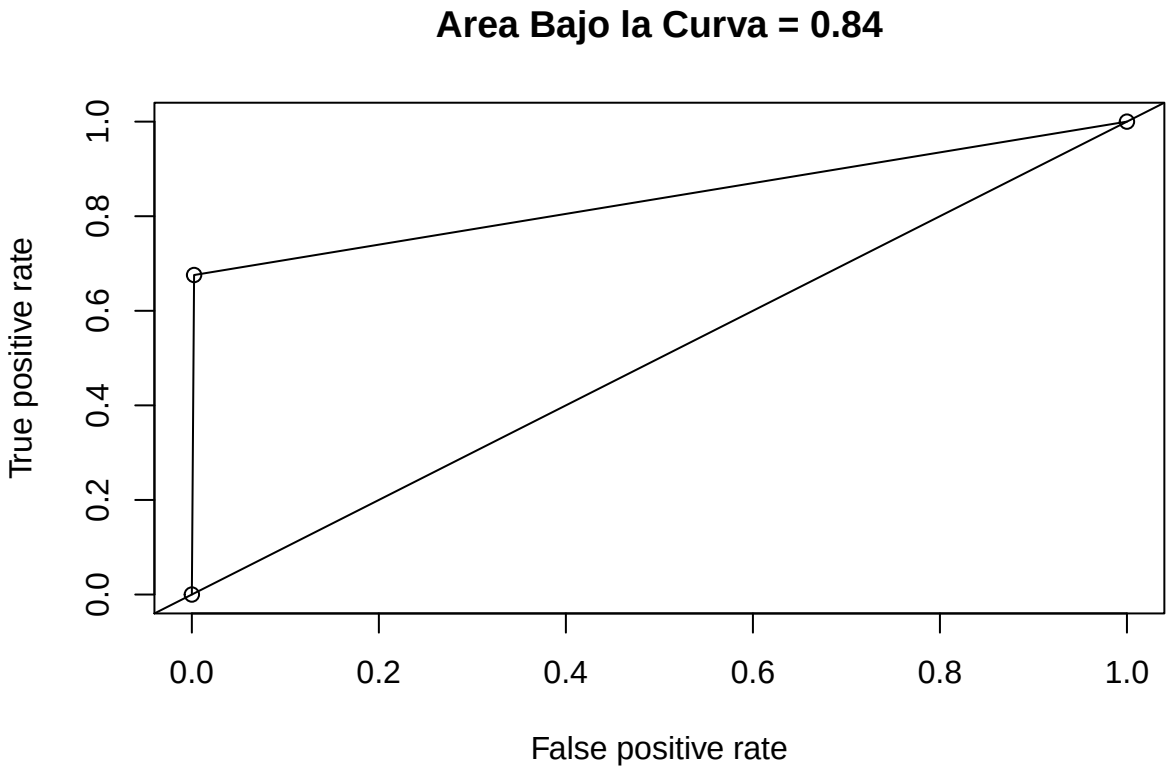
```r
100 * sum(test.sites_labels == knn.11)/100  # Para knn = 11
```

```
## [1] 1310
```

Curva ROC



AUC



Resultados

El modelo tiene una precisión (accuracy) para k=3 de 0.8865, lo que nos indica que está clasificando correctamente el 88.65 % del juego de test. Aunque a priori pueda parecer un un buen dato, si observamos el valor de sensitivity (proporción de cleavage identificado) tenemos un valor muy bajo, de 0.27979, lo que nos indica que solamente un 27.979 % de los cleavage sites se identifican correctamente. Aún así, la sensibilidad del test resulta todavía menor para los otros valores de k.

El valor de specificity es muy alto, de 0.98415, lo cual no debe sorprender con el valor de accuracy que tenemos, ya que un octámero tenga un cleavage site es una condición "rara" y el modelo clasifica (correctamente) la mayoría de negativos. Finalmente comentar que el valor de kappa (0.3571) es bajo, pero tenemos que tener en cuenta la poca prevalencia (Positives) en los juegos de datos.

Se observa que el número de falsos positivos disminuye a la vez que aumenta k, disminuyen los errores tipo I. En cambio, el número de falsos negativos aumenta aumentando los errores de tipo II.

Por otro lado, si tenemos en cuenta los objetivos del modelo, que no es otro que encontrar cleavage sites en los octámeros, nos interesaría maximizar el valor de sensitivity (encontrar todos los cleavage sites posibles), por lo que buscaremos maximizar TPR = TP/P = TP/(TP+FN)

Con esta premisa, modificar las k no mejora el rendimiento (el valor de sensitivity disminuye para k>3).

Respecto a la curva ROC, cualitativamente podemos ver que esta parece ocupar el espacio en la esquina superior izquierda del diagrama, lo que sugiere que está más cerca de un clasificador perfecto que la diagonal que representa un clasificador inútil. Cuantitativamente, el AUC (área bajo la curva) es 0.84, que se considera

excelente/bueno. Pero para saber si el modelo funcionaría bien con otros datos, antes deberíamos analizar si podemos extrapolar las predicciones más allá de los datos de prueba.