Naive Bayes Programming with R

Taylor Shrode

September 13, 2020

Introduction

The Naive Bayes algorithm describes a simple method to apply Bayes' theorem to classification problems. Bayes Theorem states

$$P(c|x) = \frac{P(x|c) * P(c)}{P(x)}$$

where $P(x|c)$ is the conditional probability (likelihood), $P(c)$ is the prior probability, and $P(x)$ is the probability of the predictor variables (Lantz, 2015). Naive Bayes assumes that all features in the dataset are equally important and independent, which are rarely true in most real-world applications (Lantz, 2015). The Naive Bayes classifier utilizes training data to calculate an observed probability of each outcome based on the evidence provided by feature values (Lantz, 2015). Later, the classifier is applied to unlabeled data and predicts the most likely class for the new features based on the observed probabilities (Lantz, 2015).

For this assignment, we will be using the Naive Bayes algorithm to classify SMS messages as a spam message or ham (not-spam) message. The dataset we will be using contains a public set of SMS labeled messages that have been collected for mobile phone spam research (Almeida, n.d.). A collection of 425 SMS spam messages were extracted from the Grumbletext website. An additional 3,375 SMS random ham messages were extracted from the NUS SMS Corpus (Almeida, n.d.). A list of 450 SMS ham messages were then collected from Caroline Tag's PhD Thesis. Finally, 1,002 ham and 322 spam messages were collected from the SMS Spam Corpus (Almeida, n.d.). Thus, the dataset contains 5574 SMS messages. It should be noted that the messages in the dataset are **not** chronologically sorted.

Libraries

Before we begin building our Naive Bayes model, we need to load the necessary libraries into R.

```
library("tm")
library("SnowballC")
library(wordcloud)
library(e1071)
library(gmodels)
library(caret)
```

```
library(DataExplorer)
library("RColorBrewer")
```

The **tm** package allows us to perform text mining. In other words, it allows us to remove numbers, punctuation, stopwords, and break apart sentences into individual words (Lantz, 2015). The **SnowballC** package allows us to apply their word stemming function, so that we can return a word in its root form (Lantz, 2015). The **wordcloud** package provides a simple function to create a word cloud. A word cloud is a way to visualize the frequency a word appears in text (Lantz, 2015). The **e1071** package is loaded so we can build our Naive Bayes model. The **gmodels** package allows us to evaluate our model by comparing the predictions to the true values. The **caret** package is loaded to allow us to create a confusion matrix, which outputs the accuracy of the model and a table that is used to visualize the performance of the model. The **DataExplorer** library allows us to perform data exploration analysis and the **RColorBrewer** package is loaded to apply their color palettes to our word clouds.

Load Data

To load our data into R, we can get our file directly from the online source (URL), unzip the files, and retrieve the file we need (Ngalo, 2018). First, we need to create a temporary file. Then, we "get" our online file using the **download.file()** function and place the zipped file in our temporary file. To extract the file from the temporary file, we need to use the **unz()** function and then we can use the **read.csv2** function to read and store our file in its own variable. Finally, we can remove the temporary file using the **unlink()** function (Ngalo, 2018).

```
temp <- tempfile() #create temporary file
download.file(
  "http://archive.ics.uci.edu/ml/machine-learning-
databases/00228/smsspamcollection.zip",
  temp) #get online file
sms_file <- unz(temp, "SMSSpamCollection") #extract file from temporary file
spam_df <- read.csv2(sms_file, header= FALSE, sep= "\t", quote= "",
                     col.names= c("type","text"), stringsAsFactors= FALSE)
#read file
unlink(temp) #remove temp file using unlink
```

Now, our dataset is stored in the **spam_df** dataframe.

Exploratory Data Analysis

To better understand our data, we can use the **str()** function.

```
str(spam_df)

## 'data.frame':    5574 obs. of  2 variables:
##  $ type: chr  "ham" "ham" "spam" "ham" ...
##  $ text: chr  "Go until jurong point, crazy.. Available only in bugis n
great world la e buffet... Cine there got amore wat..." "Ok lar... Joking wif
```

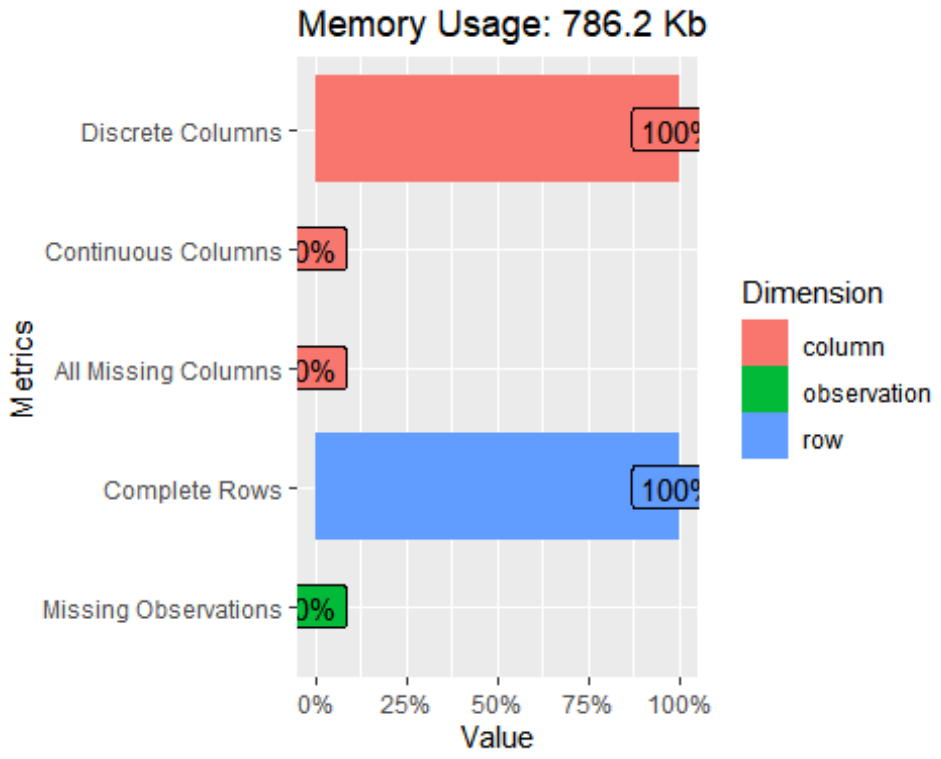
```
u oni..." "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May
2005. Text FA to 87121 to receive entry question("| __truncated__ "U dun say
so early hor... U c already then say..." ...
```

From our output above, we can see that we have a total of 5,574 SMS messages, which we can confirm by our dataset description above. We can also see that our dataframe contains two features: **type** and **text**. The SMS type has been labeled as **spam** or **ham** (not spam) in the **type** column, while the **text** column contains the full raw SMS text. We can see that the **type** column is currently a character vector. We need to convert this column into a factor because it contains categorical variables. The **factor()** function is used to do this.

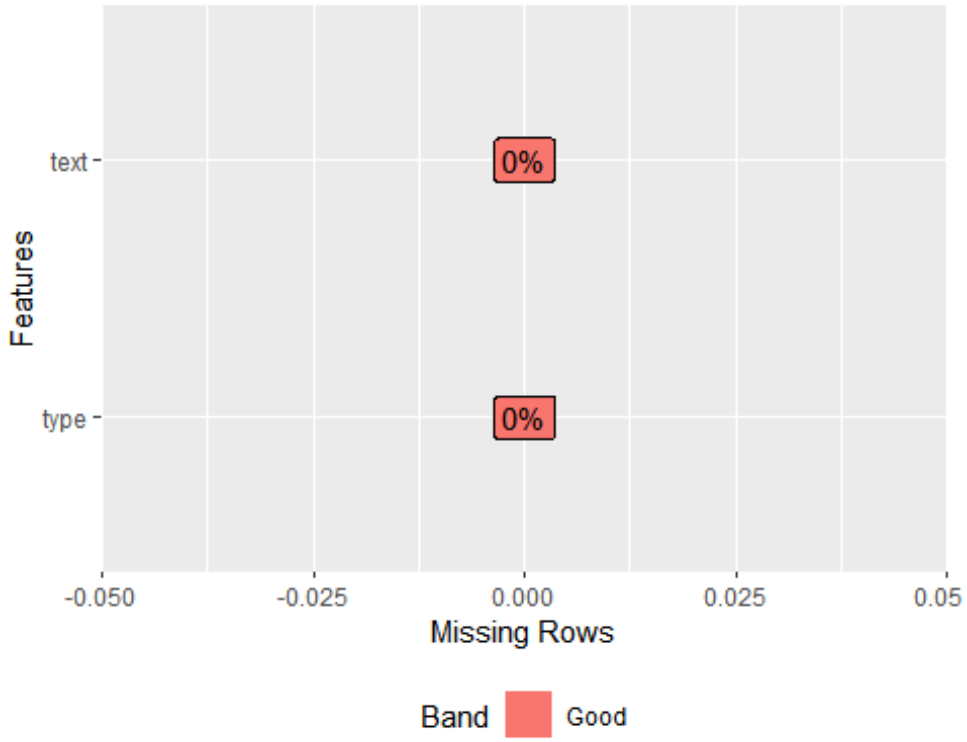
```
spam_df$type <- factor(spam_df$type)
str(spam_df$type)

##  Factor w/ 2 levels "ham","spam": 1 1 2 1 1 2 1 1 2 2 ...

table(spam_df$type)

##
##  ham spam
## 4827  747
```

To confirm that the column was converted from a character vector to a factor, we can use the **str()** function. The **table()** function allows us to see how how many SMS messages in our dataset have been labeled as spam and ham. We see that 4,827 SMS messages are labeled as ham, while 747 are labeled as spam. Now, the dataset description states that there are no missing values. We can confirm this by using the **plot_intro()** and **plot_missing()** functions in the **DataExplorer** package. This allows us to see how many complete rows we have and how many missing values we have and where they are.

```
plot_intro(spam_df)
```




```
plot_missing(spam_df)
```



From the outputs above, we can confirm that our dataset does not contain any missing values.

Cleaning and Standardizing Text Data

The SMS messages are strings of text composed of words, spaces, numbers, and punctuation (Lantz, 2015). To clean the our text, we can use the **tm_map()** function in the **tm** package. The first step to clean the text data is to create a "corpus", which is a collection of text documents (Lantz, 2015). To create our corpus, we will use the **VCorpus()** function, which refers to a volatile corpus. Volatile refers to the corpus being stored in memory rather than being stored on a disk (Lantz, 2015). The **VCorpus()** function requires us to specify the source of documents for the corpus. We will use the **VectorSource()** since we already have the SMS messages loaded into R.

```
sms_corpus <- VCorpus(VectorSource(spam_df$text))
print(sms_corpus)

## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 5574
```

By printing the **sms_corpus**, we can see that it contains documents for each SMS message. To receive a summary of specific messages, we can use the **inspect()** function. Below, we view a summary for the first and second SMS messages in the corpus. To view the actual message of a text message, we can use the **as.character()** function. To view the actual messages for more than one text, we need to use the **lappy()** function, which will apply the **as.character()** function to each element.

```
inspect(sms_corpus[1:2]) #can use list operators

## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 2
##
## [[1]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 111
##
## [[2]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 29

as.character(sms_corpus[[1]]) #view actual message text of one message

## [1] "Go until jurong point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat..."
```

```
lapply(sms_corpus[1:2], as.character) #view actual message text of multiple
messages

## $`1`
## [1] "Go until jurong point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat..."
##
## $`2`
## [1] "Ok lar... Joking wif u oni..."
```

As seen above, our corpus contains 5,574 text messages. To clean our data, we begin by standardizing the messages to use only lowercase characters by using the **tolower()** function wrapped with the **content_transformer()** function (Lantz, 2015). This treats the lowercase function as a transformation function that can be used to access the corpus. To confirm the transformation, we can use the **as.character()** function.

```
sms_corpus_clean <- tm_map(sms_corpus, content_transformer(tolower))
#standardize message to use only lowercase characters
as.character(sms_corpus_clean[[1]]) #confirm transformation

## [1] "go until jurong point, crazy.. available only in bugis n great world
la e buffet... cine there got amore wat..."
```

Now, we will remove numbers from our SMS messages because the majority of numbers will not provide useful patterns accross all messages. We can do this by using the **removeNumbers()** function. Then, we will filter out stop words, which are words like "and", "but", "or", etc. These words are not unique and appear very frequently, which will not provide useful information. We will remove stop words by using the **removeWords()** function and define the list of stop words by using the **stopwords()** function in the **tm()** package (Lantz, 2015). Next, we will remove punctuation using the **removePunctuation()** function. Another common standardization technique for text data involves reducing words to their root form using a process called *stemming* (Lantz, 2015). "This allows machine learning algorithms to treat the related terms as a single concept rather than attempting to learn a pattern for each variant (Lantz, 2015)." The **SnowballC** package can be integrated with the **tm** package so we can use the stemming function **stemDocument**. Finally, we will remove any additional whitespace by using the **stripWhitespace()** function (Lantz, 2015).

```
sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers) #remove numbers
  #use getTransformations() to get built-in transformations

sms_corpus_clean <- tm_map(sms_corpus_clean, removeWords, stopwords())
#remove stopwords

sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation) #remove
Punctuation

sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument) #apply wordStem to
entire corpus to perform stemming (stemDocument() needs SnowballC)
```

```
sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace) #final step:
remove additional whitespace
```

To confirm all of the transformations, we can compare the first five documents in our original corpus, **sms_corpus**, to the first five documents in our clean corpus, **sms_corpus_clean**.

```
lapply(sms_corpus[1:5], as.character) #original corpus

## $`1`
## [1] "Go until jurong point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat..."
##
## $`2`
## [1] "Ok lar... Joking wif u oni..."
##
## $`3`
## [1] "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005.
Text FA to 87121 to receive entry question(std txt rate)T&C's apply
08452810075over18's"
##
## $`4`
## [1] "U dun say so early hor... U c already then say..."
##
## $`5`
## [1] "Nah I don't think he goes to usf, he lives around here though"

lapply(sms_corpus_clean[1:5], as.character) #clean corpus

## $`1`
## [1] "go jurong point crazi avail bugi n great world la e buffet cine got
amor wat"
##
## $`2`
## [1] "ok lar joke wif u oni"
##
## $`3`
## [1] "free entri wkli comp win fa cup final tkts st may text fa receiv
entri questionstd txt ratetc appli s"
##
## $`4`
## [1] "u dun say earli hor u c alreadi say"
##
## $`5`
## [1] "nah think goe usf live around though"
```

For future use, we can simplify the steps above by creating a function to clean text data. The function takes in a corpus. Then, using the **tm_map** function, we can apply **tolower**, **removeNumbers**, **removeWords**, **removePunctuation**, **stemDocument**, and the **stripWhitespace** functions. It will then return the cleaned corpus.

```
clean_corpus <- function(corpus) {
  corpus <- tm_map(corpus, content_transformer(tolower)) #lowercase
  corpus <- tm_map(corpus, removeNumbers) #remove numbers
  corpus <- tm_map(corpus, removeWords, stopwords()) #remove stopwords
  corpus <- tm_map(corpus, removePunctuation) #remove Punctuation
  corpus <- tm_map(corpus, stemDocument) #apply wordStem to entire corpus to
perform stemming (stemDocument() needs SnowballC)
  corpus <- tm_map(corpus, stripWhitespace)
}
```

This will be useful when we separate the spam and ham documents from the original dataset.

Splitting Text Documents (SMS Messages) into Tokens (Words)

Now that our text data has been cleaned, we need to split the messages into individual words using a process called *tokenization*. A token is a single element of a text string. In this case, a token is a single word (Lantz, 2015). To tokenize the SMS messages, we can use the **DocumentTermMatrix()** function which takes in a corpus and creates a data structure called a *Document Term Matrix*. In a Document Term Matrix (DTM), the rows indicate documents (SMS messages) and the columns indicate terms (words) (Lantz, 2015). Each cell in the matrix stores a number which indicates a count of the times the word represented by the column appears in each document (Lantz, 2015). This structure is referred to as a *sparse matrix* because the majority of cells in the matrix are filled with zeros. Creating this matrix is simple when given a **tm** corpus.

```
#Split through tokenization with tm package
sms_dtm <- DocumentTermMatrix(sms_corpus_clean) #create a DTM sparse matrix;
will contain tokenized corpus
```

It should be noted that the preprocessing steps above can also be performed in this step by providing a list of **control** parameters (Lantz, 2015). We will see a slight difference in the number of terms in the matrix.

```
#could also preprocess during this step using:
sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
  tolower = TRUE,
  removeNumbers = TRUE,
  removePunctuation = TRUE,
  stopwords = TRUE,
  stemming = TRUE
))

#compare dtm to preprocessing with dtm
sms_dtm

## <<DocumentTermMatrix (documents: 5574, terms: 6630)>>
## Non-/sparse entries: 42680/36912940
## Sparsity           : 100%
```

```
## Maximal term length: 40
## Weighting          : term frequency (tf)

sms_dtm2

## <<DocumentTermMatrix (documents: 5574, terms: 6982)>>
## Non-/sparse entries: 43802/38873866
## Sparsity           : 100%
## Maximal term length: 40
## Weighting          : term frequency (tf)
```

The differences are due to the ordering of the preprocessing steps. The **DocumentTermMatrix()** function applies its cleanup functions to text strings after they have been split into words (Lantz, 2015).

Split Data into Training and Test Datasets

Now that our data has been prepared for analysis, we can split the data into training and test sets. Our classifier will be built and trained with the training set and then evaluated on unseen data: the test set. We will divide the data into two portions: 70% for training and 30% for testing. We can split the data manually because the SMS messages are sorted in random order (Lantz, 2015). Thus, our splits will be as follows:

```
## Original Dataset Size: 5574

## Training Set Size (70% of Original Dataset): 3902

## Testing Set Size (30% of Original Dataset): 1672
```

Therefore, we will use the first 3,902 documents for training and the remaining 1,672 for testing. The DTM object allows us to split the data like a dataframe using the standard **[rows, col]** operations. Recall, the DTM stores SMS messages as rows and words as columns. Thus, we must request a specific range of rows and all columns for the training and testing sets (Lantz, 2015). We will also save vectors that contain the labels for each the rows in the training set as well as the testing set. These are stored in the original **spam_df** dataframe.

```
sms_dtm_train <- sms_dtm[1:3902,]
sms_train_labels <- spam_df[1:3902,]$type
sms_dtm_test <- sms_dtm[3902:5574,]
sms_test_labels <- spam_df[3902:5574,]$type
```

Now, we can confirm that our subsets are representative of the complete SMS data by comparing the proportion of spam and ham in the training and test dataframes.

```
#confirm proportions
prop.table(table(sms_train_labels))

## sms_train_labels
##       ham      spam
## 0.8669913 0.1330087
```

```
prop.table(table(sms_test_labels))

## sms_test_labels
##       ham      spam
## 0.8637179 0.1362821
```

Both the training and testing sets contain the same amount of spam and ham messages, which suggests the data was divided evenly.

Word Clouds

One way to visualize the frequency of terms is to create a word cloud. A word cloud is a cloud composed of words scattered around the figure where words that appear often in the text are shown in a larger font (Lantz, 2015). Using the **wordcloud** package, we can create a word cloud using **wordcloud()** function. First, we will create a word cloud for our clean corpus (**sms_corpus_clean**). We will look at words that occur more than 50 times and we will plot the words in decreasing frequency (Fellows, n.d.). Further, we will use a **RColorBrewer** color palette (Text mining and word cloud fundamentals in R, n.d.).

```
wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE, colors =
brewer.pal(8,"Accent")) #wordcloud library; wordcloud for full clean corpus
```

The most popular terms are those with a larger font. From these terms, we see that “you”, “the”, and “have” are among the top frequent occurring words. Other top occurring words in the word cloud include “love”, “sorry”, “text”, “free”. Thus, a more interesting

visualization involves the word clouds for spam and ham SMS messages. To obtain these messages, we will subset our original dataframe where **type = spam** and **type = ham**.

```
spam_wc <- subset(spam_df, type == "spam") #get documents for type = spam
ham_wc <- subset(spam_df, type == "ham") #get documents for type = ham
```

Now, similar to the steps above, we need to create our spam corpus and ham corpus using the **VCorpus()** function. Then, using our corpus cleaning function defined above, we will clean the spam corpus and ham corpus.

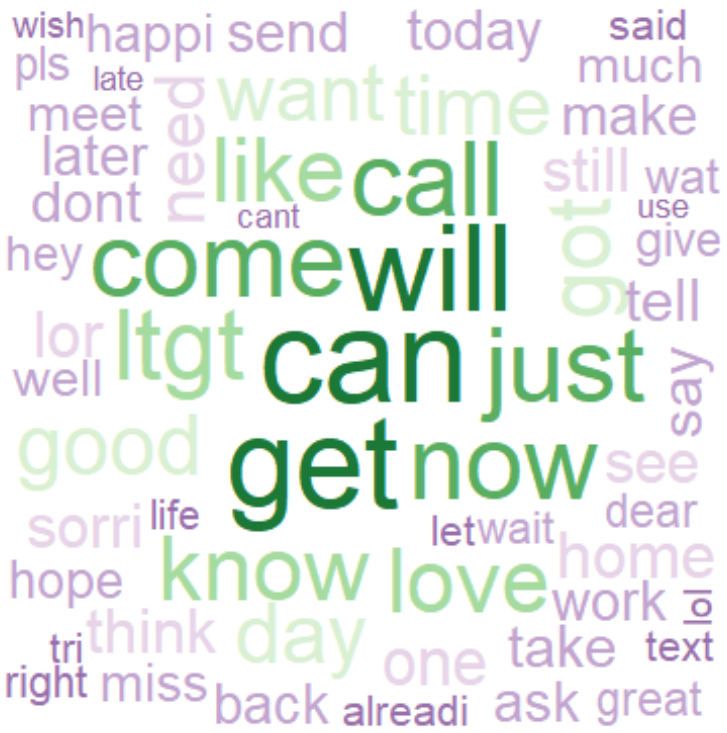
```
spam_corpus <- VCorpus(VectorSource(spam_wc$text))
spam_corpus_clean <- clean_corpus(spam_corpus)

ham_corpus <- VCorpus(VectorSource(ham_wc$text))
ham_corpus_clean <- clean_corpus(ham_corpus)
```

Using the clean corpus for spam SMS messages, we can create the word cloud.

```
wordcloud(spam_corpus_clean, random.order = FALSE, colors =
brewer.pal(8,"Spectral"), max.words = 100) #wordcloud for spam
```

From the word cloud above, we can see the top words are “call”, “free”, “text”, “mobil”, “servic”, “claim”, “cash”, etc. We can see patterns among the spam. One being that words in the messages are misspelled. The messages also claim to give away a free prize or want the user to claim money. Now, let’s look at our clean ham corpus word cloud.

```
wordcloud(ham_corpus_clean, random.order = FALSE, colors =
brewer.pal(8,"PRGn"), max.words = 100) #wordcloud for ham
```



The top words in our ham text messages seem to contain "normal" words like "good", "come", "call", "love", etc.

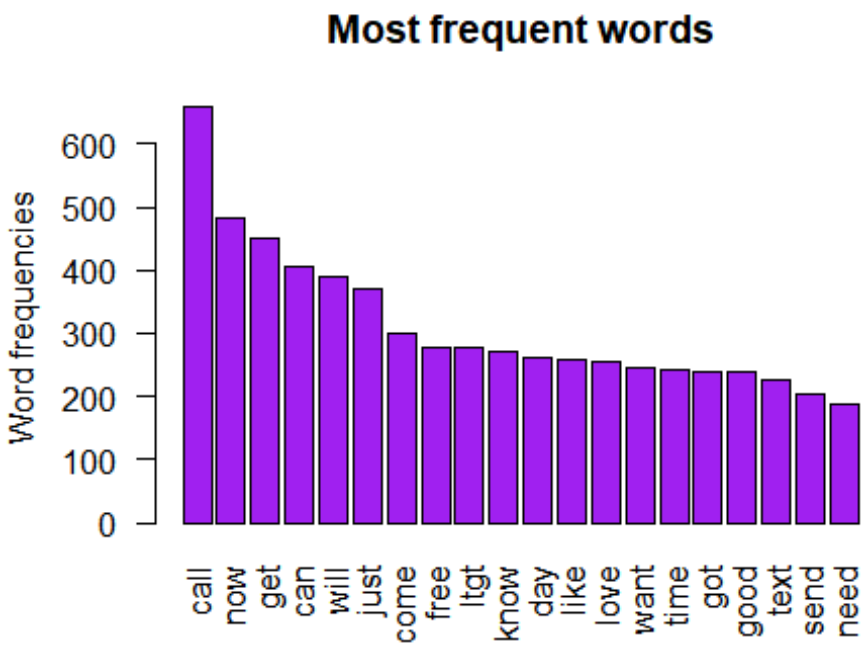
Explore Frequent Terms and Their Associations

We can also look at the frequent terms in a *Term Document Matrix*. A Term Document Matrix (TDM) is a way of representing the text as a table, where the rows represent the documents (SMS messages) and the columns represent the terms (words) (Facer, 2017). A TDM allows us to extract and view word counts and then plot our results (Text mining and word cloud fundamentals in R, n.d.). First, we will look at the top 10 words in our clean corpus.

```
#(Text mining and word cloud fundamentals in R, n.d.)
tdm <- TermDocumentMatrix(sms_corpus_clean)
m <- as.matrix(tdm)
v <- sort(rowSums(m), decreasing = TRUE)
d <- data.frame(word = names(v), freq = v)
head(d, 10)

##      word freq
## call call  658
## now   now  483
## get   get  451
## can   can  405
```

```
## will will  391
## just just  369
## come come  301
## free free  278
## ltgt ltgt  276
## know know  272
```

```
barplot(d[1:20,]$freq, las = 2, names.arg = d[1:20,]$word,
        col ="purple", main ="Most frequent words",
        ylab = "Word frequencies")
```



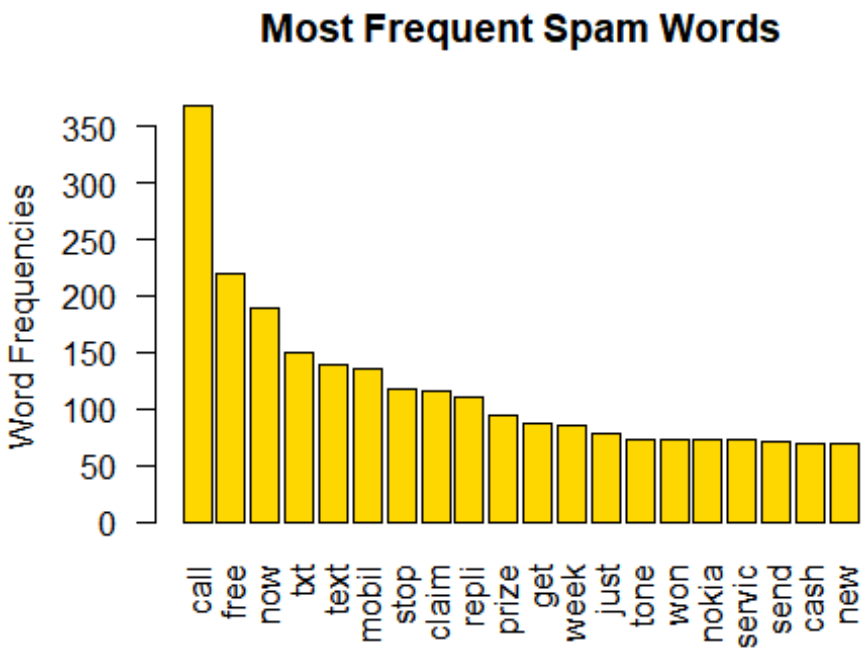
Again, a more interesting visual would be word frequencies for the spam corpus and the ham corpus. First, we will look at the spam corpus.

```
spam_tdm <- TermDocumentMatrix(spam_corpus_clean)
spam_m <- as.matrix(spam_tdm)
spam_v <- sort(rowSums(spam_m), decreasing = TRUE)
spam_d <- data.frame(word = names(spam_v), freq = spam_v)
head(spam_d, 10)
```

```
##        word freq
## call   call  368
## free   free  219
## now     now  190
## txt     txt  150
## text   text  139
## mobil mobil  136
```

```
## stop   stop  118
## claim claim  115
## repli repli  110
## prize prize   95
```

```
barplot(spam_d[1:20,]$freq, las = 2, names.arg = spam_d[1:20,]$word,
        col ="gold", main ="Most Frequent Spam Words",
        ylab = "Word Frequencies")
```



Again, we see the "free" and "prize" are among the top terms in our spam corpus. With our top frequent terms, we can analyze the association between these terms using the **findAssocs()** (Text mining and word cloud fundamentals in R, n.d.). For example, we will look at the term "free".

```
findAssocs(spam_tdm, terms = "free", corlimit = 0.25)
```

```
## $free
##           colour            updat             mths deliveredtomorrow
##             0.34             0.33             0.32             0.28
##            nokia            phone            orang           latest
##             0.28             0.28             0.26             0.25
```

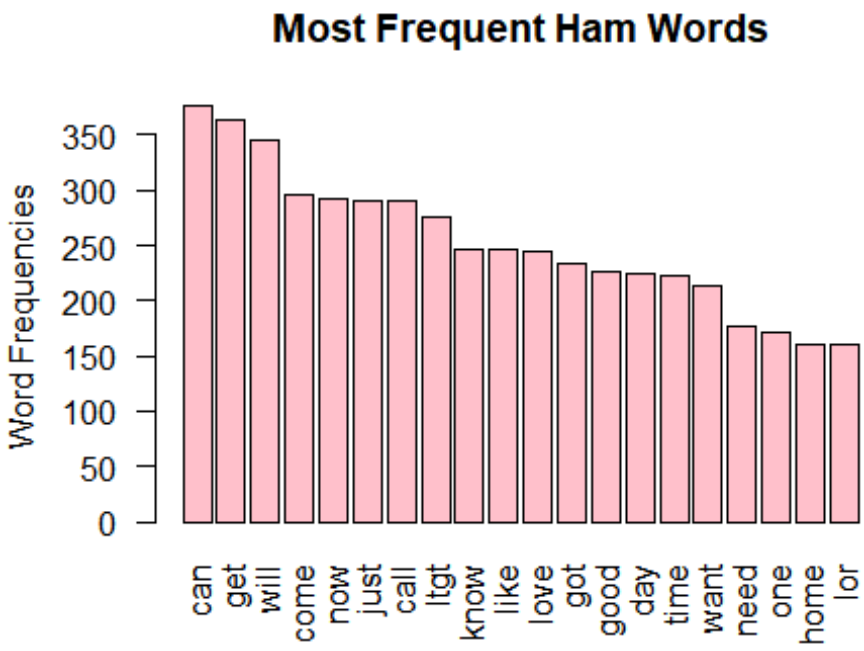
We can see that scam messages that include "free" refer to a free nokia, or that the free prize will be delivered tomorrow. Now, let's look at our ham corpus.

```
ham_corpus <- VCorpus(VectorSource(ham_wc$text))
ham_corpus_clean <- clean_corpus(ham_corpus)
```

```r
ham_tdm <- TermDocumentMatrix(ham_corpus_clean)
ham_m <- as.matrix(ham_tdm)
ham_v <- sort(rowSums(ham_m), decreasing = TRUE)
ham_d <- data.frame(word = names(ham_v), freq = ham_v)
head(ham_d, 10)
```

```
##      word freq
## can   can  376
## get   get  364
## will will  346
## come come  296
## now   now  293
## just just  291
## call call  290
## ltgt ltgt  276
## know know  246
## like like  246
```

```r
barplot(ham_d[1:20,]$freq, las = 2, names.arg = ham_d[1:20,]$word,
        col ="pink", main ="Most Frequent Ham Words",
        ylab = "Word Frequencies")
```



There seems to be no obvious pattern to the terms in our ham corpus. Now, we will use the **findAssocs()** function for the term "love".

```r
findAssocs(spam_tdm, terms = "love", corlimit = 0.25)
```

```
## $love
##     adam    lover    mobno    txtno    yahoo    heart     name      eve
##     0.76     0.76     0.76     0.76     0.76     0.62     0.57     0.47
##     logo poboxwwq  bedroom    buffi     fall  lapdanc    ppmsg  qlynnbv
##     0.47     0.47     0.27     0.27     0.27     0.27     0.27     0.27
##      sue    tarot
##     0.27     0.27
```

In reference to "love", the most associated terms include the name "adam" or the term "heart" and "lover".

Creating Indicator Features

The final step before train our Naive Bayes model is to transform our sparse matrix into a structure that can be used to train the model. Currently, our sparse matrix includes over 6,600 features, which is a feature for every word that appears in at least one SMS message. To reduce the number of features, we will eliminate any word that appears in less than five SMS messages (Lantz, 2015). We remove these terms because it is unlikely that these terms are useful for classification. We can remove these terms by using the **findFreqTerms()** function in the **tm** package (Lantz, 2015).

```
sms_freq_words <- findFreqTerms(sms_dtm_train, 5) #tm package; display words appearing at least 5 times in train matrix
str(sms_freq_words) #look at contents of freq words above

##  chr [1:1115] "â\200¦" "â\200“" "abiola" "abl" "abt" "accept" "access" ...
```

Now, we have 1,115 terms that appear in five or more SMS messages. We can filter our DTM to include only the terms in this vector. We will want to keep all the rows, but will be reducing the columns.

```
sms_dtm_freq_train <- sms_dtm_train[,sms_freq_words] #filter train and test DTM's to include terms in vector above
sms_dtm_freq_test <- sms_dtm_test[,sms_freq_words]
```

Recall, the cells in our sparse matrix are numeric and measure the number of times a word appears in a message (Lantz, 2015). We need to change this to a categorical variable that indicates yes or no depending on whether the word appears at all. We will use the function defined by Brett Lantz (2015) to convert the counts.

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

The function states that if the cell contains a number greater than 0, to convert the value to a "Yes", otherwise, change the value to "No". Now, we need to apply the function to the columns in our matrices.

```
#apply function to columns in matrix; MARGIN = 2 is for columns
sms_train <- apply(sms_dtm_freq_train, MARGIN = 2, convert_counts)
sms_test <- apply(sms_dtm_freq_test, MARGIN = 2, convert_counts)
```

Now, we have two matrices that contain "Yes" or "No" values to indicate whether the word represented by the column appears in the SMS message.

Train and Evaluate Naive Bayes Model

With our transformed SMS messages data, we can apply the Naive Bayes Algorithm, which will use the presence or absence of words to estimate the probability that a given SMS message is spam (Lantz, 2015). In the **e1071** package, we can use the **naiveBayes()** function to build our model using our training data.

```
sms_model <- naiveBayes(sms_train, sms_train_labels) #build model; e1071
package
```

Notice, in this model we are not including the Laplace estimator. Basically, "the Laplace estimator adds a small number to each of the counts in the frequency table, which ensures that each feature has a nonzero probability of occurring with each class. Typically, the Laplace estimator is set to 1, which ensures that each class-feature combination is found in the data at least once (Lantz, 2015)."

Now, we need to evaluate our model on the unseen data (test data).

```
sms_test_prediction <- predict(sms_model, sms_test) #test predictions on
unseen data (test data)
CrossTable(sms_test_prediction, sms_test_labels, prop.chisq = FALSE, prop.t =
FALSE,
           dnn = c("predicted", "actual")) #gmodels package; compare
predictions to true values

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1673
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
##          ham |      1435 |        25 |      1460 |
##              |     0.983 |     0.017 |     0.873 |
```

```
##              |     0.993 |     0.110 |           |
## -------------|-----------|-----------|-----------|
##         spam |        10 |       203 |       213 |
##              |     0.047 |     0.953 |     0.127 |
##              |     0.007 |     0.890 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1445 |       228 |      1673 |
##              |     0.864 |     0.136 |           |
## -------------|-----------|-----------|-----------|
##
##
```

```
confusionMatrix(sms_test_prediction, sms_test_labels)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  ham spam
##       ham  1435   25
##       spam   10  203
##
##                Accuracy : 0.9791
##                  95% CI : (0.971, 0.9854)
##     No Information Rate : 0.8637
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.9086
##
##  Mcnemar's Test P-Value : 0.01796
##
##             Sensitivity : 0.9931
##             Specificity : 0.8904
##          Pos Pred Value : 0.9829
##          Neg Pred Value : 0.9531
##              Prevalence : 0.8637
##          Detection Rate : 0.8577
##    Detection Prevalence : 0.8727
##       Balanced Accuracy : 0.9417
##
##        'Positive' Class : ham
##
```

Looking at the cross table, we can see that a total of 35 of the 1,673 SMS messages were incorrectly classified. In other words, 2% messages were incorrectly classified. Looking at our confusion matrix, our model is ~98% accurate. Now, we will build a model using the Laplace estimator and compare the results.

```
sms_model_2 <- naiveBayes(sms_train, sms_train_labels,laplace = 1)
sms_test_prediction_2 <- predict(sms_model_2,sms_test)
CrossTable(sms_test_prediction_2, sms_test_labels, prop.chisq = FALSE, prop.t
```

```
= FALSE, prop.r = FALSE,
           dnn = c("predicted", "actual"))

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1673
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
##          ham |      1414 |        14 |      1428 |
##              |     0.979 |     0.061 |           |
## -------------|-----------|-----------|-----------|
##         spam |        31 |       214 |       245 |
##              |     0.021 |     0.939 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1445 |       228 |      1673 |
##              |     0.864 |     0.136 |           |
## -------------|-----------|-----------|-----------|
##
##

confusionMatrix(sms_test_prediction_2, sms_test_labels)

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  ham spam
##       ham  1414   14
##       spam   31  214
##
##                Accuracy : 0.9731
##                  95% CI : (0.9642, 0.9803)
##     No Information Rate : 0.8637
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8892
##
##  Mcnemar's Test P-Value : 0.01707
##
##             Sensitivity : 0.9785
##             Specificity : 0.9386
```

```
##          Pos Pred Value : 0.9902
##          Neg Pred Value : 0.8735
##              Prevalence : 0.8637
##          Detection Rate : 0.8452
##    Detection Prevalence : 0.8536
##       Balanced Accuracy : 0.9586
##
##        'Positive' Class : ham
##
```

Now, looking at the cross table, we can see that a total of 45 of the 1,673 SMS messages were incorrectly classified. In other words, 3% messages were incorrectly classified. Using the Laplace estimator in the Naive Bayes model caused the number of incorrect classifications to increase. Also, according to the confusion matrix, we can see that the accuracy of the model dropped ~0.07%.

Conclusion

For this assignment, we classified SMS messages into spam or ham categories using Naive Bayes, which is often used to text classification. Before using the model, we used the **DataExplorer** package to verify that our dataset does not contain any missing values. Then, we used the **tm** and **SnowballC** packages to preprocess/clean our data. Next, using the **wordcloud** package to visualize the terms in our documents and identify patterns between our spam documents and our ham documents. Then, we trained our Naive Bayes classification algorithm and employed it on our testing data. We found that our model without the Laplace estimator was more accurate in classifying SMS messages correctly. To verify this, we created a confusion matrix. Ultimately, the Naive Bayes algorithm performed extremely well as it had an accuracy of ~98%.

Resources

Almeida , T. A. (n.d.). Uci machine learning repository: Sms spam collection data set. Retrieved September 13, 2020, from http://archive.ics.uci.edu/ml/datasets/SMS+Spam+Collection

Facer, C. (2017, February 26). Text analysis: Hooking up your term document matrix to custom r code. Displayr. https://www.displayr.com/text-analysis-hooking-up-your-term-document-matrix-to-custom-r-code/

Fellows, I. (n.d.). Wordcloud function. Retrieved September 13, 2020, from https://www.rdocumentation.org/packages/wordcloud/versions/2.6/topics/wordcloud

Lantz, B. (2015). Machine Learning with R : Expert Techniques for Predictive Modeling to Solve All Your Data Analysis Problems: Vol. Second edition. Packt Publishing.

Ngalo, D. (2018, June 21). Rpubs—Downloading and unzipping files in r. https://rpubs.com/otienodominic/398952

Text mining and word cloud fundamentals in R. (n.d.). Retrieved September 13, 2020, from http://www.sthda.com/english/wiki/text-mining-and-word-cloud-fundamentals-in-r-5-simple-steps-you-should-know