Filtrado de spam de teléfonos móviles con el algoritmo naïve Bayes

A medida que el uso de los teléfonos móviles ha crecido en todo el mundo, se ha abierto una nueva vía para el correo basura electrónico para los vendedores de dudosa reputación. Estos anunciantes utilizan mensajes de texto SMS para dirigirse a consumidores potenciales con publicidad no deseada, conocida como **spam SMS**. Este tipo de spam es problemático porque, a diferencia del spam de correo electrónico, un mensaje SMS es particularmente perturbador debido a la omnipresencia del teléfono móvil.

El desarrollo de un algoritmo de clasificación que pudiera filtrar el spam SMS proporcionaría una herramienta útil para los proveedores de telefonía celular.

Dado que Naïve Bayes se ha utilizado con éxito para filtrar el spam de correo electrónico, parece probable que también se pueda aplicar al spam SMS. Sin embargo, en relación con el spam de correo electrónico, el spam SMS plantea desafíos adicionales para los filtros automáticos. Los mensajes SMS suelen estar limitados a 160 caracteres, lo que reduce la cantidad de texto que se puede utilizar para identificar si un mensaje es basura (*junk*). El límite, combinado con los pequeños teclados de los teléfonos móviles, ha llevado a muchos a adoptar una forma de jerga abreviada para SMS (*shorthand lingo*), que difumina aún más la línea entre los mensajes legítimos y el spam. Veamos cómo un clasificador Naïve Bayes simple maneja estos desafíos.

Paso 1: recopilación de datos

Para desarrollar el clasificador Naïve Bayes, utilizaremos datos adaptados de la Colección de Spam SMS en el Repositorio de Aprendizaje Automático de UCI.

Para leer más sobre cómo se desarrolló la Colección de Spam SMS, consulta el artículo: On the validity of a new SMS spam collection, Gómez, J. M., Almeida, T. A., y Yamakami, A., Proceedings of the 11th IEEE International Conference on Machine Learning and Applications, 2012.

Este conjunto de datos incluye el texto de los mensajes SMS, junto con una etiqueta que indica si el mensaje es no deseado. Los mensajes basura se etiquetan como spam, mientras que los mensajes legítimos se etiquetan como ‘ham’. Algunos ejemplos de spam y ham se muestran en la siguiente tabla:

Ejemplo de ham SMS	Ejemplo de SMS spam
• Mejor. Me recuperé del viernes y ayer me atiborré como un cerdo. Ahora me siento mal. Pero al menos no es un tipo de mal que me retuerce de dolor. • Si empiezas a buscar trabajo, en pocos días lo conseguirás. Tienes un gran potencial y talento. • Conseguí otro trabajo! El del hospital, donde hago análisis de datos o algo así, empieza el lunes! No estoy seguro de cuándo terminaré mi tesis.	• Felicitaciones, has ganado 500 vales en CD o 125 regalos garantizados y entrada gratis al sorteo semanal de 100. Envía un mensaje de texto con la palabra MUSIC al 87066. • Solo en diciembre! ¿Has tenido tu móvil durante 11 meses o más? Tienes derecho a actualizarlo a la última cámara a color para móvil de forma gratuita! Llama a The Mobile Update Co GRATIS al 08002986906 • Especial de San Valentín! Gana más de £1000 en nuestro concurso y lleva a tu pareja al viaje de tu vida! Envía GO al 83600 ahora. 150p/mensaje recibido.

Al observar los mensajes anteriores, ¿observas alguna característica distintiva del spam? Una característica notable es que dos de los tres mensajes de spam utilizan la palabra gratis, aunque esta palabra no aparece en ninguno de los mensajes de ham. Por otro lado, dos de los mensajes de ham citan días específicos de la semana, en comparación con cero en los mensajes de spam.

Nuestro clasificador Naïve Bayes aprovechará estos patrones en la frecuencia de palabras para determinar si los mensajes SMS parecen ajustarse mejor al perfil de spam o ham. Si bien no es inconcebible que la palabra gratis aparezca fuera de un SMS de spam, es probable que un mensaje legítimo proporcione palabras adicionales que den contexto.

Por ejemplo, un mensaje de ham podría preguntar: "¿Estás libre el domingo?", mientras que un mensaje de spam podría utilizar la frase "tonos de llamada gratis". El clasificador calculará la probabilidad de spam y ham dada la evidencia proporcionada por todas las palabras del mensaje.

Paso 2: exploración y preparación de los datos

El primer paso para construir nuestro clasificador implica procesar los datos sin procesar para su análisis. Los datos de texto son difíciles de preparar porque es necesario transformar las palabras y oraciones en un formato que una computadora pueda entender. Transformaremos nuestros datos SMS en una representación conocida como bolsa de palabras (**bag-of-words**), que proporciona una característica binaria que indica si cada palabra aparece en el ejemplo dado, ignorando el orden de las palabras o el contexto en el que aparece la palabra. Aunque se trata de una representación relativamente simple, como demostraremos pronto, funciona bastante bien para muchas tareas de clasificación.

El conjunto de datos utilizado aquí se ha modificado ligeramente con respecto al original para que sea más fácil trabajar con él en R. Si planeas seguir el ejemplo, descarga el archivo adjunto sms_spam.csv y guárdalo en tu directorio de trabajo de R.

Comenzaremos importando los datos CSV y guardándolos en un frame de datos:

```
> sms_raw <- read.csv("sms_spam.csv")
```

Usando la función str(), vemos que el frame de datos sms_raw incluye 5,559 mensajes SMS en total con dos características: tipo (*type*) y texto (*text*). El tipo de SMS se ha codificado como ham o spam. El elemento de texto almacena el texto completo del mensaje SMS sin procesar:

```
> str(sms_raw)

'data.frame':   5559 obs. of  2 variables:
 $ type: chr  "ham" "ham" "ham" "spam" ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my thanks." "Am
also doing in cbe only. But have to pay." "complimentary 4 STAR Ibiza Holiday or £10,000 cash
needs your URGENT collection. 09066364349 NOW from Landline "
| __truncated__ ...
```

El elemento de tipo es actualmente un vector de caracteres. Dado que se trata de una variable categórica, sería mejor convertirla en un factor, como se muestra en el siguiente código:

```
> sms_raw$type <- factor(sms_raw$type)
```

Al examinar esto con las funciones str() y table(), vemos que el tipo ahora se ha recodificado correctamente como un factor. Además, vemos que 747 (aproximadamente el 13 por ciento) de los mensajes SMS en nuestros datos se etiquetaron como spam, mientras que los demás se etiquetaron como ham:

```
> str(sms_raw$type)
 Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
> table(sms_raw$type)
 ham spam
4812  747
```

Por ahora, dejaremos el texto del mensaje en paz. Como aprenderás en la siguiente sección, el procesamiento de mensajes SMS sin procesar requerirá el uso de un nuevo conjunto de herramientas poderosas diseñadas específicamente para procesar datos de texto.

Preparación de datos - limpieza y estandarización de datos de texto

Los mensajes SMS son cadenas de texto compuestas de palabras, espacios, números y puntuación. El manejo de este tipo de datos complejos requiere una gran cantidad de reflexión y esfuerzo. Es necesario considerar cómo eliminar números y puntuación; manejar palabras poco interesantes, como and, but, and or; y dividir oraciones en palabras individuales. Afortunadamente, esta funcionalidad ha sido proporcionada por miembros de la comunidad R en un paquete de minería de texto llamado tm.

El paquete tm se puede instalar mediante el comando install.packages("tm") y cargarse con el comando library(tm). Incluso si ya lo tienes instalado, puede que valga la pena volver a realizar la instalación para asegurarte de que tu versión esté actualizada, ya que el paquete tm aún se encuentra en desarrollo activo.
Esto ocasionalmente da como resultado cambios en su funcionalidad.

```
> install.packages("tm")
> library(tm)
```

El primer paso para procesar datos de texto implica la creación de un **corpus**, que es una colección de documentos de texto. Los documentos pueden ser cortos o largos, desde artículos de noticias individuales, páginas de un libro, páginas de la web o incluso libros completos. En nuestro caso, el corpus será una colección de mensajes SMS.

Para crear un corpus, utilizaremos la función VCorpus() del paquete tm, que hace referencia a un corpus volátil (el término “volátil” significa que se almacena en la memoria en lugar de almacenarse en el disco; la función PCorpus() se utiliza para acceder a un corpus permanente almacenado en una base de datos). Esta función requiere que especifiquemos la fuente de los documentos para el corpus, que podría ser el sistema de archivos de una computadora, una base de datos, la web o cualquier otro lugar.

Dado que ya cargamos el texto del mensaje SMS en R, utilizaremos la función de lectura VectorSource() para crear un objeto de origen a partir del vector sms_raw$text existente, que luego se puede suministrar a VCorpus() de la siguiente manera:

```
> sms_corpus <- VCorpus(VectorSource(sms_raw$text))
```

El objeto de corpus resultante se guarda con el nombre sms_corpus.

Al especificar un parámetro readerControl opcional, se puede utilizar la función VCorpus() para importar texto de fuentes como archivos PDF y Microsoft Word. Para obtener más

información, examine la sección Importación de datos en el paquete tm vignette utilizando el comando vignette("tm").

Al imprimir el corpus, vemos que contiene documentos para cada uno de los 5,559 mensajes SMS en los datos de entrenamiento:

```
> print(sms_corpus)
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 5559
```

Ahora, debido a que el corpus tm es esencialmente una lista compleja, podemos utilizar operaciones de lista para seleccionar documentos en el corpus. La función inspect() muestra un resumen del resultado. Por ejemplo, el siguiente comando mostrará un resumen del primer y segundo mensaje SMS en el corpus:

```
> inspect(sms_corpus[1:2])
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 2

[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 49

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 23
```

Para ver el texto del mensaje real, se debe aplicar la función as.character() a los mensajes deseados. Para ver un mensaje, utiliza la función as.character() en un solo elemento de la lista, teniendo en cuenta que se requiere la notación de corchetes dobles:

```
> as.character(sms_corpus[[1]])
[1] "Hope you are having a good week. Just checking in"
```

Para ver varios documentos, necesitaremos aplicar as.character() a varios elementos en el objeto sms_corpus. Para esto, utilizaremos la función lapply(), que forma parte de una familia de funciones R que aplica un procedimiento a cada elemento de una estructura de datos R. Estas funciones, que incluyen apply() y sapply() entre otras, son uno de los modismos clave del lenguaje R. Los programadores R experimentados las utilizan de forma muy similar a la forma en que se utilizan los ciclos for o while en otros lenguajes de programación, ya que

dan como resultado un código más legible (y, a veces, más eficiente). La función lapply() para aplicar as.character() a un subconjunto de elementos del corpus es la siguiente:

```
> lapply(sms_corpus[1:2], as.character)
$`1`
[1] "Hope you are having a good week. Just checking in"

$`2`
[1] "K..give back my thanks."
```

Como se señaló anteriormente, el corpus contiene el texto sin procesar de 5,559 mensajes de texto. Para realizar nuestro análisis, necesitamos dividir estos mensajes en palabras individuales. Primero, necesitamos limpiar el texto para estandarizar las palabras y eliminar los caracteres de puntuación que ensucian el resultado. Por ejemplo, nos gustaría que las cadenas Hello!, HELLO y hello se cuenten como instancias de la misma palabra.

La función tm_map() proporciona un método para aplicar una transformación (también conocida como mapeo) a un corpus tm. Usaremos esta función para limpiar nuestro corpus usando una serie de transformaciones y guardaremos el resultado en un nuevo objeto llamado corpus_clean.

Nuestra primera transformación estandarizará los mensajes para que utilicen solo caracteres en minúscula. Para ello, R proporciona una función tolower() que devuelve una versión en minúscula de las cadenas de texto. Para aplicar esta función al corpus, necesitamos utilizar la función contenedora de tm content_transformer() para tratar a tolower() como una función de transformación que se puede utilizar para acceder al corpus. El comando completo es el siguiente:

```
> sms_corpus_clean <- tm_map(sms_corpus,
+  content_transformer(tolower))
```

Para comprobar si el comando funcionó como se esperaba, inspeccionemos el primer mensaje del corpus original y comparémoslo con el mismo en el corpus transformado:

```
> as.character(sms_corpus[[1]])
[1] "Hope you are having a good week. Just checking in"
> as.character(sms_corpus_clean[[1]])
[1] "hope you are having a good week. just checking in"
```

Como se esperaba, las letras mayúsculas del corpus limpio se han reemplazado por versiones en minúscula.

La función content_transformer() se puede utilizar para aplicar procesos de limpieza y procesamiento de texto más sofisticados, como la búsqueda y el reemplazo de patrones de grep.
Simplemente escribe una función personalizada y envuélvela (*wrap*) antes de aplicar la función tm_map().

Continuemos con nuestra limpieza eliminando números de los mensajes SMS. Aunque algunos números pueden proporcionar información útil, es probable que la mayoría sean exclusivos de los remitentes individuales y, por lo tanto, no proporcionarán patrones útiles en todos los mensajes. Con esto en mente, eliminaremos todos los números del corpus de la siguiente manera:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

Ten en cuenta que el código anterior no utilizó la función content_transformer(). Esto se debe a que removeNumbers() está incluido con tm junto con varias otras funciones de mapeo que no necesitan ser encapsuladas. Para ver las otras transformaciones integradas, simplemente escriba getTransformations().

Nuestra próxima tarea es eliminar palabras de relleno como to, and, but, and or de los mensajes SMS. Estos términos se conocen como palabras vacías y generalmente se eliminan antes de la minería de texto. Esto se debe al hecho de que, aunque aparecen con mucha frecuencia, no brindan mucha información útil para nuestro modelo, ya que es poco probable que distingan entre spam y ham.

En lugar de definir una lista de palabras vacías (*stop words*) nosotros mismos, utilizaremos la función stopwords() proporcionada por el paquete tm. Esta función nos permite acceder a conjuntos de palabras vacías de varios idiomas. De forma predeterminada, se utilizan palabras vacías comunes en inglés. Para ver la lista predeterminada, escribe stopwords() en el símbolo del sistema de R. Para ver los demás idiomas y opciones disponibles, escribe ?stopwords para acceder a la página de documentación.

Aqui me quedé

Incluso dentro de un mismo idioma, no existe una única lista definitiva de palabras vacías. Por ejemplo, la lista predeterminada en inglés de tm incluye alrededor de 174 palabras, mientras que otra opción incluye 571 palabras. Incluso puedes especificar tu propia lista de palabras vacías. Independientemente de la lista que elijas, ten en cuenta el objetivo de esta transformación, que es eliminar datos inútiles y mantener la mayor cantidad posible de información útil.

Definir las palabras vacías por sí solas no es una transformación. Lo que necesitamos es una forma de eliminar cualquier palabra que aparezca en la lista de palabras vacías. La solución

está en la función removeWords(), que es una transformación incluida en el paquete tm. Como hemos hecho antes, utilizaremos la función tm_map() para aplicar esta asignación a los datos, proporcionando la función stopwords() como parámetro para indicar las palabras que nos gustaría eliminar. El comando completo es el siguiente:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean,
+   removeWords, stopwords())
```

Dado que stopwords() simplemente devuelve un vector de palabras vacías, si así lo hubiéramos elegido, podríamos haber reemplazado esta llamada de función con nuestro propio vector de palabras para eliminar. De esta manera, podríamos expandir o reducir la lista de palabras vacías a nuestro gusto o eliminar un conjunto diferente de palabras por completo.

Continuando con nuestro proceso de limpieza, también podemos eliminar cualquier puntuación de los mensajes de texto utilizando la transformación incorporada removePunctuation():

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

La transformación removePunctuation() elimina por completo los caracteres de puntuación del texto, lo que puede generar consecuencias no deseadas. Por ejemplo, considera lo que sucede cuando se aplica de la siguiente manera:

```
> removePunctuation("hello...world")
[1] "helloworld"
```

Como se muestra, la falta de un espacio en blanco después de los puntos suspensivos provocó que las palabras hola y mundo se unieran como una sola palabra. Si bien esto no es un problema sustancial en este momento, vale la pena señalarlo para el futuro.

Para evitar el comportamiento predeterminado de removePunctuation(), es posible crear una función personalizada que reemplace en lugar de eliminar los caracteres de puntuación:

```
> replacePunctuation <- function(x) {
+   gsub("[[:punct:]]+", " ", x)
+ }
```

Esto utiliza la función gsub() de R para sustituir cualquier carácter de puntuación en x con un espacio en blanco. Esta función replacePunctuation() se puede utilizar con tm_map() como con otras transformaciones. La sintaxis extraña del comando gsub() aquí se debe al uso de una expresión regular, que especifica un patrón que coincide con los caracteres de texto.

Otra estandarización común para los datos de texto implica reducir las palabras a su forma raíz en un proceso llamado 'derivación' (**stemming**). El proceso de derivación toma palabras como learned, learning y learns y elimina el sufijo para transformarlas en la forma base, learn. Esto permite que los algoritmos de aprendizaje automático traten los términos relacionados como un concepto único en lugar de intentar aprender un patrón para cada variante.

El paquete tm proporciona la funcionalidad de derivación a través de la integración con el paquete SnowballC.
Al momento de escribir este documento, SnowballC no está instalado de manera predeterminada con tm, por lo que debes hacerlo con install.packages("SnowballC") si aún no lo has hecho.

```
> install.packages("SnowballC")
> library(SnowballC)
```

El paquete SnowballC es mantenido por Milan Bouchet-Valat y proporciona una interfaz R para la biblioteca libstemmer basada en C, basada en el algoritmo de derivación de palabras "Snowball" de M. F. Porter, un método de derivación de código abierto ampliamente utilizado. Para obtener más detalles, consulta http://snowballstem.org.

El paquete SnowballC proporciona una función wordStem(), que para un vector de caracteres devuelve el mismo vector de términos en su forma raíz. Por ejemplo, la función deriva correctamente las variantes de la palabra learn como se describió anteriormente:

```
> wordStem(c("learn", "learned", "learning", "learns"))
[1] "learn" "learn" "learn" "learn"
```

Para aplicar la función wordStem() a un corpus completo de documentos de texto, el paquete tm incluye una transformación stemDocument(). Aplicamos esto a nuestro corpus con la función tm_map() exactamente como antes:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument
```

Si recibes un mensaje de error al aplicar la transformación stemDocument(), confirma que tienes instalado el paquete SnowballC.

Después de eliminar números, palabras vacías y puntuación, y luego realizar también la lematización, los mensajes de texto quedan con los espacios en blanco que alguna vez separaron las partes que ahora faltan. Por lo tanto, el paso final en nuestro proceso de limpieza de texto es eliminar espacios en blanco adicionales utilizando la transformación stripWhitespace() incorporada:

```
> sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

La siguiente tabla muestra los primeros tres mensajes en el corpus SMS antes y después del proceso de limpieza. Los mensajes se han limitado a las palabras más interesantes, y se han eliminado la puntuación y las mayúsculas:

Mensajes SMS antes de la limpieza	Mensajes SMS después de la limpieza
> as.character(sms_corpus[1:3]) [[1]] Hope you are having a good week. Just checking in [[2]] K..give back my thanks. [[3]] Am also doing in cbe only. But have to pay.	> as.character(sms_corpus_clean[1:3]) [[1]] hope good week just check [[2]] kgive back thank [[3]] also cbe pay

Preparación de datos - división de documentos de texto en palabras

Ahora que los datos se procesaron a nuestro gusto, el paso final es dividir los mensajes en términos individuales a través de un proceso llamado t**okenización**. Un token es un elemento único de una cadena de texto; en este caso, los tokens son palabras.

Como puedes suponer, el paquete tm proporciona una funcionalidad para convertir en tokens el corpus de mensajes SMS.
La función DocumentTermMatrix() toma un corpus y crea una estructura de datos denominada matriz de documentos y términos (DTM, **document-term matrix**) en la que las filas indican documentos (mensajes SMS) y las columnas indican términos (palabras).

El paquete tm también proporciona una estructura de datos para una matriz de términos y documentos (TDM), que es simplemente un DTM transpuesto en el que las filas son términos y las columnas son documentos. ¿Por qué se necesitan ambas? A veces, es más conveniente trabajar con una u otra.

Por ejemplo, si la cantidad de documentos es pequeña, mientras que la lista de palabras es grande, puede tener sentido usar una TDM porque suele ser más fácil mostrar muchas filas que muchas columnas. Dicho esto, los algoritmos de aprendizaje automático generalmente requerirán una DTM, ya que las columnas son las características y las filas son los ejemplos.

Cada celda de la matriz almacena un número que indica un recuento de las veces que la palabra representada por la columna aparece en el documento representado por la fila. La siguiente figura muestra solo una pequeña parte del DTM para el corpus de SMS, ya que la matriz completa tiene 5,559 filas y más de 7000 columnas:

message #	balloon	balls	bam	bambling	band
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0

Figura 1: El DTM para los mensajes SMS está lleno principalmente de ceros.

El hecho de que cada celda de la tabla sea cero implica que ninguna de las palabras que aparecen en la parte superior de las columnas aparece en ninguno de los primeros cinco mensajes del corpus. Esto resalta la razón por la que esta estructura de datos se denomina matriz dispersa; la gran mayoría de las celdas de la matriz están llenas de ceros. Expresado en términos del mundo real, aunque cada mensaje debe contener al menos una palabra, la probabilidad de que aparezca una palabra en un mensaje determinado es pequeña.

La creación de una matriz dispersa DTM a partir de un corpus tm implica un solo comando:

```
> sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
```

Esto creará un objeto sms_dtm que contiene el corpus tokenizado utilizando la configuración predeterminada, que aplica un procesamiento adicional mínimo. La configuración predeterminada es adecuada porque ya hemos preparado el corpus manualmente.

Por otro lado, si aún no hubiéramos realizado el preprocesamiento, podríamos hacerlo aquí proporcionando una lista de opciones de parámetros de control para anular los valores predeterminados. Por ejemplo, para crear un DTM directamente a partir del corpus SMS sin procesar, podemos utilizar el siguiente comando:

```
> sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
+   tolower = TRUE,
+   removeNumbers = TRUE,
+   stopwords = TRUE,
+   removePunctuation = TRUE,
+   stemming = TRUE
+ ))
```

Esto aplica los mismos pasos de preprocesamiento al corpus SMS en el mismo orden que se hizo anteriormente.

Sin embargo, al comparar sms_dtm con sms_dtm2, vemos una ligera diferencia en la cantidad de términos en la matriz:

```
> sms_dtm
<<DocumentTermMatrix (documents: 5559, terms: 6542)>>
Non-/sparse entries: 42113/36324865
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)

> sms_dtm2
<<DocumentTermMatrix (documents: 5559, terms: 6940)>>
Non-/sparse entries: 43186/38536274
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
```

La razón de esta discrepancia tiene que ver con una pequeña diferencia en el orden de los pasos de preprocesamiento. La función DocumentTermMatrix() aplica sus funciones de limpieza a las cadenas de texto solo después de que se han dividido en palabras.

Por lo tanto, utiliza una función de eliminación de palabras vacías ligeramente diferente. En consecuencia, algunas palabras se dividen de manera diferente que cuando se limpian antes de la tokenización.

Para forzar que los dos DTM anteriores sean idénticos, podemos anular la función de palabras vacías predeterminada con la nuestra, que utiliza la función de reemplazo original. Simplemente reemplaza stopwords = TRUE con lo siguiente:

```
stopwords = function(x) { removeWords(x, stopwords()) }
```

Las diferencias entre estos dos elementos hacen surgir un principio importante de la limpieza de datos de texto: el orden de las operaciones es importante. Teniendo esto en cuenta, es muy importante pensar en cómo los primeros pasos del proceso afectarán a los posteriores.

El orden presentado aquí funcionará en muchos casos, pero cuando el proceso se adapta con más cuidado a conjuntos de datos y casos de uso específicos, puede ser necesario replanteárselo.

Por ejemplo, si hay determinados términos que deseas excluir de la matriz, considera si buscarlos antes o después de la lematización. Además, considera cómo la eliminación de la puntuación (y si la puntuación se elimina o se reemplaza por un espacio en blanco) afecta a estos pasos.

Preparación de datos - creación de conjuntos de datos de prueba y de entrenamiento

Con nuestros datos preparados para el análisis, ahora necesitamos dividirlos en conjuntos de datos de prueba y de entrenamiento para que, después de que se construya nuestro clasificador de spam, pueda evaluarse con datos que no haya visto anteriormente.

Sin embargo, aunque necesitamos mantener al clasificador 'ciego' en cuanto al contenido del conjunto de datos de prueba, es importante que la división se produzca después de que los datos se hayan limpiado y procesado. Necesitamos que se hayan realizado exactamente los mismos pasos de preparación en los conjuntos de datos de entrenamiento y de prueba.

Dividiremos los datos en dos partes: 75 por ciento para entrenamiento y 25 por ciento para prueba. Dado que los mensajes SMS se ordenan de forma aleatoria, podemos simplemente tomar los primeros 4169 para entrenamiento y dejar los 1390 restantes para prueba. Afortunadamente, el objeto DTM actúa de forma muy similar a un frame de datos y se puede dividir utilizando las operaciones estándar [fila, columna]. Como nuestro DTM almacena los mensajes SMS como filas y las palabras como columnas, debemos solicitar un rango específico de filas y todas las columnas para cada una:

```
> sms_dtm_train <- sms_dtm[1:4169, ]
> sms_dtm_test <- sms_dtm[4170:5559, ]
```

Para mayor comodidad más adelante, también es útil guardar un par de vectores con las etiquetas para cada una de las filas en las matrices de entrenamiento y prueba. Estas etiquetas no se almacenan en el DTM, por lo que debemos extraerlas del frame de datos sms_raw original:

```
> sms_train_labels <- sms_raw[1:4169, ]$type
> sms_test_labels <- sms_raw[4170:5559, ]$type
```

Para confirmar que los subconjuntos son representativos del conjunto completo de datos de SMS, comparemos la proporción de spam en los frames de datos de entrenamiento y de prueba:

```
> prop.table(table(sms_train_labels))
sms_train_labels
     ham      spam
0.8647158 0.1352842
> prop.table(table(sms_test_labels))
sms_test_labels
     ham      spam
0.8683453 0.1316547
```

Tanto los datos de entrenamiento como los de prueba contienen aproximadamente un 13 por ciento de spam. Esto sugiere que los mensajes de spam se dividieron de manera uniforme entre los dos conjuntos de datos.

Visualización de datos de texto - nubes de palabras

Una nube de palabras es una forma de representar visualmente la frecuencia con la que aparecen las palabras en los datos de texto. La nube está compuesta de palabras esparcidas de manera algo aleatoria alrededor de la figura. Las palabras que aparecen con más frecuencia en el texto se muestran en una fuente más grande, mientras que los términos menos comunes se muestran en fuentes más pequeñas. Este tipo de figura se hizo popular como una forma de observar temas de tendencia en los sitios web de redes sociales.

El paquete wordcloud proporciona una función R simple para crear este tipo de diagrama. La usaremos para visualizar las palabras en los mensajes SMS. Comparar las nubes de mensajes de spam y de ham nos ayudará a evaluar si es probable que nuestro filtro de spam de Naïve Bayes tenga éxito. Si aún no lo has hecho, instala y carga el paquete escribiendo install.packages("wordcloud") y library(wordcloud) en la línea de comandos de R.

```
> install.packages("wordcloud")
> library(wordcloud)
```

El paquete wordcloud fue escrito por Ian Fellows. Para obtener más información sobre este paquete, visita su blog en http://blog.fellstat.com/?cat=11.

Se puede crear una nube de palabras directamente a partir de un objeto de corpus tm utilizando la sintaxis:

```
> wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
```

Esto creará una nube de palabras a partir de nuestro corpus SMS preparado. Como especificamos random.order = FALSE, la nube se organizará en un orden no aleatorio, con las palabras de mayor frecuencia ubicadas más cerca del centro. Si no especificamos random.order, la nube se organizará aleatoriamente de manera predeterminada.

El parámetro min.freq especifica la cantidad de veces que debe aparecer una palabra en el corpus antes de que se muestre en la nube. Como una frecuencia de 50 es aproximadamente el 1 por ciento del corpus, esto significa que se debe encontrar una palabra en al menos el 1 por ciento de los mensajes SMS para que se incluya en la nube.

Es posible que recibas un mensaje de advertencia que indique que R no pudo incluir todas las palabras en la figura. Si es así, intenta aumentar la frecuencia mínima para reducir la cantidad de palabras en la nube. También puede resultar útil utilizar el parámetro de escala para reducir el tamaño de la fuente.

La nube de palabras resultante debería ser similar a la siguiente:

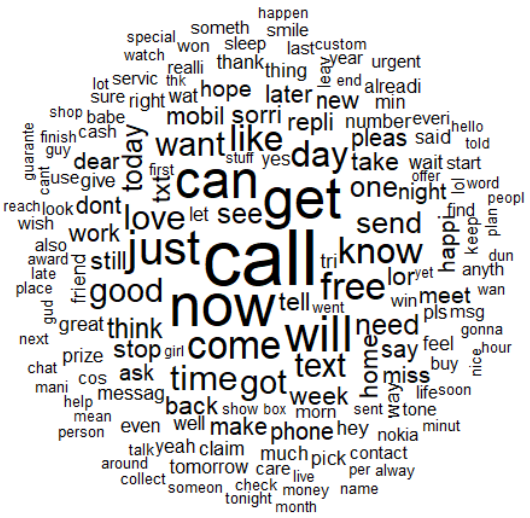


Figura 2: Una nube de palabras que muestra las palabras que aparecen en todos los mensajes SMS.

Una visualización quizás más interesante implica comparar las nubes de spam y ham de SMS. Como no construimos corpus separados para spam y ham, este es un momento apropiado para destacar una característica muy útil de la función wordcloud(). Dado un vector de cadenas de texto sin procesar, aplicará automáticamente procesos de preparación de texto comunes antes de mostrar la nube.

Usemos la función subset() de R para tomar un subconjunto de los datos sms_raw por el tipo de SMS. Primero, crearemos un subconjunto donde el tipo sea spam:

```
> spam <- subset(sms_raw, type == "spam")
```

A continuación, haremos lo mismo para el subconjunto ham:

```
> ham <- subset(sms_raw, type == "ham")
```

Ten cuidado de observar el doble signo igual. Como muchos lenguajes de programación, R usa == para probar la igualdad. Si accidentalmente usas un solo signo igual, terminarás con un subconjunto mucho más grande de lo que esperabas.

Ahora tenemos dos frames de datos, spam y ham, cada uno con una función de texto que contiene las cadenas de texto sin procesar para los mensajes SMS. Crear nubes de palabras es tan simple como antes. Esta vez, usaremos el parámetro max. words para ver las 40 palabras más comunes en cada uno de los 2 conjuntos. El parámetro scale ajusta los tamaños de fuente máximo y mínimo para las palabras en la nube. Siéntete libre de cambiar estos parámetros como creas conveniente. Esto se ilustra en el siguiente código:

```
> wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
> wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```

Ten en cuenta que R proporciona mensajes de advertencia cuando ejecutas este código que indican que la "transformación descarta documentos". Las advertencias están relacionadas con los procedimientos removePunctuation() y removeWords() que wordcloud() realiza de manera predeterminada cuando se le proporcionan datos de texto sin procesar en lugar de una matriz de términos. Básicamente, hay algunos mensajes que se excluyen del resultado porque no queda texto de mensaje después de la limpieza.

Por ejemplo, el mensaje ham con el texto :) que representa el emoji sonriente se elimina del conjunto después de la limpieza. Esto no es un problema para las nubes de palabras y las advertencias se pueden ignorar.

Las nubes de palabras resultantes deberían ser similares a las que aparecen a continuación. ¿Tienes una idea de cuál es la nube de spam y cuál representa ham?

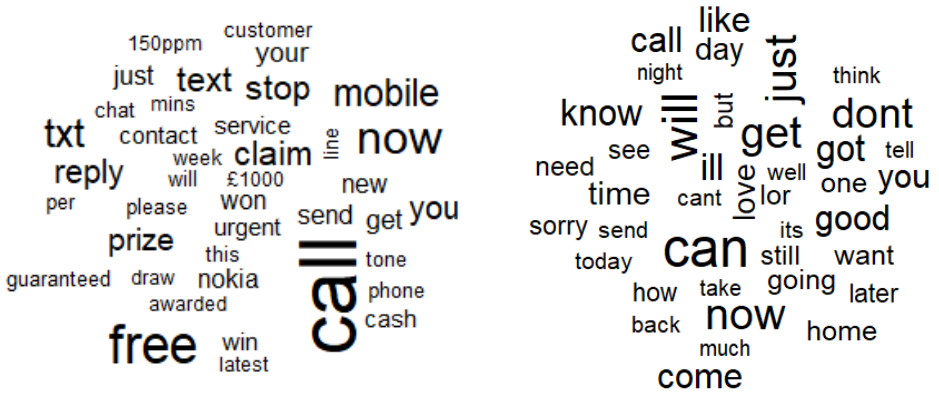


Figura 3: Nubes de palabras en paralelo que representan mensajes SMS spam y mensajes de ham.

Como probablemente hayas adivinado, la nube de spam se encuentra a la izquierda. Los mensajes de spam incluyen palabras como llamar, gratis, móvil, reclamar y parar; estos términos no aparecen en absoluto en la nube de ham. En cambio, los mensajes de ham utilizan palabras como can, sorry, love y time. Estas marcadas diferencias sugieren que nuestro modelo Naïve Bayes tendrá algunas palabras clave sólidas para diferenciar entre las clases.

Preparación de datos - creación de características indicadoras para palabras frecuentes

El paso final en el proceso de preparación de datos es transformar la matriz dispersa en una estructura de datos que se pueda utilizar para entrenar un clasificador Naive Bayes. Actualmente, la matriz dispersa incluye más de 6500 características; se trata de una característica por cada palabra que aparece en al menos un mensaje SMS. Es poco probable que todas ellas sean útiles para la clasificación. Para reducir la cantidad de características, eliminaremos cualquier palabra que aparezca en menos de 5 mensajes, o en menos de aproximadamente el 0.1 por ciento de los registros en los datos de entrenamiento.

Para encontrar palabras frecuentes, se requiere el uso de la función findFreqTerms() en el paquete tm.
Esta función toma un DTM y devuelve un vector de caracteres que contiene palabras que aparecen al menos una cantidad mínima de veces. Por ejemplo, el siguiente comando muestra las palabras que aparecen al menos cinco veces en la matriz sms_dtm_train:

```
> findFreqTerms(sms_dtm_train, 5)
```

El resultado de la función es un vector de caracteres, así que guardemos nuestras palabras frecuentes para más adelante:

```
> sms_freq_words <- findFreqTerms(sms_dtm_train, 5)
```

Un vistazo al contenido del vector nos muestra que hay 1,139 términos que aparecen en al menos 5 mensajes SMS:

```
> str(sms_freq_words)
 chr [1:1137] "£wk" "abiola" "abl" "abt" "accept" "access" "account" "across" "act" "activ" "actual"
"add" ...
```

Ahora debemos filtrar nuestro DTM para incluir solo los términos que aparecen en el vector de palabras frecuentes.
Como antes, utilizaremos operaciones de estilo de frame de datos [row, col] para solicitar secciones específicas del DTM, teniendo en cuenta que los nombres de las columnas del DTM se basan en las palabras que contiene el DTM. Podemos aprovechar este hecho para limitar el DTM a palabras específicas. Dado que queremos todas las filas, pero solo las columnas que representan las palabras en el vector sms_freq_words, nuestros comandos son:

```
> sms_dtm_freq_train <- sms_dtm_train[ , sms_freq_words]
> sms_dtm_freq_test <- sms_dtm_test[ , sms_freq_words]
```

Los conjuntos de datos de entrenamiento y prueba ahora incluyen 1,137 características, que corresponden a palabras que aparecen en al menos 5 mensajes.

El clasificador Naïve Bayes generalmente se entrena con datos con características categóricas. Esto plantea un problema ya que las celdas en la matriz dispersa son numéricas y miden la cantidad de veces que aparece una palabra en un mensaje. Necesitamos cambiar esto a una variable categórica que simplemente indique sí o no, dependiendo de si la palabra aparece o no.

A continuación se define una función convert_counts() para convertir los recuentos en cadenas de Sí o No:

```
> convert_counts <- function(x) {
+   x <- ifelse(x > 0, "Yes", "No")
+ }
```

A estas alturas, algunas de las partes de la función anterior deberían resultar familiares. La primera línea define la función. La declaración ifelse(x > 0, "Yes", "No") transforma los valores en x de tal manera que si el valor es mayor que 0, entonces será reemplazado con "Yes"; de lo contrario, será reemplazado con una cadena "No". Por último, se devuelve el vector x recién transformado.

Ahora necesitamos aplicar convert_counts() a cada una de las columnas en nuestra matriz dispersa. Es posible que puedas adivinar el nombre de la función R que hace exactamente esto. La función se llama simplemente apply() y se usa de manera muy similar a como se usaba lapply() anteriormente.

La función apply() permite utilizar una función en cada una de las filas o columnas de una matriz.
Utiliza un parámetro MARGIN para especificar filas o columnas. Aquí, utilizaremos MARGIN = 2 ya que nos interesan las columnas (MARGIN = 1 se utiliza para las filas). Los comandos para convertir las matrices de entrenamiento y prueba son los siguientes:

```
> sms_train <- apply(sms_dtm_freq_train, MARGIN = 2,
+   convert_counts)
> sms_test <- apply(sms_dtm_freq_test, MARGIN = 2,
+   convert_counts)
```

El resultado serán dos matrices de tipo carácter, cada una con celdas que indican "Yes" o "No" si la palabra representada por la columna aparece en algún punto del mensaje representado por la fila.

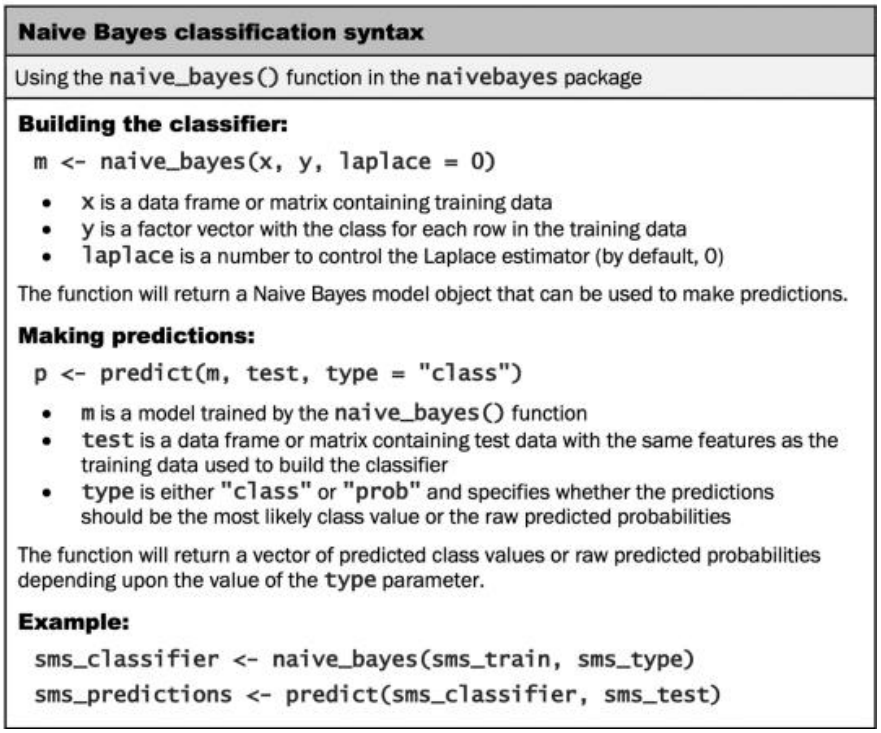
Paso 3 - entrenamiento de un modelo con los datos

Ahora que hemos transformado los mensajes SMS sin procesar en un formato que se puede representar mediante un modelo estadístico, es hora de aplicar el algoritmo Naïve Bayes. El algoritmo utilizará la presencia o ausencia de palabras para estimar la probabilidad de que un mensaje SMS determinado sea spam.

La implementación de Naïve Bayes que emplearemos está en el paquete naivebayes. Este paquete es mantenido por Michal Majka y es una implementación moderna y eficiente de R. Si aún no lo has hecho, asegúrate de instalar y cargar el paquete utilizando los comandos install.packages("naivebayes") y library(naivebayes) antes de continuar.

> Muchos enfoques de aprendizaje automático se implementan en más de un paquete R, y Naïve Bayes no es una excepción. Otra opción es naiveBayes() en el paquete e1071, que se utiliza a menudo en mi curso de minería de datos, pero que, por lo demás, es casi idéntico a naive_bayes() en su uso. El paquete naivebayes utilizado en este documento ofrece un mejor rendimiento y una funcionalidad más avanzada, que se describe en su sitio web: https://majkamichal.github.io/naivebayes/.

A diferencia del algoritmo k-NN que utilizamos para la clasificación en el tema anterior, el entrenamiento de un aprendiz de Naïve Bayes y su uso para la clasificación se producen en etapas separadas. Aun así, como se muestra en la siguiente tabla, estos pasos son bastante sencillos:



Naive Bayes classification syntax

Using the `naive_bayes()` function in the `naivebayes` package

Building the classifier:

```
m <- naive_bayes(x, y, laplace = 0)
```

- `x` is a data frame or matrix containing training data
- `y` is a factor vector with the class for each row in the training data
- `laplace` is a number to control the Laplace estimator (by default, 0)

The function will return a Naive Bayes model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "class")
```

- `m` is a model trained by the `naive_bayes()` function
- `test` is a data frame or matrix containing test data with the same features as the training data used to build the classifier
- `type` is either `"class"` or `"prob"` and specifies whether the predictions should be the most likely class value or the raw predicted probabilities

The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the `type` parameter.

Example:

```
sms_classifier <- naive_bayes(sms_train, sms_type)
sms_predictions <- predict(sms_classifier, sms_test)
```

Figura 4: Sintaxis de clasificación de Naive Bayes.

Utilizando la matriz sms_train, el siguiente comando entrena un objeto clasificador naive_bayes que puede utilizarse para hacer predicciones:

```
> sms_classifier <- naive_bayes(sms_train, sms_train_labels)
```

Después de ejecutar el comando anterior, puede observar el siguiente resultado:

```
There were 50 or more warnings (use warnings() to see the first 50
```

Esto no es nada de lo que alarmarse por ahora; al escribir el comando Warnings() se revela la causa de este problema:

```
> warnings()
Warning messages:
1: naive_bayes(): Feature £wk - zero probabilities are present. Consider Laplace smoothing.
2: naive_bayes(): Feature abiola - zero probabilities are present. Consider Laplace smoothing.
3: naive_bayes(): Feature abl - zero probabilities are present. Consider Laplace smoothing.
4: naive_bayes(): Feature abt - zero probabilities are present. Consider Laplace smoothing.
5: naive_bayes(): Feature accept - zero probabilities are present. Consider Laplace smoothing.
.
```

Estas advertencias son causadas por palabras que aparecieron en mensajes de spam cero o mensajes de ham cero y tienen poder de veto sobre el proceso de clasificación debido a sus probabilidades cero asociadas. Por ejemplo, debido a que la palabra accept solo apareció en mensajes de ham los datos de entrenamiento, no significa que todos los mensajes futuros con esta palabra deban clasificarse automáticamente como ham.

Hay una solución fácil para este problema utilizando el estimador de Laplace descrito anteriormente, pero por ahora, evaluaremos este modelo utilizando laplace = 0, que es la configuración predeterminada del modelo.

Paso 4 - evaluación del rendimiento del modelo

Para evaluar el clasificador SMS, necesitamos probar sus predicciones en los mensajes no vistos en los datos de prueba. Recuerda que las características de los mensajes no vistos se almacenan en una matriz llamada sms_test, mientras que las etiquetas de clase (spam o ham) se almacenan en un vector llamado sms_test_labels. El clasificador que hemos entrenado se ha llamado sms_classifier. Utilizaremos este clasificador para generar predicciones y luego comparar los valores predichos con los valores verdaderos.
La función predict() se utiliza para hacer las predicciones. Las almacenaremos en un vector llamado sms_test_pred. Simplemente suministramos a esta función los nombres de nuestro clasificador y el conjunto de datos de prueba como se muestra:

```
> sms_test_pred <- predict(sms_classifier, sms_test)
```

Para comparar las predicciones con los valores verdaderos, utilizaremos la función CrossTable() del paquete gmodels, que utilizamos anteriormente. Esta vez, agregaremos algunos parámetros adicionales para eliminar proporciones de celdas innecesarias y utilizaremos el parámetro dnn (nombres de dimensión) para volver a etiquetar las filas y columnas como se muestra en el siguiente código:

```
> library(gmodels)
> CrossTable(sms_test_pred, sms_test_labels,
+   prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+   dnn = c('predicted', 'actual'))
```

Esto produce la siguiente tabla:

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  1390


             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1201 |        30 |      1231 |
             |     0.864 |     0.022 |           |
-------------|-----------|-----------|-----------|
        spam |         6 |       153 |       159 |
             |     0.004 |     0.110 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
-------------|-----------|-----------|-----------|
```

Al observar la tabla, podemos ver que un total de solo 6 + 30 = 36 de 1,390 mensajes SMS se clasificaron incorrectamente (2.6 por ciento). Entre los errores, se encontraron 6 de 1,207 mensajes de ham que se identificaron erróneamente como spam y 30 de 183 mensajes de spam que se etiquetaron incorrectamente como ham.

Teniendo en cuenta el poco esfuerzo que pusimos en el proyecto, este nivel de rendimiento parece bastante impresionante. Este estudio de caso ejemplifica la razón por la que Naïve Bayes se utiliza tan a menudo para la clasificación de texto: directamente de fábrica, funciona sorprendentemente bien.

Por otro lado, los seis mensajes legítimos que se clasificaron incorrectamente como spam podrían causar problemas importantes para la implementación de nuestro algoritmo de

filtrado, ya que el filtro podría hacer que una persona no vea un mensaje de texto importante. Deberíamos intentar ver si podemos modificar ligeramente el modelo para lograr un mejor rendimiento.

Paso 5 - mejorar el rendimiento del modelo

Recordarás que no establecimos un valor para el estimador de Laplace al entrenar nuestro modelo; de hecho, ¡fue difícil pasar por alto el mensaje de R que nos advertía sobre más de 50 características con probabilidades cero! Para solucionar este problema, crearemos un modelo Naïve Bayes como antes, pero esta vez estableceremos laplace = 1:

```
> sms_classifier2 <- naive_bayes(sms_train, sms_train_labels,
+   laplace = 1)
```

A continuación, haremos predicciones como antes:

```
> sms_test_pred2 <- predict(sms_classifier2, sms_test)
```

Por último, compararemos las clases predichas con las clasificaciones reales mediante tabulación cruzada:

```
> CrossTable(sms_test_pred2, sms_test_labels,
+   prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
+   dnn = c('predicted', 'actual'))
```

Esto produce la siguiente tabla:

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  1390


             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1202 |        28 |      1230 |
             |     0.865 |     0.020 |           |
-------------|-----------|-----------|-----------|
        spam |         5 |       155 |       160 |
             |     0.004 |     0.112 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
-------------|-----------|-----------|-----------|
```

Añadir un estimador de Laplace estableciendo laplace = 1 redujo la cantidad de falsos positivos (mensajes de radioaficionados clasificados erróneamente como spam) de 6 a 5, y la

cantidad de falsos negativos de 30 a 28. Aunque esto parece un cambio pequeño, es sustancial considerando que la precisión del modelo ya era bastante impresionante.

Tendríamos que ser cuidadosos antes de ajustar demasiado el modelo, ya que es importante mantener un equilibrio entre ser demasiado agresivo y demasiado pasivo al filtrar el spam.

Los usuarios prefieren que una pequeña cantidad de mensajes de spam pasen por el filtro en lugar de la alternativa, en la que los mensajes de ham se filtran de manera demasiado agresiva.

Resumen

El clasificador Naïve Bayes se utiliza a menudo para la clasificación de texto. Para ilustrar su eficacia, empleamos Naïve Bayes en una tarea de clasificación que incluía mensajes SMS spam. La preparación de los datos de texto para el análisis requirió el uso de paquetes R especializados para el procesamiento y la visualización de texto.

Finalmente, el modelo pudo clasificar más del 97 por ciento de todos los mensajes SMS correctamente como spam o ham.

En el siguiente tema, examinaremos dos métodos de aprendizaje automático más. Cada uno realiza la clasificación mediante la partición de los datos en grupos de valores similares. Como descubrirás en breve, estos métodos son bastante útiles por sí solos. Sin embargo, si miramos más allá, estos algoritmos básicos también sirven como una base importante para algunos de los métodos de aprendizaje automático más poderosos que se conocen en la actualidad.