PEC4: Clasificacion de splice junctions

Machine Learning

Jessica Vera

1/12/2022

Contents

1 Resumen

El presente documento utiliza la potencia de las herramientas de *machine learning* en la identificación de límites de exones e introns dada una secuencia de ADN. Para ello, recibiremos un archivo con las secuencias en *one-hot encoding*; de modo que cada caracter codificado de las secuencias representan una columna. Posteriormente usaremos esa información para reducir la dimensionalidad mediante un *autoencoder convolucional*. Con las nuevas coordenadas de los datos, analizaremos estos datos mediante la implementación de los diferentes algoritmos estudiados: *k-Nearest Neighbour*, *Naive Bayes*, *Support Vector Machine*, á*rbol de Decisión* y *Random Forest* para predecir el tipo de splice junctions.

2 Analisis

2.1 Lectura de los datos

Los datos a importar del fichero splice.csv, corresponden a secuencias codificadas con *one-hot encoding*, las cuales se encuentran categorizadas, según si contienen un límite exón/intrón (EI), un límite intrón/exón (IE) o ninguno (N). En este sentido, cada secuencia presenta sus caracteres codificados y cada caracter representa

una columna. Las columnas correspondientes a las secuencias son binarias, según la codificación ***one-hot***. Para iniciar, importaremos el conjunto de datos mediante la función `read.csv()`.

```
# definiendo el archivo
fichero <- ifelse(params$folderdata=="", # si no se especifica ruta
                  # usar como fichero directamente al archivo
                   params$file1,
                  # caso contrario, usar la ruta.archivo
                   file.path(params$folderdata,params$file1))

# importaci$\'{o}$n del fichero
splice <- read.csv(file=fichero)
```

```
# dimensiones del dataset
dim(splice)
```

```
[1] 3190  482
```

Este dataset contiene 3190 secuencias y 482 columnas, las cuales especifican:

1) `class`: La clase de la secuencia: EI (límites exón/intrón), IE (límites intrón/exón) y N (no incluye sitio de *splicing*)
2) `seq_name`: Nombre identificador de la secuencia
3) $V_i, i = 1, 2, ..., 480$: 482 columnas de los caracteres codificados con ***one-hot encoding***.

Definiremos como factor a la clase de límite que contiene la secuencia.

```
splice$class <- as.factor(splice$class)
```

2.2 Division de la muestra

Dividiremos aleatoriamente la muestra en dos: 67% de los datos será de entrenamiento *training* y el restante se usará para *test*. Usaremos la función `createDataPartition()` para asegurarnos que la proporción de ejemplos en cada clase sea similar a la del conjunto de datos originales.

```
# Establecer semilla
set.seed(params$seed.train)

# Aleatorizar posiciones para datos de entrenamiento
index_train <- createDataPartition(splice$class, p = params$p.train, list = FALSE)
index_train <- as.numeric(index_train)
# Particion
datatrain <- splice[index_train,-2] # contendran las posiciones de training
datatest <- splice[-index_train,-2] # todas las posiciones menos las de training
```

Podemos corroborar que las muestras de *training* y *test* tienen distribución similar a la del dataset original.

```
## Corroborar dimensiones

# dataset completo
table(splice$class)/dim(splice)[1]
```

```
       EI        IE         N
0.2404389 0.2407524 0.5188088
```

```
# training
table(datatrain$class)/dim(datatrain)[1]
```

```
       EI        IE         N
0.2406015 0.2406015 0.5187970
```

```
# test
table(datatest$class)/dim(datatest)[1]
```

```
       EI        IE         N
0.2401130 0.2410546 0.5188324
```

2.3 Reduccion de dimensiones

Dada la cantidad de variables, haremos uso de una arquitectura de *autoencoder convolucional* para reducir dimensiones.

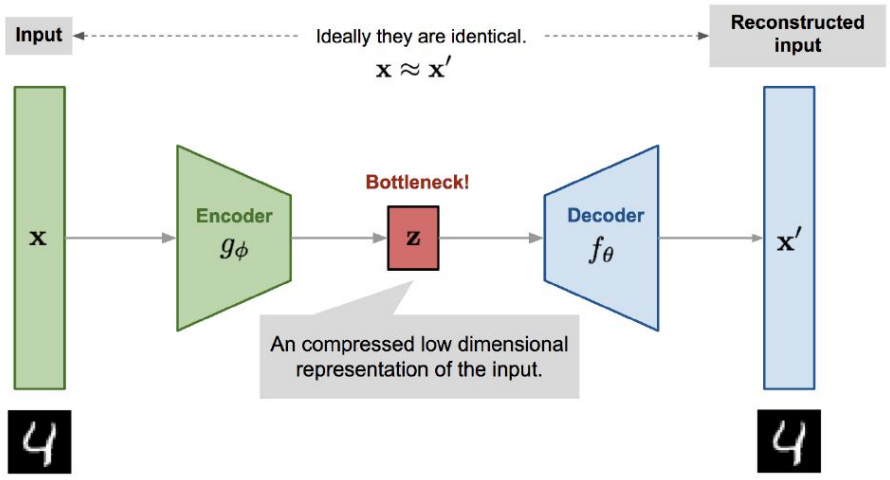

Figure 1: Esquema general de un autoencoder

El esquema del que haremos uso, contará con las siguientes capas:

- capa input
- capa de convolución
- capa de aplanado (*flatten*) que proporciona el (*bottleneck*)
- capa de convolución transpuesta
- capa output

Haremos uso de la partición que realizamos previamente para definir la arquitectura. El autoencoder es útil para trabajar en datos no supervisados, por lo que no utilizaremos la variable de respuesta para esta arquitectura. únicamente tomaremos las variables predictoras de `train` y `test` y las convertiremos en matriz, mediante el comando `as.matrix()`.

```
## Definicion de los grupos que usaremos
# grupo de variables predictoras de la muestra de entrenamiento
train.x <- as.matrix(datatrain[,-1])
# grupo de variables predictoras de la muestra de prueba
test.x <- as.matrix(datatest[,-1])
```

A continuación definiremos las capas. Usaremos como capa de *input* a las 480 columnas correspondientes a las secuencias codificadas. Es necesario especificar una dimensión más para la capa de entrada: 480, 1. Para definirla usaremos la función `layer_input()`.

```
## definiendo los nodos
# nodos de ingreso
ae.inputlayer <- layer_input(shape = c(dim(train.x)[2], 1))
```

```
Loaded Tensorflow version 2.7.0
```

Para la capa *encoder*, usamos como entrada la capa de *input* y le añadimos una capa de convolución, mediante la función `layer_conv_1d()` porque los datos en análisis no son de más de una dimensión. Para la primera capa de convolución usamos 20 filtros, 3 como tamaño de kernel, la función de activación `relu`.

También, añadimos una capa de agrupamiento, con máx pooling, mediante la función `layer_max_pooling_1d()`. Esta capa usa un agrupamiento de tamaño 2.

La capa cuello de bottella o *bottleneck* la obtenemos mediante una capa de convolución de un filtro para obtener un único nodo. Esto lo lograremos mediante la función `layer_conv_1d()` con parámetro `filter=1, kernel_size=3, activation="relu"`. Previamente, estableceremos una semilla con argumento 123.

```
## definiendo los nodos
# encoder
set.seed(params$seed.train)
ae.encoder <-
  # input
  ae.inputlayer %>%
  # capa de convolucion
  layer_conv_1d(filters = 20, kernel_size = 3, activation="relu", padding="same",
                input_shape = c(dim(train.x)[2])) %>%
  # capa de max_pooling
  layer_max_pooling_1d(pool_size = 2, padding = "same") %>%
  # capa de convolucion - bottleneck
  layer_conv_1d(filters = 1, kernel_size = 3, activation="relu", padding="same")
```

Para la capa *decoder*, usamos como entrada la capa de *encoder* y realizamos la acción contraria que en el encoder. Esta vez iremos aumentando el número de nodos hasta volver al número de dimensiones originales. Para lo cual, añadimos una capa de convolución, mediante la función `layer_conv_1d()` y usamos 20 filtros, 3 como tamaño de kernel, la función de activación `relu`. Para cumplir con la convolución transpuesta, usaremos la función `layer_upsampling_1d()`. Esta contendrá el mismo número de agrupamientos que realizó la capa de agrupamiento de máx pooling, 2. Por último, la capa de salida será una capa de convolución de un filtro para no reducir el número de dimensiones obtenidas hasta la capa anterior. La función de activación que usaremos será la función `sigmoid`. Previamente, estableceremos una semilla con argumento 123.

```
## definiendo los nodos
# decoder
set.seed(params$seed.train)
```

```r
ae.decoder <-
  # input = encoder hasta el bottleneck
  ae.encoder %>%
  # capa de convolucion - retroceso
  layer_conv_1d(filters = 20, kernel_size = 3, activation="relu", padding="same") %>%
  # retroceso del max.pooling
  layer_upsampling_1d(2) %>%
  # capa de salida
  layer_conv_1d(filters = 1, kernel_size = 3, activation="sigmoid", padding="same")
```

Revisemos el modelo completo. Para ello, indicaremos el *input* y *output* del modelo en la función `keras_model()`. Previamente, estableceremos una semilla con argumento 123.

Podemos observar que en el *autoencoder* disminuimos el número de nodos de 480 a 240 y luego aumenta a 480 nuevamente.

```r
# todo el autoencoder
set.seed(params$seed.train)
autoencoder_model <- keras_model(inputs = ae.inputlayer, outputs = ae.decoder)

# compilacion de modelo
autoencoder_model %>% compile(
  loss='mean_squared_error',
  optimizer='adam',
  metrics = c('accuracy')
)

# resumen
summary(autoencoder_model)
```

```
Model: "model"
_________________________________________________________________________________________
 Layer (type)                            Output Shape                          Param #
=========================================================================================
 input_1 (InputLayer)                    [(None, 480, 1)]                      0

 conv1d_1 (Conv1D)                       (None, 480, 20)                       80

 max_pooling1d (MaxPooling1D)            (None, 240, 20)                       0

 conv1d (Conv1D)                         (None, 240, 1)                        61

 conv1d_3 (Conv1D)                       (None, 240, 20)                       80

 up_sampling1d (UpSampling1D)            (None, 480, 20)                       0

 conv1d_2 (Conv1D)                       (None, 480, 1)                        61

=========================================================================================
Total params: 282
Trainable params: 282
Non-trainable params: 0
_________________________________________________________________________________________
```

Verifiquemos el desempeño del modelo en el grupo de prueba, analizando el historial de épocas.

```
# fit the model on the train data
set.seed(params$seed.train)
ae.history <- autoencoder_model %>%
  keras::fit(test.x, test.x, epochs=20, batch_size=120)
```

Podemos observar en la Figura ?? que la precisión alcanzada está por encima de 0.8 con una media cuadrática del error de menos de 0.1.

```
# Historial de epocas
plot(ae.history)
```

```
`geom_smooth()` using formula 'y ~ x'
```

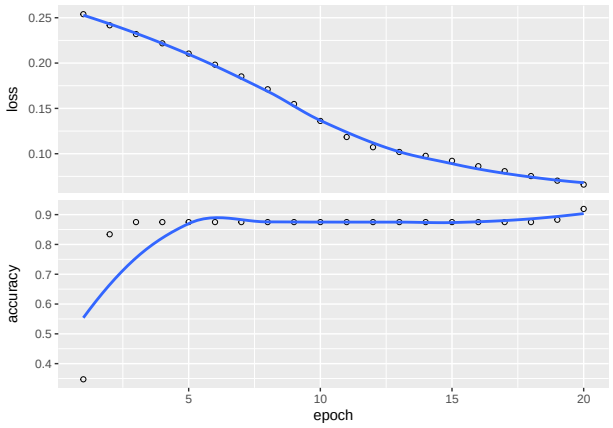

Figure 2: Evaluacion del rendimiento del autoencoder en la uestra de prueba por epocas

Dado que nuestro interés es reducir dimensiones. Usaremos los resultados de la sección del *encoder*. Realizaremos la transformación sobre todo el conjunto de datos.

```
### usando solo la parte del encoder
set.seed(params$seed.train)
encoder_model <- keras_model(inputs = ae.inputlayer, outputs = ae.encoder)

# transformaciones sobre las predictoras del dataset completo.
embeded_points <- encoder_model %>%
  keras::predict_on_batch(x = as.matrix(splice[,-c(1,2)]))

# conversion a data.frame
data_transf <- as.data.frame(embeded_points[, , 1])

# dimensiones
dim(data_transf)
```

```
[1] 3190  240
```

De este modo, pasamos de 480 a 240 columnas y con la transformación, los valores cambian de ser binarios a reales, como observamos en 3

```r
# distribucion de los valores transformados
hist(c(embeded_points), main = "Histogram of encoded values",
     xlab = "Encoded values")
```

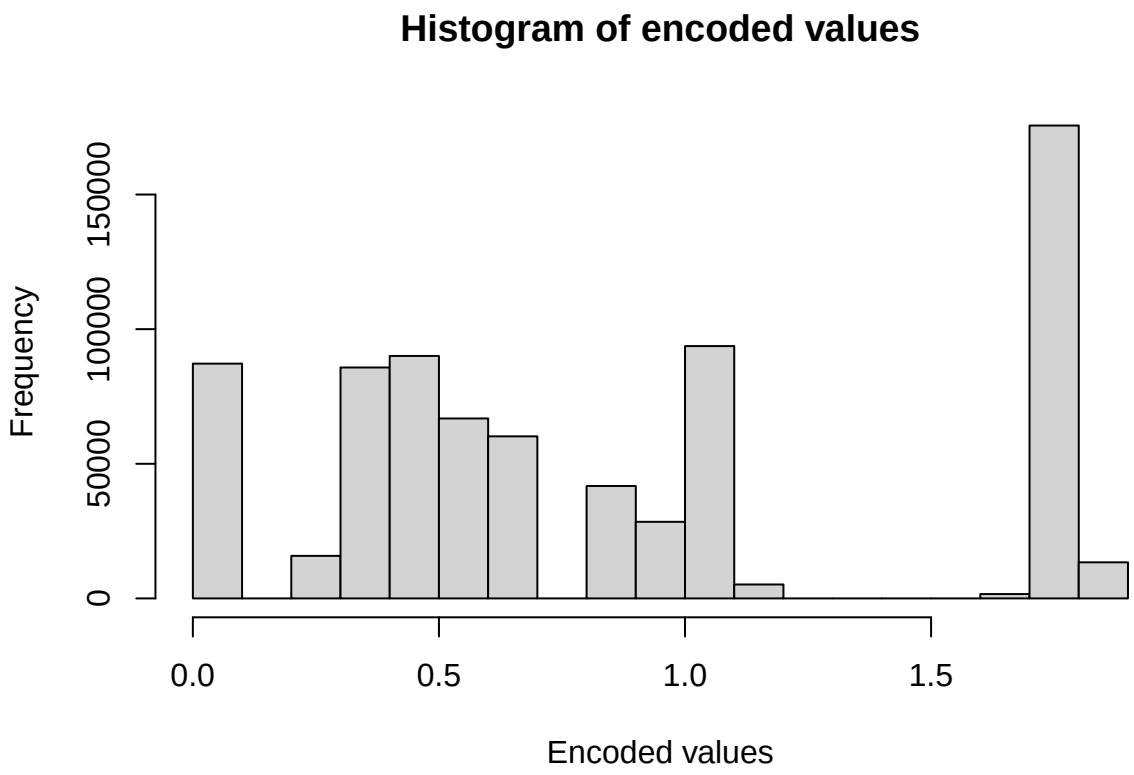

Figure 3: Histograma de los valores transformados

Los datos transformados se unen a la clase que le corresponde y exportaremos este dataset para almacenarlo, mediante la función `write.csv()`.

```r
# unir los datos transformados con la clase a la que pertenecen
data_transf$clase <- splice$class
```

2.4 Modelos de prediccion

En esta sección aplicaremos los algoritmos para la clasificación, donde se analizarán las secuencias en las nuevas coordenadas. Está formado por subsecciones que corresponden a cada algoritmo. Con fines de asegurar reproducibilidad en los resultados, no usaremos el archivos de datos que creamos con el *autoencoder*, sino que emplearemos el fichero `deep_descriptors.csv` que contiene tamién una transformación de datos con un número de columnas reducido.

```r
# definiendo el archivo
fichero2 <- ifelse(params$folderdata=="",
                   params$file2,
```

```
                file.path(params$folderdata,params$file2))

# importacion del fichero
splice.transf <- read.csv(file=fichero2)
#splice.transf <- read.csv("deep_descriptors.csv")
# dimensiones
dim(splice.transf)
```

```
[1] 3190  321
```

Adicionalmente nos aseguraremos de que la clase esté claramente categorizada

```
# definiendo como factor
splice.transf$class <- factor(splice.transf$class,
                              levels = names(table(splice.transf$class)),
                              labels=c("EI", "IE", "N"))
# niveles del factor respuesta
niveles <- levels(splice.transf$class)
```

Adicionalmente, usaremos la división de la muestra, tal como se describió en la sección de división de la muestra.

```
## division de la muestra transformada
# variables predictoras en entrenamiento
xtrain <- splice.transf[index_train,-1]
# variables predictoras en prueba
xtest <- splice.transf[-index_train,-1]
# variable de respuesta en entrenamiento
ytrain <- splice.transf[index_train,1]
# variable de respuesta en prueba
ytest <- splice.transf[-index_train,1]
# dataset train
df.train <- splice.transf[index_train,]
# dataset test
df.test <- splice.transf[-index_train,]
```

También, crearemos una función genérica, `summaryconfmat()`, para resumir ciertas métricas de las matrices de confusión que nos servirán para comparar entre modelos. El detalle de esta función se incluye en la sección del Apéndice.

2.4.1 k-Nearest Neighbour

En este apartado usaremos el algoritmo *k-Nearest Neighbour* para predecir la clase de límite que contiene una secuencia.

El algoritmo k-NN (*k-nearest neighbor*) es un método perteneciente a los clasificadores del vecino más cercano (*nearest neighbors*) y toma en consideración la clase de los k ejemplos más similares.

El algoritmo, por tanto, inicia con la selección de un valor de k. Exploraremos los valores para el número de vecinos $k = 1, 3, 5, 7$.

Para ello, crearemos una función, `knnperform()` que podremos usar posteriormente también para determinar la clase de límite que contienen las secuencias. Denominaremos a esta función `knnperform()` y lo que hará será determinar el desempeño en la predicción del algoritmo k-NN, dado un determinado valor k.

La función ejecuta un algoritmo k-NN con la función `knn()` del paquete `class`. Posteriormente emplea la función `confusionMatrix()` del paquete `caret` para determinar los diferentes valores de desempeño: *accuracy*, *kappa*, tasa de error (*error rate*), *sensitivity* o *recall*, *specificity* y *precision*. El detalle de esta función se incluye en la sección del Apéndice.

Aplicando la función creada `knnperform()`, obtenemos los siguientes resultados para los distintos valores de k definidos. Podemos observar que todos los indicadores reportados incrementan a mayor valor de k, `k=7`.

```
k1 <- c(1,3,5,7)
(knn.seq <- knnperform(xtrain, xtest, ytrain, ytest, k1, niveles))
```

```
           k=1   k=3   k=5   k=7
Accuracy 0.669 0.681 0.704 0.727
Kappa    0.495 0.516 0.548 0.580
Error    0.331 0.319 0.296 0.273
Sens.EI  0.716 0.753 0.749 0.765
Sens.IE  0.806 0.825 0.857 0.865
Sens.N   0.589 0.586 0.617 0.649
Spec.EI  0.814 0.812 0.833 0.840
Spec.IE  0.832 0.831 0.842 0.856
Spec.N   0.873 0.903 0.901 0.915
Prec.EI  0.534 0.543 0.571 0.587
Prec.IE  0.599 0.603 0.628 0.651
Prec.N   0.841 0.874 0.877 0.898
```

El mejor modelo presenta una precisión global de 72.7%. A nivel de `k=7`, la tasa de verdaderos positivos (Sensitivity) es menor para los casos en que no hay límite de splicing alguno, 0.649. También, podemos observar que del total de secuencias predichas como "EI," el 58.7% estaría correctamente clasificado. De forma similar, el 65.1% de los predichos como límite "IE," estaría correctamente predicho. Estos son de los indicadores más deficientes del modelo.

2.4.2 Naive Bayes

En esta sección aplicaremos un algoritmo de Naive Bayes, el cual se basa en el teorema de Bayes para resolver problemas de clasificación. Este método asume independencia entre las características predictoras. Para realizar este análisis, emplearemos la función `naiveBayes()` del paquete `e1071`.

Exploraremos el parámetro *laplace* para corregir aquellos casos en que los predictores tengan ausencia de ejemplos, llevando a probabilidades nulas por dicha ausencia.

Iniciaremos creando los modelos, uno sin considerar la corrección de *Laplace*, mediante el argumento `laplace = 0` en la función de `naiveBayes()`; y el otro considerando la corrección (`laplace = 1`).

```
# Modelo con laplace = 0
naivenolaplace <- naiveBayes(x = xtrain, y = ytrain, laplace = 0)
# Modelo con laplace = 1
naivesilaplace <- naiveBayes(x = xtrain, y = ytrain, laplace = 1)
```

Evaluemos el desempeño de los modelos usando la data de prueba.

```
# prediccion del modelo sin correccion
naivetest0 <- predict(naivenolaplace, xtest)
# prediccion del modelo con correccion
naivetest1 <- predict(naivesilaplace, xtest)
```

Verifiquemos el desempeño de los modelos, mediante las principales métricas. Usaremos la función que creamos `summaryconfmat()`.

```
# modelo sin correccion
naivecm0 <- caret::confusionMatrix(naivetest0, ytest)
sum.naive0 <- summaryconfmat(naivecm0, niveles, "laplace0")

# modelo con correccion
naivecm1 <- caret::confusionMatrix(naivetest1, ytest)
sum.naive1 <- summaryconfmat(naivecm1, niveles, "laplace1")

# matriz de desempe$\~{n}$o de los modelos
(naive.seq <- cbind(sum.naive0, sum.naive1))
```

```
         laplace0 laplace1
Accuracy    0.844    0.844
Kappa       0.748    0.748
Error       0.156    0.156
Sens.EI     0.815    0.815
Sens.IE     0.885    0.885
Sens.N      0.838    0.838
Spec.EI     0.926    0.926
Spec.IE     0.921    0.921
Spec.N      0.917    0.917
Prec.EI     0.764    0.764
Prec.IE     0.777    0.777
Prec.N      0.921    0.921
```

Podemos observar que el desempeño de ambos modelos es el mismo y esto es debido a que, las clases formadas para las variables predictoras no presentan ausencia de ejemplos, considerando la discretización que realiza el algoritmo para cada característica predictora. Por tanto, la corrección de *laplace* no es necesaria. La precisión del modelo es 84.4% y el valor de kappa es 0.748. Además, los indicadores de predicción de cada una de las clases es eficiente. La tasa de verdaderos positivos de cada clase supera el 0.8. En términos generales, este algoritmo realizó mejores predicciones que el algoritmo kNN.

2.4.3 Support vector machine

La máquina de soporte vectorial es un algoritmo de aprendizaje supervisado que pretende encontrar un hiperplano multidimensional (dependiendo del número de características que se tengan en el conjunto de datos) que distinga los grupos de entrenamiento entre los puntos de datos, de modo que los grupos que se formen sean homógeneos dentro de ellos. El algoritmo busca el hiperplano que maximice la distancia entre los grupos encontrados.

Su uso puede extenderse a grupos que no son linealmente separables. Para estos casos puntuales se emplea una función Kernel, que pudier ser lineal, polinomial, gausiana y sigmoide.

En este ejercicio exploraremos las funciones *kernel lineal* y *rbf*.

2.4.3.1 Modelo Lineal svmLinear Para este modelo usaremos como parámetros al método `svmLinear` para realizar una clasificación lineal sobre el conjunto de datos `df.train`. Así también, escalaremos las variables predictoras, mediante el argumento `preProcess`; y definiremos el control del entrenamiento de *5-fold crossvalidation*, en el parámetro `trControl`.

```r
# numero de cross-validation
numero.cv <- 5
# definir semilla
set.seed(params$seed.train)
# entrenamiento del modelo
modelosvmlineal <- caret::train(class ~ ., df.train, method = 'svmLinear',
                trControl= trainControl("cv", number = numero.cv),
                preProcess = c("center","scale"),
                tuneGrid= NULL, trace = FALSE)
modelosvmlineal
```

```
Support Vector Machines with Linear Kernel

2128 samples
 320 predictor
   3 classes: 'EI', 'IE', 'N'

Pre-processing: centered (320), scaled (320)
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 1702, 1702, 1701, 1704, 1703
Resampling results:

  Accuracy   Kappa
  0.8543291  0.766081

Tuning parameter 'C' was held constant at a value of 1
```

```r
# resultados
result.svmlineal <- modelosvmlineal$results
```

La información que devuelve el modelo nos menciona que la precisión de la predicción realizada es 0.854 en el remuestreo; así como una tasa de coincidencias no debidas al azar (kappa) de 0.766. Así también, podemos verificar que el costo es el definido por defecto: 1.

Ahora evaluemos el modelo en la data de prueba y observemos sus resultados en la subsección de comparación.

```r
# predicciones
svmlineal.pred <- predict(modelosvmlineal, df.test)
# confusion matrix
confmat.svmlineal <- confusionMatrix(svmlineal.pred, df.test$class)
# resumen de la matriz de confusion
sum.svmlineal <- summaryconfmat(confmat.svmlineal, niveles, "svmLineal")
```

2.4.3.2 Modelo no-lineal svmRadial Para este modelo usaremos como parámetros al método `svmRadial` para realizar una clasificación no lineal sobre el conjunto de datos `df.train`. Así también, escalaremos las variables predictoras, mediante el argumento `preProcess`. Definiremos el control del entrenamiento de 5-fold crossvalidation, en el parámetro `trControl`.

```r
# definir semilla
set.seed(params$seed.train)
# entrenamiento del modelo
modelonolineal <- caret::train(class ~ ., df.train, method = 'svmRadial',
```

```
               trControl= trainControl("cv", number = numero.cv),
               preProcess = c("center","scale"),
               tuneGrid= NULL, trace = FALSE)
modelonolineal
```

```
Support Vector Machines with Radial Basis Function Kernel

2128 samples
 320 predictor
   3 classes: 'EI', 'IE', 'N'

Pre-processing: centered (320), scaled (320)
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 1702, 1702, 1701, 1704, 1703
Resampling results across tuning parameters:

  C     Accuracy   Kappa
  0.25  0.8655823  0.7772846
  0.50  0.8886192  0.8190970
  1.00  0.9013185  0.8402180

Tuning parameter 'sigma' was held constant at a value of 0.001594704
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.001594704 and C = 1.
```

```
# resultados
result.svmnolineal <- cbind(modelonolineal$results[3,],
                            modelonolineal$bestTune)
```

La información que devuelve el modelo nos menciona que la precisión de la predicción realizada es 0.901 en el remuestreo; así como una tasa de coincidencias no debidas al azar (kappa) de 0.84. Así también, podemos verificar que el costo óptimo es 1 y el valor de sigma óptimo es 0.002.

Ahora evaluemos el modelo en la data de prueba y observemos sus resultados en la subsección de comparación.

```
# predicciones
svmnolineal.pred <- predict(modelonolineal, df.test)
# confusion matrix
confmat.svmnolineal <- confusionMatrix(svmnolineal.pred, df.test$class)
# resumen de la matriz de confusion
sum.svmnolineal <- summaryconfmat(confmat.svmnolineal, niveles, "rbf")
```

2.4.3.3 Evaluacion y comparacion

Observemos los resultados de desempeño y comparemos los modelos.

```
# matriz de desempe$\~{n}$o de los modelos
(svm.seq <- cbind(sum.svmlineal, sum.svmnolineal))
```

```
         svmLineal   rbf
Accuracy     0.866 0.920
Kappa        0.782 0.868
Error        0.134 0.080
```

```
Sens.EI      0.856 0.885
Sens.IE      0.849 0.917
Sens.N       0.878 0.937
Spec.EI      0.928 0.976
Spec.IE      0.949 0.964
Spec.N       0.915 0.927
Prec.EI      0.779 0.915
Prec.IE      0.839 0.888
Prec.N       0.922 0.937
```

El modelo no lineal resultó más adecuado ya que presenta mayor precisión, 92%, y el valor de kappa es 0.868. Además, los indicadores de predicción de cada una de las clases es eficiente. Tanto la sensibilidad como la especificidad de cada clase está próxima o por encima de 0.9. El modelo de *support vector machine* no lineal con el método *rbf* superó en indicadores al modelo naive Bayes.

2.4.4 Arboles de decision

A continuación exploraremos la potencia de los árboles de decisión en la predicción de clases de límites de las secuencias. Este algoritmo tiene la ventaja de que selecciona las predictoras con mayor relación a la variable de respuesta, de modo que solo estas queden incluidas en el modelo.

Se explorará la opción de activar o no *boosting*.

2.4.4.1 Sin activacion de boosting Iniciaremos creando el modelo estableciendo la semilla y entrenándolo mediante la función `C5.0()`.

```
# fijar semilla
set.seed(params$seed.train)
# modelo sin boosting
treemod.noboost<- C5.0(x = xtrain, y = ytrain)
treemod.noboost
```

```
Call:
C5.0.default(x = xtrain, y = ytrain)

Classification Tree
Number of samples: 2128
Number of predictors: 320

Tree size: 195

Non-standard options: attempt to group attributes
```

De acuerdo con los resultados del modelo, de los 320 predictores, el algoritmo considera que 195 es el tamaño adecuado del árbol.

No mostraremos el summary de las reglas de asociación generadas por el modelo, debido a la gran cantidad de información que representa.

```
# resumen del modelo
summary.tree0 <- summary(treemod.noboost)
```

Seguidamente, evaluaremos el desempeño del modelo mediante la data de prueba. El resultado lo observaremos en la subsección de comparación..

```
# predicciones
tree0.predict <- predict(treemod.noboost, xtest)
# matriz de confusiones
tree0.cf <- confusionMatrix(tree0.predict, ytest)
# resumen de la matriz de confusiones
sumtree0 <- summaryconfmat(tree0.cf, niveles, "TreeNoBoost")
```

2.4.4.2 Con activacion de boosting

El procedimiento es similar al anterior, con la diferencia de que añadiremos un argumento `trials`. Usaremos como valor, al 10, el cual es el valor de convención.

De acuerdo con los resultados del modelo, de los 320 predictores, el algoritmo considera que el tamaño promedio del árbol óptimo es 128.7.

```
# fijar semilla
set.seed(params$seed.train)
# modelo sin boosting
treemod.boost<- C5.0(x = xtrain, y = ytrain, trials = 10)
treemod.boost
```

```
Call:
C5.0.default(x = xtrain, y = ytrain, trials = 10)

Classification Tree
Number of samples: 2128
Number of predictors: 320

Number of boosting iterations: 10
Average tree size: 128.7

Non-standard options: attempt to group attributes
```

No mostraremos el summary de las reglas de asociación generadas por el modelo, debido a la gran cantidad de información que representa.

```
# resumen del modelo
summary.tree1 <- summary(treemod.boost)
```

Seguidamente, evaluaremos el desempeño del modelo mediante la data de prueba. El resultado lo observaremos en la subsección de comparación..

```
# predicciones
tree1.predict <- predict(treemod.boost, xtest)
# matriz de confusiones
tree1.cf <- confusionMatrix(tree1.predict, ytest)
# resumen de la matriz de confusiones
sumtree1 <- summaryconfmat(tree1.cf, niveles, "TreeBoost")
```

2.4.4.3 Evaluacion y comparacion

Observemos los resultados de desempeño y comparemos los modelos.

```
# matriz de desempe$\~{n}$o de los modelos
(tree.seq <- cbind(sumtree0, sumtree1))
```

```
         TreeNoBoost TreeBoost
Accuracy       0.660     0.793
Kappa          0.436     0.649
Error          0.340     0.207
Sens.EI        0.502     0.613
Sens.IE        0.635     0.746
Sens.N         0.739     0.891
Spec.EI        0.857     0.951
Spec.IE        0.899     0.927
Spec.N         0.673     0.756
Prec.EI        0.510     0.788
Prec.IE        0.661     0.761
Prec.N         0.721     0.807
```

El árbol con boosting permitió una mejoría al modelo, obteniéndose una precisión de 79.3% y el valor de kappa es 0.649. Además, los indicadores de predicción de cada una de las clases también mejoró y sitúan sus valores entre 0.7 y 0.9. No obstante este modelo no supera el desempeño de algoritmos anteriores.

2.4.5 Random Forest

En esta sección evaluaremos la potencia de los árboles aleatorios o *random forest* en la clasificación de límites de splicing. Se explorará la opción de número de árboles $n = 50, 100$. Para esto, emplearemos la función `randomForest()`.

```
# fijar semilla
set.seed(params$seed.train)
# modelo de 50 arboles
rf50<- randomForest(class~., data = df.train, ntree = 50)
# modelo de 100 arboles
rf100<- randomForest(class~., data = df.train, ntree = 100)
```

Evaluamos el desempeño de los modelos en los datos de prueba.

```
## modelo de 50 arboles
# prediccion
predict.rf50<- predict(rf50, df.test)
# confusion matrix
rf50.confmat <- confusionMatrix(predict.rf50, df.test$class)

## modelo de 100 arboles
# prediccion
predict.rf100<- predict(rf100, df.test)
# confusion matrix
rf100.confmat <- confusionMatrix(predict.rf100, df.test$class)
```

Comparemos los resultados de desempeño de ambos modelos.

```r
## modelo de 50 arboles
sumcf.rf50 <- summaryconfmat(rf50.confmat, niveles, "RF.n=50")

## modelo de 100 arboles
sumcf.rf100 <- summaryconfmat(rf100.confmat, niveles, "RF.n=100")

(rf.seq <- cbind(sumcf.rf50, sumcf.rf100))
```

```
         RF.n=50 RF.n=100
Accuracy   0.790    0.807
Kappa      0.627    0.653
Error      0.210    0.193
Sens.EI    0.506    0.494
Sens.IE    0.694    0.702
Sens.N     0.954    0.988
Spec.EI    0.980    0.995
Spec.IE    0.965    0.977
Spec.N     0.638    0.632
Prec.EI    0.885    0.968
Prec.IE    0.862    0.903
Prec.N     0.751    0.755
```

Podemos observar que el algoritmo de random forest presenta mejores resultados con número de árboles $n = 100$. La precisión global de este modelo es 0.807 y el valor de kappa es 0.653. No obstante, observamos que la tasa de verdaderos positivos de la predicción de "EI" es deficiente.

3 Conclusiones y discusion

El análisis realizado permite observar la potencia de diferentes algoritmos populares en *machine learning*, sobre la predicción de *splice junctions*, en donde la cantidad de características es alta. De los modelos estudiados los que presentaron mayor desempeño fueron el algoritmo de soporte de máquina vectorial con una arquitectura no lineal `rbf` y el algoritmo de Naive Bayes. Esto sugiere que las clases de límites presentan una posible separación no linal, pese a que no se observó la distribución multivariante de las variables predictoras. De entre los modelos con menor desempeño estuvieron el algoritmo kNN y los árboles de decisión. No obstante y en términos generales, los métodos han presentado un desempeño desde aceptable hasta excelente.

Adicionalmente, el grupo de datos con el que se hizo el análisis, proviene de una reducción de dimensiones, realizada con un *autoencoder*, por lo que pasamos de 480 características a 320. Así también, el tipo de datos pasó de ser binario a real. Los algoritmos realizaon predicciones con suficiente precisión, pese a la reducción.

4 References

Allaire, JJ. 2018. *Deep Learning with r*. Simon; Schuster.

Bagén, Lozano, Bosch Rué Toni, Casas Roma Anna, and others. 2019. "Deep Learning: Principios y Fundamentos." *Deep Learning*, 1–260.

DataTechNotes. 2020. "Convolutional Autoencoder Example with Keras in r." https://www.datatechnotes.com/2020/03/convolutional-autoencoder-example-with-keras-in-r.html.

Hodnett, Mark, Joshua F Wiley, Yuxi (Hayden) Liu, and Pablo Maldonado. 2019. *Deep Learning with r for Beginners: Design Neural Network Models in r 3. 5 Using TensorFlow, Keras, and MXNet*. Packt Publishing, Limited.

Lantz, Brett. 2015. *Machine Learning with r Discover How to Build Machine Learning Algorithms, Prepare Data, and Dig Deep into Data Prediction Techniques with r.* Packt publishing ltd.

StatsLab. 2018. "Let's Play with Autoencoders (Keras, r)." https://statslab.eighty20.co.za/posts/autoencoders_keras_r/.

5 Apendice

Función para resumir indicadores de las matrices de confusión

```
summaryconfmat <- function(confmatrix, niveles, nombre) {
  # confmatrix = objeto confusionMatrix de caret
  # niveles = niveles del factor de respuesta
  # nombre = nombre alternativo del modelo a evaluar

  # posibles respuestas
  nnivel <- length(niveles)

  # creacion de vectores vacio
  Sens <- c()
  Spec <- c()
  Prec <- c()
  Clase <- c()
  etiquetas.ind <- c()
  matriz.metricas <- c()

  # tabla de valores de matriz de confusion
  cmatriz <- as.matrix(confmatrix$table)
  # tabla de resultados por clase
  resbyclass <- confmatrix$byClass
  # resultados globales
  resglobal <- confmatrix$overall

  # Global accuracy
  acc <- round(as.numeric(resglobal["Accuracy"]),3)
  names(acc) <- "Accuracy"
  # Global kappa
  kap <- round(as.numeric(resglobal["Kappa"]),3)
  names(kap) <- "Kappa"
  # Global Error rate
  er <- 1-acc
  names(er) <- "Error"

  for (j in 1:nnivel) {
  # M$\'{e}$tricas para la clase
  clase <- paste("Class:", niveles[j])

  # sensitivity
  Sens[j] <- round(as.numeric(resbyclass[clase, "Sensitivity"]),3)
  # specificity
  Spec[j] <- round(as.numeric(resbyclass[clase, "Specificity"]),3)
  # precision
  Prec[j] <- round(as.numeric(resbyclass[clase, "Pos Pred Value"]),3)
  # Clase
```

```
  Clase[j] <- paste("Class:", niveles[j])
  }
  # metricas individuales
  metricas <- c(rep("Sens",times=3),
                rep("Spec",times=3),
                rep("Prec",times=3))
  etiquetas.ind <- rep(niveles, times=3)
  etiquetas.ind2 <- paste(metricas, etiquetas.ind, sep = ".")
  # vector de metricas finales
  k.ind <- c(Sens,Spec,Prec)
  names(k.ind) <- etiquetas.ind2
  k.ind <- c(acc, kap, er, k.ind)

  matriz.metricas <- cbind(matriz.metricas,k.ind)
  colnames(matriz.metricas) <- nombre
  return(matriz.metricas)
}
```

Función para ejecutar varios valores de k en algoritmo kNN

```
knnperform <- function(seqtrain, seqtest, clastrain, clastest, k, niveles) {
  # seqtrain = data.frame de variables predictoras en muestra training
  # seqtest = data.frame de variables predictoras en muestra test
  # clastrain = vector de clase a predecir en muestra de entrenamiento
  # clastest = vector de clase a predecir en muestra de prueba
  # k = vector de valores de k a evaluar en algoritmo knn
  # positivo = categoria de interes
  # niveles = niveles del factor de respuesta

  # veces que repetiremos la funcion knnperform
  veces <- length(k)

  # posibles respuestas
  nnivel <- length(niveles)

  # creacion de vectores vacio
  Sens <- c()
  Spec <- c()
  Prec <- c()
  Clase <- c()
  etiquetas.ind <- c()
  matriz.metricas <- c()

  for (i in 1:veces) {

  # Prediccion del grupo en test para cada valor en k
  pred <- knn(train = seqtrain, test = seqtest, cl = clastrain, k = k[i])

  # Para determinar los indicadores de desempeno haremos uso de los
  # resultados que ofrece la funcion confusionMatrix del paquete caret

  # creacion de matriz de confusion y todos sus valores
  confmatrix <- caret::confusionMatrix(pred, clastest)
```

```
  # tabla de valores de matriz de confusion
  cmatriz <- as.matrix(confmatrix$table)
  # tabla de resultados por clase
  resbyclass <- confmatrix$byClass
  # resultados globales
  resglobal <- confmatrix$overall

  # Global accuracy
  acc <- round(as.numeric(resglobal["Accuracy"]),3)
  names(acc) <- "Accuracy"
  # Global kappa
  kap <- round(as.numeric(resglobal["Kappa"]),3)
  names(kap) <- "Kappa"
  # Global Error rate
  er <- 1-acc
  names(er) <- "Error"

  for (j in 1:nnivel) {
  # M$\'{e}$tricas para la clase
  clase <- paste("Class:", niveles[j])

  # sensitivity
  Sens[j] <- round(as.numeric(resbyclass[clase, "Sensitivity"]),3)
  # specificity
  Spec[j] <- round(as.numeric(resbyclass[clase, "Specificity"]),3)
  # precision
  Prec[j] <- round(as.numeric(resbyclass[clase, "Pos Pred Value"]),3)
  # Clase
  Clase[j] <- paste("Class:", niveles[j])
  }
  # metricas individuales
  metricas <- c(rep("Sens",times=3),
                rep("Spec",times=3),
                rep("Prec",times=3))
  etiquetas.ind <- rep(niveles, times=3)
  etiquetas.ind2 <- paste(metricas, etiquetas.ind, sep = ".")
  # vector de metricas finales
  k.ind <- c(Sens,Spec,Prec)
  names(k.ind) <- etiquetas.ind2
  k.ind <- c(acc, kap, er, k.ind)

  matriz.metricas <- cbind(matriz.metricas,k.ind)
  }
  colnames(matriz.metricas) <- paste("k", k, sep="=")

  return(matriz.metricas)
}
```