PEC1: Clasificación de familias de genes a partir de secuencias de ADN

Demetrio Muñoz Alvarez

2023-11-06

Contents

Algoritmo k-NN

La bioinformática, un campo multidisciplinario, se centra en el análisis y la interpretación de datos biológicos mediante el uso de técnicas computacionales y estadísticas. Uno de los algoritmos fundamentales en este dominio es el algoritmo ***k-Nearest Neighbors (kNN)***

El algoritmo kNN es una técnica de aprendizaje automático supervisado utilizada para clasificar e inferir datos basados en la similitud con observaciones previamente categorizadas. Esta similitud se calcula generalmente utilizando la distancia euclidiana, que mide la distancia lineal entre dos puntos en un espacio

multidimensional, permitiendo determinar la semejanza entre ellos. La fórmula de la ***distancia euclidiana*** se expresa como:

$$dist(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + ... + (p_n - q_n)^2}$$

En el ámbito de la bioinformática, el algoritmo kNN se aplica en diversas áreas, como la clasificación de secuencias genómicas, la identificación de proteínas, el análisis de datos de expresión génica y la predicción de propiedades biológicas, entre otras aplicaciones.

A pesar de su simplicidad, el algoritmo kNN es ampliamente utilizado en el aprendizaje automático. A continuación, se presentará una tabla que resumirá las ***fortalezas*** y ***debilidades*** del algoritmo (Lantz 2019):

Fortalezas	Debilidades
Es simple y efectivo	No genera un modelo, lo que limita la capacidad de comprender cómo las características están relacionadas con la clase
No hace suposiciones sobre la distribución subyacente de los datos	Requiere la selección de un valor apropiado para 'k'
La fase de entrenamiento es rápida	La fase de clasificación puede ser lenta
	Las características nominales y los datos faltantes requieren procesamiento adicional

Introducción

En esta primera práctica, vamos a analizar el conjunto de datos **"human_data.txt"**, el cual contiene 2000 secuencias de genes junto con sus respectivas familias, representadas por números. Hay siete familias diferentes: 0 para receptores acoplados a proteínas G, 1 para quinasas de tirosina, 2 para fosfatasas de tirosina, 3 para sintetasas, 4 para sintasas, 5 para canales de iones y 6 para factores de transcripción.

Estas secuencias se dividirán en hexámeros de longitud $\mathbf{K = 6}$. Dado que el alfabeto biológico consta de 4 letras (A, C, G, T), existen un total de 4^6 = 4096 hexámeros distintos que pueden aparecer al extraer los hexámeros de un gen.

El objetivo final de la práctica es evaluar la capacidad del **algoritmo kNN** para predecir la familia a la que pertenece un gen a partir de su secuencia. Para lograrlo, se creará una función capaz de dividir las secuencias en hexámeros de longitud $\mathbf{K = 6}$ y se construirá una matriz que contará los hexámeros en cada secuencia de genes. Este paso se implementará en un script, y con los resultados obtenidos, se aplicarán los algoritmos **kNN** de esta actividad.

Desarrollar una función propia en R (o python) que implemente el contaje de los hexámeros de un secuencia dada.

La función ***kmers_count_function*** toma una lista de secuencias de nucleótidos y un valor ***K*** como argumentos. Luego, genera una matriz que representa la frecuencia de ocurrencia de todos los hexámeros en las secuencias proporcionadas, excluyendo aquellas que contienen la letra **'N'** en su nomenclatura para simmplificar el análisis en posteriores apartados.

La matriz resultado tiene filas que representan las secuencias de entrada y columnas que representan los hexámeros de las secuencias. Cada celda de la matriz almacena la ocurrencia de un hexámero en una secuencia en particular.

```r
# Definimos la función "kmers_count_function" que toma dos argumentos:
  # Primer argumento "sequences": vector de secuencias de ADN o ARN.
  # Segundo argumento "K": longitud de corte de hexámeros.
# La función consta de dos partes:
  # Primera parte: Genera una lista de hexámeros de las secuencias dadas.
  # Segunda parte: Genera una matriz de ocurrencia de los hexámeros a partir de las secuencias.
kmers_count_function <- function(sequences, K) {
# Creamos una lista para almacenar los hexámeros de las secuencias.
  kmers_list <- list()
  # Iteramos a través de cada secuencia del argumento "sequences".
  for (seq in sequences) {
    # Creamos vector de caracteres vacío para los hexámeros.
    kmers <- character(0)
    # Iteramos a través de la secuencia, cortando hexámeros de longitud "K"
    for (position in 1:(nchar(seq) - K + 1)) {
      hexamero <- substr(seq, position, position + K - 1)  # Extraemos el hexámero.
      kmers <- c(kmers, hexamero)  # Agregamos el hexámero al vector.
    }
    # Almacenamos la lista hexámeros de la secuencia actual en la lista "kmers_list".
    kmers_list <- c(kmers_list, list(kmers))
  }
  # Creamos un vector con todos los hexámeros únicos y sin "N".
  unique_hexamers <- unique(unlist(kmers_list))
  kmers_filter <- unique_hexamers[!grepl("N", unique_hexamers)]
  # Número de secuencias en "kmers_list". Número de filas.
  num_sequences <- length(kmers_list)
  # Número de hexámeros únicos filtrados. Número de columnas.
  # Este valor debería ser igual a 4^K.
  num_kmers_filter <- length(kmers_filter)
  # Creamos la matriz de ceros, con filas igual al número de secuencias y columnas
  # igual al número de hexámeros únicos.
  hexamer_matrix <- matrix(0, nrow = num_sequences, ncol = num_kmers_filter,
                           dimnames = list(NULL, kmers_filter))
  # Creamos un diccionario para mapear los hexámeros.
  hexamer_dict <- setNames(1:num_kmers_filter, kmers_filter)
  # Completamos la matriz de ocurrencia:
  for (i in 1:num_sequences) {
    for (hexamero in kmers_list[[i]]) {
      if (hexamero %in% kmers_filter) {
        hexamer_index <- hexamer_dict[hexamero]
        hexamer_matrix[i, hexamer_index] <- hexamer_matrix[i, hexamer_index] + 1
      }
    }
  }
  return(hexamer_matrix)
}
```

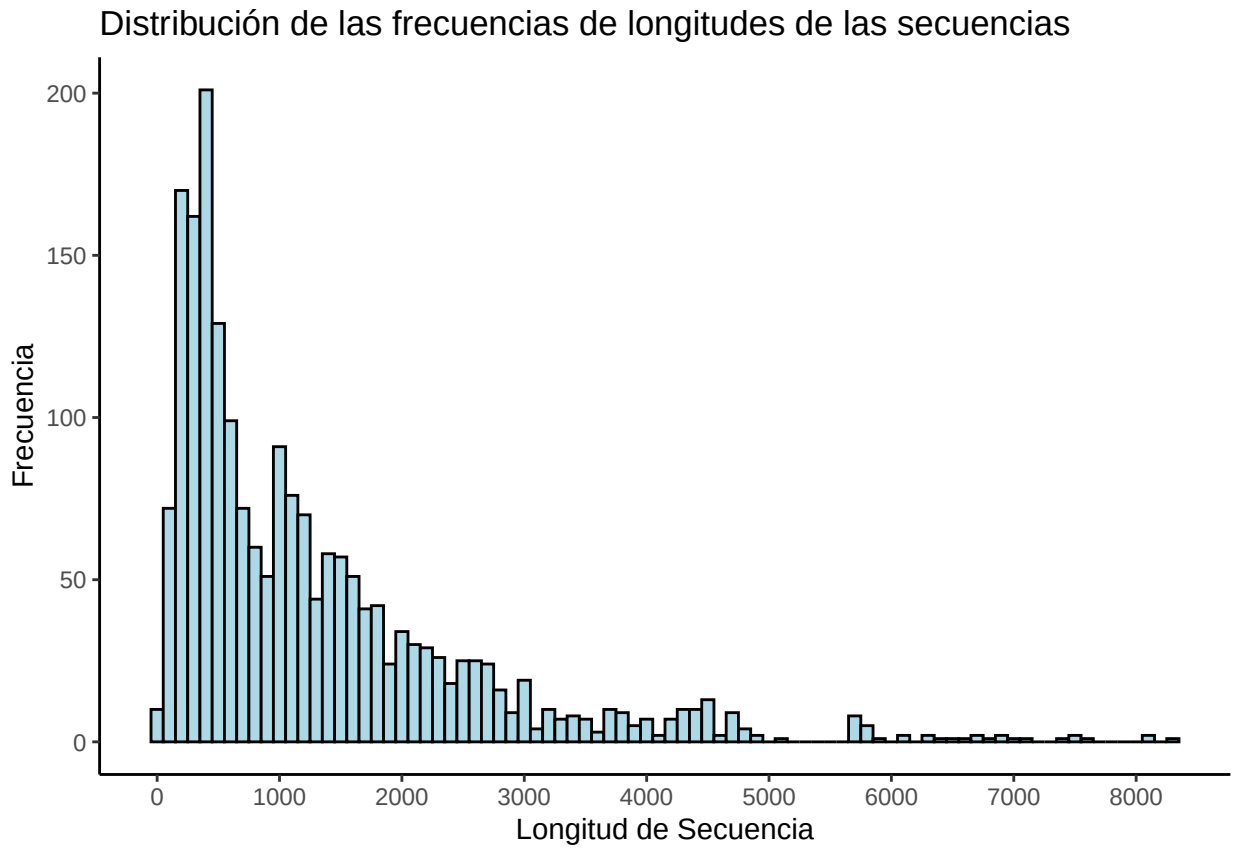
Desarrollar un script en R que implemente un clasificador k-nn. El script realiza los siguientes apartados

El script ***clasificador_knn*** realiza las siguientes acciones:

1. Carga un conjunto de datos especificado en la variable human_data.txt.
2. Genera un histograma de las secuencias del conjunto de datos human_data.txt que muestra la distribución de las frecuencias de longitudes de las secuencias.
3. Implementa la función previamente desarrollada para contar los hexámeros en las secuencias, almacenando los resultados en un objeto llamado **kmers_count**. Se establece $\mathbf{K = 6}$ como indica la actividad.

En resumen, el script carga datos, crea un histograma de las longitudes de las secuencias y calcula la matriz de ocurrencia de hexámeros en las secuencias dadas.

```
source("C:/Users/Deme/Desktop/Master/Machine Learning/pec1/clasificador_knn.R")
```



Antes de continuar, comprobamos si la matriz generada con nuestra función es igual a la proporcionada en la actividad, comparando la columna del hexámero **'AAAAAA'** en ambas matrices con algunas líneas de código:

```
# Cargamos la matriz de conteo del fichero hexameros.RData
load("C:/Users/Deme/Desktop/Master/Machine Learning/pec1/hexameros.RData")
# Comparamos las columnas "AAAAAA" de ambas matrices:
col1 <- count[, "AAAAAA"] # Extraemos la columna "AAAAAA" de la matriz.
col2 <- kmers_count[, "AAAAAA"] # Extraemos la columna "AAAAAA" de la matriz.
comparacion <- identical(col1, col2) # Comparamos los datos.
if (comparacion) {
  print("Las columnas de ambas matrices son iguales para el hexámero 'AAAAAA'")
} else {
```

```
  print("Las columnas de ambas matrices iguales para el hexámero 'AAAAAA'")
}
```

```
## [1] "Las columnas de ambas matrices son iguales para el hexámero 'AAAAAA'"
```

Realizar la implementación del algoritmo knn, con los siguientes pasos:

Utilizando la semilla aleatoria 123, separar los datos en dos partes, una parte para training (75%)y una parte para test (25%)

A partir de la matriz generada por el script y la información de las clases en human_data.txt, creamos un conjunto de datos que combina el conteo de hexámeros con la clase asignada a cada gen.

Este conjunto de datos se divide en dos partes: un **conjunto de entrenamiento** (75% de los genes) y otro **conjunto de prueba** (25% de los genes), que se utilizarán para evaluar el algoritmo **kNN** que implementaremos posteriormente. La separación se realiza utilizando una semilla aleatoria para garantizar la reproducibilidad.

En esta actividad, debido a la naturaleza de nuestros datos (mismo tipo de variables), no se ha realizado normalización.

```
# Extraemos de la columna "class" de "data_script".
class_labels <- data_script[,2]
# Combina los datos "kmers_count" con las etiquetas anteriores en el conjunto de datos "matriz_knn".
matriz_knn <- as.data.frame(cbind(kmers_count, class_labels))
# Semilla aleatoria para la reproducibilidad.
set.seed(123)
# Tamaño de la muestra de entrenamiento (75% de las filas).
size <- floor(nrow(matriz_knn)*0.75)
# Extraemos los indices de una muestra aleatoria del 75% de la filas de los datos.
train_ind <- sample(seq_len(nrow(matriz_knn)), size = size)
# Extraemos las etiquetas para elegir las filas de los datos de entrenamiento.
data_labels <- matriz_knn[,"class_labels"]
# Vector de etiquetas de clase para el conjunto de entrenamiento.
train_labels <- matriz_knn[train_ind,"class_labels"]
# Vector de etiquetas de clase para el conjunto de prueba
test_labels <- data_labels[-train_ind]
# Dataframe de entrenamiento utilizando los índices generados.
data_train <- matriz_knn[train_ind,]
# Dataframe de prueba utilizando los índices restantes.
data_test <- matriz_knn[-train_ind,]
```

Aplicar el k-nn (k = 1, 3, 5, 7) basado en el training para predecir la familia de las secuencias del test

Una vez que hemos dividido los datos en conjuntos de entrenamiento y prueba, implementamos el algoritmo **kNN** en R utilizando el paquete *'class'*, que proporciona funciones básicas de clasificación. Mediante la función *'knn()'* y con valores específicos de **'k'** (1, 3, 5, 7), generamos una lista que contiene las predicciones de cada valor de **'k'**.

```r
# Semilla aleatoria para la reproducibilidad.
set.seed(123)
# Valores de 'k' para el algoritmo kNN.
# k_values <- c(1, 3, 5, 7)
# Listas para almacenar los modelos y las matrices resultantes.
models <- list()
confusion_matrices <- list()
# Preapramos un bucle para cada valor de 'K':
for (k in k_values) {
  model_name <- paste("model_knn_k", k, sep = "")
  # Entrenamiento del modelo con los datos establecidos para cada valor de 'K':
  model <- knn(train = data_train, test = data_test, cl = train_labels, k = k)
  # Almacenamos cada modelo resultante de cada valor de 'K':
  models[[model_name]] <- model

  # Calculamos la matriz de confusión y almacenamos:
  cm <- confusionMatrix(data = model, reference = as.factor(test_labels))
  confusion_matrices[[model_name]] <- cm
}
# Imprimimos las matrices de confusión:
print(confusion_matrices)
```

```
## $model_knn_k1
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6
##          0  43  16   6  14  12   1   0
##          1   6  58   1   0   3   1   0
##          2   1   0  35   2   0   0   0
##          3   1   1   0  57   0   0   2
##          4   1   0   1   1  76   3   4
##          5   0   0   1   0   1  18   0
##          6   0   2   1   2   3   1 125
##
## Overall Statistics
##
##                Accuracy : 0.824
##                  95% CI : (0.7877, 0.8564)
##     No Information Rate : 0.262
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.7878
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.8269   0.7532   0.7778   0.7500   0.8000   0.7500
## Specificity            0.8906   0.9740   0.9934   0.9906   0.9753   0.9958
## Pos Pred Value         0.4674   0.8406   0.9211   0.9344   0.8837   0.9000
## Neg Pred Value         0.9779   0.9559   0.9784   0.9567   0.9541   0.9875
```

```
## Prevalence             0.1040   0.1540   0.0900   0.1520   0.1900   0.0480
## Detection Rate         0.0860   0.1160   0.0700   0.1140   0.1520   0.0360
## Detection Prevalence   0.1840   0.1380   0.0760   0.1220   0.1720   0.0400
## Balanced Accuracy      0.8588   0.8636   0.8856   0.8703   0.8877   0.8729
##                      Class: 6
## Sensitivity            0.9542
## Specificity            0.9756
## Pos Pred Value         0.9328
## Neg Pred Value         0.9836
## Prevalence             0.2620
## Detection Rate         0.2500
## Detection Prevalence   0.2680
## Balanced Accuracy      0.9649
##
## $model_knn_k3
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6
##          0  39  23   9  19   8   0   0
##          1   5  44   1   0   4   0   2
##          2   2   2  28   4   1   0   4
##          3   1   1   2  45   1   1   1
##          4   4   4   3   3  61   7   6
##          5   0   0   0   0   1  12   0
##          6   1   3   2   5  19   4 118
##
## Overall Statistics
##
##                Accuracy : 0.694
##                  95% CI : (0.6515, 0.7341)
##     No Information Rate : 0.262
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.629
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.7500   0.5714   0.6222   0.5921   0.6421   0.5000
## Specificity            0.8683   0.9716   0.9714   0.9835   0.9333   0.9979
## Pos Pred Value         0.3980   0.7857   0.6829   0.8654   0.6932   0.9231
## Neg Pred Value         0.9677   0.9257   0.9630   0.9308   0.9175   0.9754
## Prevalence             0.1040   0.1540   0.0900   0.1520   0.1900   0.0480
## Detection Rate         0.0780   0.0880   0.0560   0.0900   0.1220   0.0240
## Detection Prevalence   0.1960   0.1120   0.0820   0.1040   0.1760   0.0260
## Balanced Accuracy      0.8092   0.7715   0.7968   0.7878   0.7877   0.7489
##                      Class: 6
## Sensitivity            0.9008
## Specificity            0.9079
## Pos Pred Value         0.7763
## Neg Pred Value         0.9626
```

```
## Prevalence             0.2620
## Detection Rate         0.2360
## Detection Prevalence   0.3040
## Balanced Accuracy      0.9043
##
## $model_knn_k5
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1   2   3   4   5   6
##          0  40  26  13  22  10   0   0
##          1   4  42   1   1   4   1   1
##          2   3   0  23   3   1   0   3
##          3   2   0   4  36   1   1   1
##          4   2   4   2   5  55   5   5
##          5   0   0   0   0   2  10   0
##          6   1   5   2   9  22   7 121
##
## Overall Statistics
##
##                Accuracy : 0.654
##                  95% CI : (0.6105, 0.6957)
##     No Information Rate : 0.262
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5793
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.7692   0.5455   0.5111   0.4737   0.5789   0.4167
## Specificity            0.8415   0.9716   0.9780   0.9788   0.9432   0.9958
## Pos Pred Value         0.3604   0.7778   0.6970   0.8000   0.7051   0.8333
## Neg Pred Value         0.9692   0.9215   0.9529   0.9121   0.9052   0.9713
## Prevalence             0.1040   0.1540   0.0900   0.1520   0.1900   0.0480
## Detection Rate         0.0800   0.0840   0.0460   0.0720   0.1100   0.0200
## Detection Prevalence   0.2220   0.1080   0.0660   0.0900   0.1560   0.0240
## Balanced Accuracy      0.8054   0.7585   0.7446   0.7262   0.7611   0.7062
##                      Class: 6
## Sensitivity            0.9237
## Specificity            0.8753
## Pos Pred Value         0.7246
## Neg Pred Value         0.9700
## Prevalence             0.2620
## Detection Rate         0.2420
## Detection Prevalence   0.3340
## Balanced Accuracy      0.8995
##
## $model_knn_k7
## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction   0   1   2   3   4   5   6
##          0  45  25  12  11   3   0   1
##          1   3  39   1   1   4   2   1
##          2   0   2  23   1   0   0   2
##          3   3   0   3  30   1   0   0
##          4   1   6   4   7  51   6   3
##          5   0   0   0   0   0   9   1
##          6   0   5   2  26  36   7 123
##
## Overall Statistics
##
##                Accuracy : 0.64
##                  95% CI : (0.5962, 0.6821)
##     No Information Rate : 0.262
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5572
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity            0.8654   0.5065   0.5111   0.3947   0.5368   0.3750
## Specificity            0.8839   0.9716   0.9890   0.9835   0.9333   0.9979
## Pos Pred Value         0.4639   0.7647   0.8214   0.8108   0.6538   0.9000
## Neg Pred Value         0.9826   0.9154   0.9534   0.9006   0.8957   0.9694
## Prevalence             0.1040   0.1540   0.0900   0.1520   0.1900   0.0480
## Detection Rate         0.0900   0.0780   0.0460   0.0600   0.1020   0.0180
## Detection Prevalence   0.1940   0.1020   0.0560   0.0740   0.1560   0.0200
## Balanced Accuracy      0.8747   0.7391   0.7501   0.6891   0.7351   0.6864
##                      Class: 6
## Sensitivity            0.9389
## Specificity            0.7940
## Pos Pred Value         0.6181
## Neg Pred Value         0.9734
## Prevalence             0.2620
## Detection Rate         0.2460
## Detection Prevalence   0.3980
## Balanced Accuracy      0.8665
```

Comentar los resultados

k	Accuracy	Kappa	Error_rate
1	0.82	0.79	0.18
3	0.69	0.63	0.31
5	0.65	0.58	0.35
7	0.64	0.56	0.36

Como se observa en las matrices de confusión y en la tabla anterior, el modelo $\boldsymbol{k = 1}$ destaca con una mayor **precisión** y coeficiente **Kappa** (un valor de Kappa más alto indica un mejor rendimiento del modelo)

teniendo un rendimiento superior en comparación con los demás modelos. En este caso, el modelo **k = 1** tiene la mayor precisión con un valor de 0.82, lo que significa que el 82% de las observaciones se clasifican correctamente. A medida que aumenta el valor de ***k***, tanto la precisión como el coeficiente Kappa tienden a disminuir.

También es importante observar la **tasa de error** que es el complemento de la precisión y representa la proporción de observaciones clasificadas incorrectamente. Solo el modelo **k = 1** presenta una tasa de error por debajo del 20% (18%), indicando mayor rendimiento en términos de errores de clasificación. A medida que aumenta el valor de ***k***, la tasa de error aumenta, suguiriendo un peor rendimiento en la clasificación.

Además, en el apartado **"Statistics by Class"** de cada matriz de confusión, se pueden encontrar otros parámetros que evalúan la capacidad del modelo para clasificar observaciones, identificando tanto verdaderos positivos como verdaderos negativos, y midiendo la precisión de las predicciones positivas y negativas para cada una de las clases.

En resumen, los resultados muestran que el modelo con **k = 1** tiene el mejor rendimiento en función de los parametros mostrados en las matrices de confusión.

Para las secuencias de las familias: 0 (=G protein coupled receptors) y 1(=Tyrosine kinase)

```
#Separar del data frame la clase 0 y 1.
data_gt <- matriz_knn[matriz_knn$class_labels == 0 | matriz_knn$class_labels == 1,]
# Preparamos los datos de entranamiento/prueba, como en el apartado anterior.
set.seed(123)
size_gt <- floor(nrow(data_gt)*0.75)
train_ind_gt <- sample(seq_len(nrow(data_gt)), size = size_gt)
data_labels_gt <- data_gt[,"class_labels"]
train_labels_gt <- data_gt[train_ind_gt,"class_labels"]
test_labels_gt <- data_labels_gt[-train_ind_gt]
data_train_gt <- data_gt[train_ind_gt,]
data_test_gt <- data_gt[-train_ind_gt,]
```

```
set.seed(123)
# Creamos las listas para almacenar los resultados:
models_gt <- list()
confusion_matrices_gt <- list()
# Bucle para ajustar el modelo KNN con diferentes valores de k
for (k in k_values) {
  model_name_gt <- paste("model_knn_gt_k", k, sep = "")
  # Entrenamiento del modelo con los datos establecidos para cada valor de 'K':
  model_gt <- knn(train = data_train_gt, test = data_test_gt, cl = train_labels_gt, k = k)
  # Almacenamos cada modelo resultante de cada valor de 'K':
  models_gt[[model_name_gt]] <- model_gt
   # Calculamos la matriz de confusión y almacenamos:
  cm_gt <- confusionMatrix(data = model_gt, reference = as.factor(test_labels_gt))
  confusion_matrices_gt[[model_name_gt]] <- cm_gt
}
# Imprimimos las matrices de confusión:
print(confusion_matrices_gt)
```

```
## $model_knn_gt_k1
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 49 12
##          1  6 53
##
##                Accuracy : 0.85
##                  95% CI : (0.7733, 0.9086)
##     No Information Rate : 0.5417
##     P-Value [Acc > NIR] : 7.529e-13
##
##                   Kappa : 0.7004
##
##  Mcnemar's Test P-Value : 0.2386
##
##             Sensitivity : 0.8909
##             Specificity : 0.8154
##          Pos Pred Value : 0.8033
##          Neg Pred Value : 0.8983
##              Prevalence : 0.4583
##          Detection Rate : 0.4083
##    Detection Prevalence : 0.5083
##       Balanced Accuracy : 0.8531
##
##        'Positive' Class : 0
##
##
## $model_knn_gt_k3
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 45 17
##          1 10 48
##
##                Accuracy : 0.775
##                  95% CI : (0.6898, 0.8462)
##     No Information Rate : 0.5417
##     P-Value [Acc > NIR] : 9.756e-08
##
##                   Kappa : 0.5512
##
##  Mcnemar's Test P-Value : 0.2482
##
##             Sensitivity : 0.8182
##             Specificity : 0.7385
##          Pos Pred Value : 0.7258
##          Neg Pred Value : 0.8276
##              Prevalence : 0.4583
##          Detection Rate : 0.3750
##    Detection Prevalence : 0.5167
##       Balanced Accuracy : 0.7783
##
```

```
##        'Positive' Class : 0
##
##
## $model_knn_gt_k5
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 43 20
##          1 12 45
##
##                Accuracy : 0.7333
##                  95% CI : (0.6449, 0.8099)
##     No Information Rate : 0.5417
##     P-Value [Acc > NIR] : 1.243e-05
##
##                   Kappa : 0.4689
##
##  Mcnemar's Test P-Value : 0.2159
##
##             Sensitivity : 0.7818
##             Specificity : 0.6923
##          Pos Pred Value : 0.6825
##          Neg Pred Value : 0.7895
##              Prevalence : 0.4583
##          Detection Rate : 0.3583
##    Detection Prevalence : 0.5250
##       Balanced Accuracy : 0.7371
##
##        'Positive' Class : 0
##
##
## $model_knn_gt_k7
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 42 18
##          1 13 47
##
##                Accuracy : 0.7417
##                  95% CI : (0.6538, 0.8172)
##     No Information Rate : 0.5417
##     P-Value [Acc > NIR] : 5.135e-06
##
##                   Kappa : 0.4833
##
##  Mcnemar's Test P-Value : 0.4725
##
##             Sensitivity : 0.7636
##             Specificity : 0.7231
##          Pos Pred Value : 0.7000
##          Neg Pred Value : 0.7833
##              Prevalence : 0.4583
```

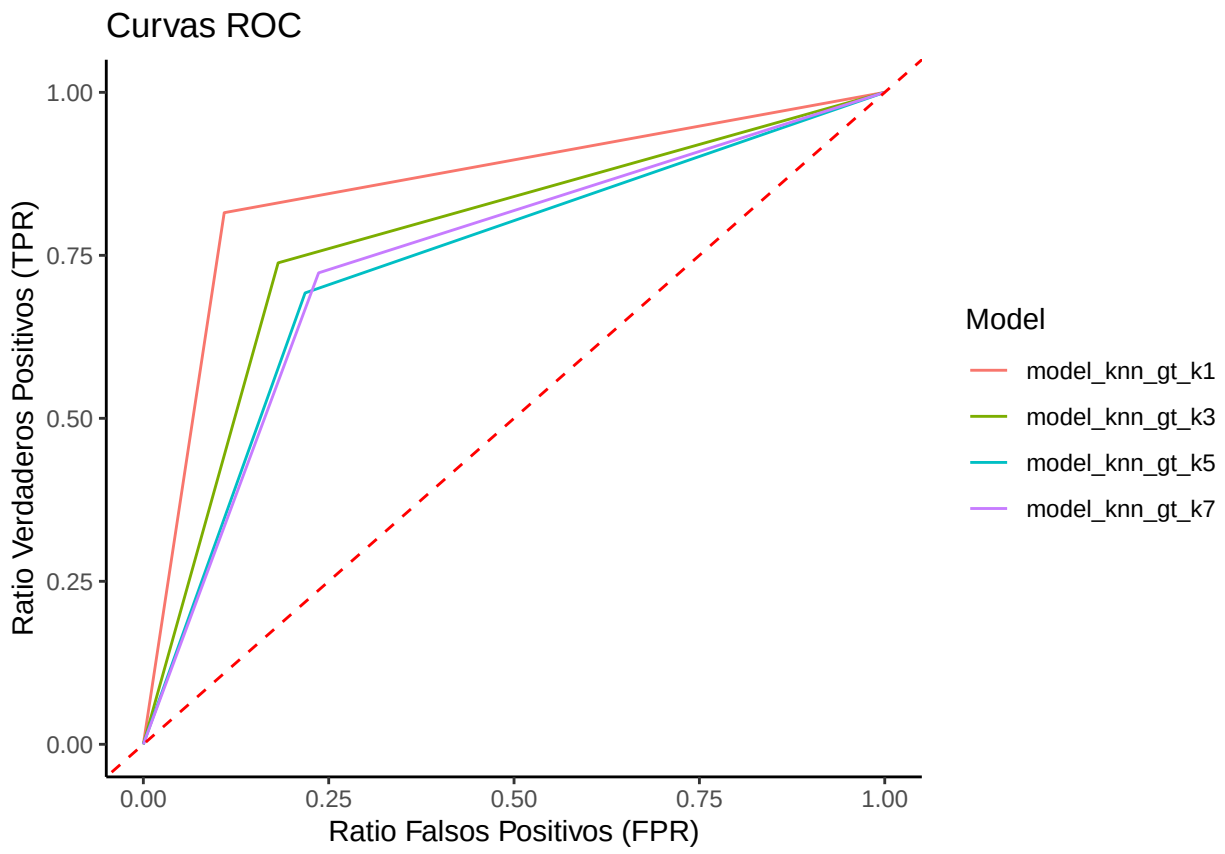
```
##          Detection Rate : 0.3500
##    Detection Prevalence : 0.5000
##       Balanced Accuracy : 0.7434
##
##        'Positive' Class : 0
##
```

Representar la curva ROC para cada valor de k = (1, 3, 5, 7)

Una curva **ROC** es una gráfica que muestra cómo un modelo de clasificación realiza la clasificacion de clases. En el eje X, muestra la tasa de falsos positivos (FPR), y en el eje Y, muestra la tasa de verdaderos positivos (TPR). El modelo que tenga una curva ROC más proxima a la esquina superior izquierda tendra una mejor ajuste en la claisificacion.

Para calcular la curva ROC, utilizamos el paquete *'ROC'*. Con las funciones de este paquete, se han calculado los parámetros necesarios para trazar las curvas correspondientes a cada valor de **'k'**. Se ha establecido como clase positiva **"0"(=G)** para receptores acoplados a proteínas.

```r
# Calculamos la tasa de verdaderos positivos (TPR) y la tasa de falsos positivos (FPR) para cada modelo
roc_models_gt <- lapply(models_gt, function(model) as.numeric(as.character(model)))
# Generamos las predicciones para los datos de prueba.
predictions_gt <- lapply(roc_models_gt, function(model) prediction(model, labels = test_labels_gt))
# Calculamos el rendimiento de la curva ROC para cada modelo.
performances_gt <- lapply(predictions_gt, function(model) performance(model, "tpr", "fpr"))
# Creamos el conjunto de datos para pintar las curvas ROC.
roc_data <- data.frame(
  Model = rep(names(performances_gt), each = length(performances_gt$model_knn_gt_k1@x.values[[1]])),
  TPR = unlist(lapply(performances_gt, function(model) model@y.values[[1]])),
  FPR = unlist(lapply(performances_gt, function(model) model@x.values[[1]]))
)
# Creamos el gráfico ROC con ggplot2
roc_plot <- ggplot(roc_data, aes(x = FPR, y = TPR, color = Model)) +
  geom_line() +
  labs(x = "Ratio Falsos Positivos (FPR)", y = "Ratio Verdaderos Positivos (TPR)" ) +
  ggtitle("Curvas ROC") +
  geom_abline(intercept = 0, slope = 1, linetype = "dashed", color = "red") +
  theme_classic()
# Imprimimos el gráfico.
print(roc_plot)
```



Comentar los resultados de la clasificación en función de la curva ROC y del número de falsos positivos, falsos negativos y error de clasificación obtenidos para los diferentes valores de k

	k	Accuracy	Error_rate	FP	FN
1	1	0.85	0.18	12	6
2	3	0.78	0.31	17	10
4	7	0.74	0.36	18	13
3	5	0.73	0.35	20	12

Con el gráfico y la tabla de parámetros para los nuevos modelos kNN, podemos observar que el modelo con k = 1 sigue siendo el que mejor se ajusta. En el gráfico ROC, vemos que este modelo se acerca más a la esquina superior izquierda, tiene una precisión del 85% y un menor número de falsos positivos y falsos negativos (12/6) en comparación con otros valores de **'k'**.

En el caso de estos modelos más simples, destacamos que el valor de k = 7 presenta una mejor clasificación en comparación con el modelo de múltiples clases realizado en el apartado anterior para el mismo valor de **'k'**. En la gráfica ROC, podemos observar que la curva de este modelo se ajusta mejor que la del valor k = 5. A pesar de esto, sigue presentando una tasa de error mínimamente mayor.

En conclusión, el valor de k = 1 sigue mostrando el mejor rendimiento para predecir la familia de las secuencias de los hexámeros evaluados, ya sea en un algoritmo multiclase o en uno de clases binarias. Este valor de **'k'** obtiene lla mayor precisión, la menor tasa de error y el menor número de observaciones clasificadas como falsos positivos y falsos negativos de todas las clasificaciones realizadas.

Referencias

Lantz, Brett. 2019. *Machine Learning with r: Expert Techniques for Predictive Modeling*. Packt publishing ltd.