Use Naive Bayes Algorithm for Classification

Nithya Sarabudla

02-18-2024

Example – filtering mobile phone spam with the Naive Bayes algorithm

Step 1 – collecting data

Step 2 – exploring and preparing the data

```
# Reading the CSV file containing SMS data and storing it in the variable SMS.
SMS <- read.csv("/Users/nithyasarabudla/Downloads/spammsg.csv")
# Displaying the structure of the SMS dataset, showing the variables and their data types.
str(SMS)
```

```
## '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... C
```

The dataset contains information about SMS messages, including their text and whether they are considered spam or not. Each SMS is labeled either "spam" if it's an unwanted message or "ham" if it's a legitimate one. The dataset helps in understanding patterns and characteristics of spam messages, such as certain keywords or phrases that might indicate spam. This information can be used to develop a classifier that automatically identifies whether a given SMS is likely to be spam or ham based on its content.

```
# Converting the 'type' variable in the SMS dataset into a factor
SMS$type <- factor(SMS$type)
# Displaying the structure of the SMS dataset to verify that the 'type' variable has been correctly rec
str(SMS)
```

```
## 'data.frame':    5574 obs. of  2 variables:
##  $ type: Factor w/ 2 levels "ham","spam": 1 1 2 1 1 2 1 1 2 2 ...
##  $ text: chr  "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... C
```

```
# Creating a table to show the distribution of spam and ham messages in the dataset
table(SMS$type)
```

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

Preparing our SMS dataset for analysis by converting the 'type' variable, which indicates whether each message is spam or ham, into a factor. Converting it into a factor helps in categorical data handling, making it easier for analysis and modeling. The str() function is then used to verify that the 'type' variable has been correctly recoded as a factor. Lastly, the table() function is used to display the distribution of spam and ham messages in the dataset. This helps us understand the balance between the two classes of messages.

Data preparation – cleaning and standardizing text data

```
# Load the 'tm' package for text mining operations.
library(tm)
```

```
## Loading required package: NLP
```

```
# Create a corpus of SMS messages from the 'text' column of the SMS dataset.
sms_corpus <- VCorpus(VectorSource(SMS$text))
# Print the corpus to view its structure and content.
print(sms_corpus)
```

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

```
# Inspect the first two SMS messages in the corpus to verify data loading.
inspect(sms_corpus[1:2])
```

```
## <<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
```

Preparing our SMS data for analysis. Creating a corpus of SMS messages using the VCorpus() function from the tm package in R. This function helps us gather all the text documents together into one structured object called a corpus. Here, we're specifically using the VectorSource() function to specify the source of documents for our corpus, which is the 'text' column of our SMS dataset stored in the SMS$text vector. After creating the corpus, we print it to confirm that it contains the expected number of SMS messages. Additionally, we inspect the first two SMS messages in the corpus to ensure that the data is loaded correctly.

```
# Display the text content of the first SMS message in the original corpus.
as.character(sms_corpus[[1]])
```

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

Examining the content of the first SMS message in our corpus. By using the as.character() function on the first element of the corpus, we convert the document into a character vector, allowing us to view the actual text content of the SMS message.

```
# Use lapply to apply as.character to the first two documents in the corpus and view them.
lapply(sms_corpus[1:2], as.character)
```

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

To further verify the format and content of the SMS messages in our corpus, we apply the as.character() function to the first two documents using the lapply() function. This helps us confirm that the SMS messages are stored as expected and that the corpus is correctly constructed.

```
# Clean the text data by converting all letters to lowercase.
sms_corpus_clean <- tm_map(sms_corpus,
    content_transformer(tolower))
```

Cleaning the text data in our corpus by converting all letters to lowercase. This is important for standardizing the text and avoiding duplication of words due to case differences. We use the tm_map() function to apply the tolower() function to the entire corpus, creating a new corpus named sms_corpus_clean.

```
# Compare the original text content of the first SMS message with its cleaned lowercase version.
as.character(sms_corpus[[1]])
```

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

```
as.character(sms_corpus_clean[[1]])
```

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

Compare the original text content of the first SMS message with its cleaned version after applying the lowercase transformation. By displaying both the original and cleaned versions of the first message, we can verify that the transformation worked as expected, ensuring consistency in the text data for further analysis.

```
# Remove numerical digits from SMS messages.
sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

Focusing on cleaning up the SMS messages by removing any numerical digits present in them. Numerical digits in SMS messages don't usually contribute to meaningful patterns and are often unique to individual senders. To achieve this, we're using the removeNumbers() function from the tm package, which strips all numerical digits from the corpus, leaving only the text data intact.

```
# Eliminate common filler words (stopwords) from SMS messages.
sms_corpus_clean <- tm_map(sms_corpus_clean,
    removeWords, stopwords())
```

Next, aim is to get rid of common filler words, also known as stopwords, from the SMS messages. These words, such as "to", "and", "but", etc., appear frequently but don't provide much insight into distinguishing between spam and ham messages. To accomplish this, we're employing the removeWords() function along with the stopwords() function, both provided by the tm package. This combination allows us to remove common English stopwords from the corpus, helping to focus on more relevant content.

```r
# Remove punctuation marks from SMS messages.
sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

Removing punctuation marks from the SMS messages to tidy up the text data. Punctuation characters like periods, commas, and question marks are removed to avoid cluttering the text and ensure consistency in the corpus. To perform this task, we're using the removePunctuation() function from the tm package, which eliminates all punctuation from the corpus, leaving only alphanumeric characters.

```r
# Reduce words to their root form to improve analysis.
library(SnowballC)
sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument)
```

Performing stemming on the words present in the SMS messages. Stemming involves reducing words to their root form, which helps in treating related terms as a single concept rather than individual variants. To achieve this, we're utilizing the stemDocument() function from the SnowballC package. This function applies stemming to the entire corpus, ensuring that words like "learned", "learning", and "learns" are all transformed to their base form "learn".

```r
# Remove excess whitespace from cleaned SMS messages.
sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

Data cleanup process, focusing on removing any excess whitespace that may remain in the cleaned SMS messages. After removing numbers, stopwords, punctuation, and performing stemming, some extra spaces might still be present. To address this, we're using the stripWhitespace() function from the tm package. This function removes any additional whitespace from the text data, ensuring that the corpus is properly formatted and ready for further analysis.

Data preparation – splitting text documents into words

```r
# Generate a document-term matrix (DTM) from the cleaned SMS corpus.
sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
sms_dtm
```

```
## <<DocumentTermMatrix (documents: 5574, terms: 6592)>>
## Non-/sparse entries: 42608/36701200
## Sparsity           : 100%
## Maximal term length: 40
## Weighting          : term frequency (tf)
```

Creating a document-term matrix (DTM) from the cleaned SMS corpus. The DTM is a data structure where rows represent individual SMS messages, and columns represent unique words found across all messages. Each cell in the matrix stores the frequency count of each word in the respective message. This allows us to quantify the presence of specific words in each SMS message, aiding in further analysis and classification.

```r
# Generate another DTM, applying preprocessing steps within the function call.
sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
    tolower = TRUE,
    removeNumbers = TRUE,
    stopwords = TRUE,
    removePunctuation = TRUE,
```

```
    stemming = TRUE
))
sms_dtm2
```

```
## <<DocumentTermMatrix (documents: 5574, terms: 6995)>>
## Non-/sparse entries: 43713/38946417
## Sparsity           : 100%
## Maximal term length: 40
## Weighting          : term frequency (tf)
```

Creating another document-term matrix (DTM), but this time, we're applying preprocessing steps directly within the DocumentTermMatrix() function call. These steps include converting all text to lowercase, removing numbers, stopwords, punctuation, and performing stemming. This ensures that the DTM is generated from the raw, unprocessed SMS corpus while applying the necessary text preprocessing steps in one go. The resulting DTM provides an identical representation of the SMS messages, with each word appropriately tokenized and cleaned for analysis.

Data preparation – creating training and test datasets

```
# Divide the document-term matrix (DTM) into training and test datasets.
sms_dtm_train <- sms_dtm[1:4169, ]
sms_dtm_test  <- sms_dtm[4170:5559, ]
```

Splitting the document-term matrix (DTM) into training and test datasets. The training dataset, sms_dtm_train, consists of the first 4,169 rows of the DTM, while the test dataset, sms_dtm_test, comprises the remaining rows. This division ensures that both datasets represent a proportionate distribution of SMS messages, allowing for unbiased model training and evaluation.

```
# Extract labels (ham/spam) for the training and test datasets.
sms_train_labels <- SMS[1:4169, ]$type
sms_test_labels  <- SMS[4170:5559, ]$type
```

Extracting the corresponding labels for the training and test datasets. The sms_train_labels vector contains the labels (ham/spam) for the SMS messages in the training dataset, while the sms_test_labels vector holds the labels for the test dataset. These labels are retrieved from the original SMS dataset, ensuring consistency between the DTM subsets and their associated labels.

```
# Calculate the proportion of spam messages in the training dataset.
prop.table(table(sms_train_labels))
```

```
## sms_train_labels
##       ham      spam
## 0.8647158 0.1352842
```

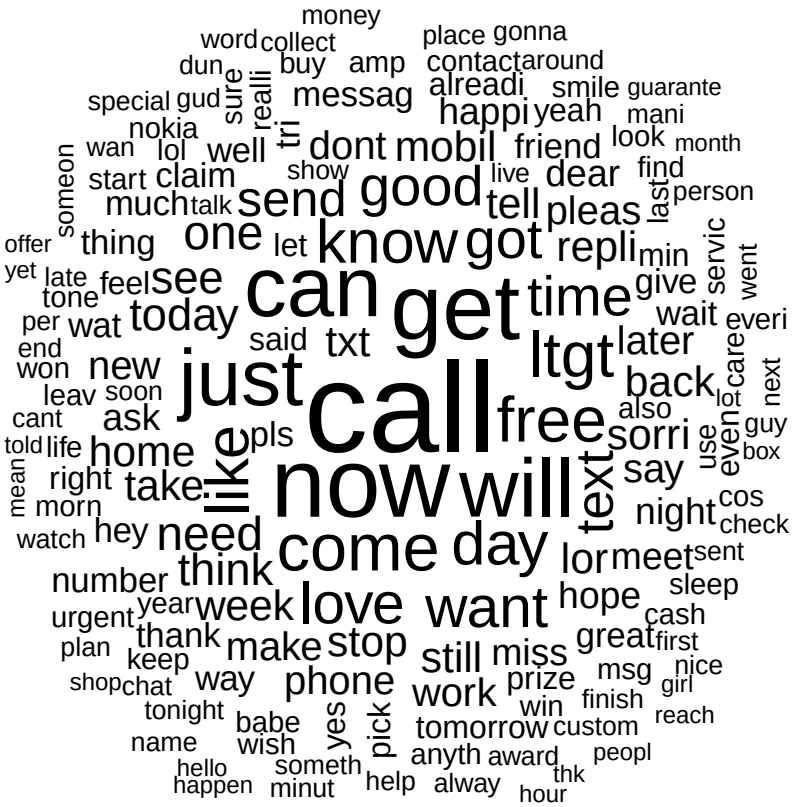
Calculating the proportion of spam messages within the training dataset. By using the prop.table() and table() functions, we determine the distribution of ham and spam messages in the training labels vector. The result shows that approximately 13% of the training dataset consists of spam messages, ensuring a balanced representation for training the spam classifier.

Visualizing text data – word clouds

```
# Load the wordcloud package
library(wordcloud)
```

```
## Loading required package: RColorBrewer
```

```
# Create a word cloud for SMS messages
wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
```



Utilize the wordcloud package to visually represent the frequency of words in our SMS messages. The wordcloud function generates a graphical representation where the size of each word corresponds to its frequency in the text. We create a word cloud from our preprocessed SMS corpus using parameters like min.freq to set a minimum frequency for inclusion and random.order to control the arrangement of words. By visualizing the most frequent words, we gain insights into the common themes or topics present in our text data.

```
# Subset the data for spam messages
spam <- subset(SMS, type == "spam")
```

Split our SMS data into subsets based on their classification as spam or ham. Using the subset function, we create a separate data frame named spam containing only spam messages. This subset enables us to create a word cloud specifically for spam messages, highlighting the most common terms used in such messages.

```r
# Subset the data for ham messages
ham <- subset(SMS, type == "ham")
```

Created another data frame named ham using the subset function to contain only ham messages. This subset allows us to generate a word cloud specifically for ham messages, showing the most frequent words used in non-spam messages.

```r
 # Create a word cloud for spam messages
wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
```

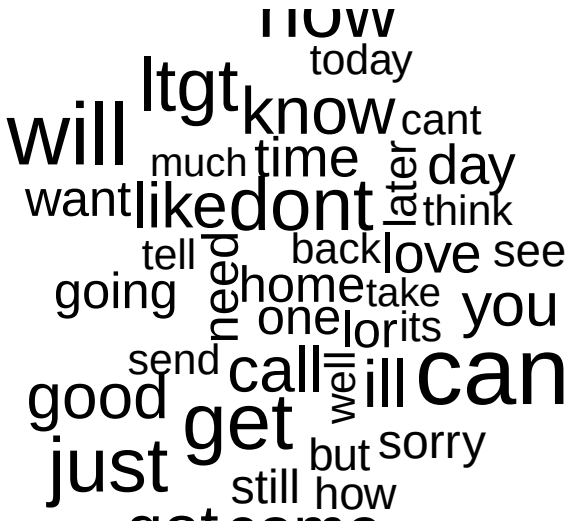
```
## Warning in tm_map.SimpleCorpus(corpus, tm::removePunctuation): transformation
## drops documents
```

```
## Warning in tm_map.SimpleCorpus(corpus, function(x) tm::removeWords(x,
## tm::stopwords())): transformation drops documents
```


```r
# Create a word cloud for ham messages
wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```

```
## Warning in tm_map.SimpleCorpus(corpus, tm::removePunctuation): transformation
## drops documents
```

```
## Warning in tm_map.SimpleCorpus(corpus, tm::removePunctuation): transformation
## drops documents
```



The wordcloud function to create word clouds for both spam and ham messages. By applying this function to each subset with parameters like max.words and scale, we generate word clouds that highlight the most common terms in each category. This comparison aids in understanding the distinctive language patterns between spam and ham messages.

Data preparation – creating indicator features for frequent words

```
# Find words appearing at least 5 times
sms_dict <- findFreqTerms(sms_dtm_train, 5)
```

Using the findFreqTerms function to identify words that appear at least 5 times in the training data. This helps us focus on words that are more likely to be informative for classification purposes. The result is stored in the variable sms_dict.

```
# Subset training data for frequent words
sms_dtm_freq_train <- sms_dtm_train[, sms_dict]
# Subset test data for frequent words
sms_dtm_freq_test <- sms_dtm_test[, sms_dict]
```

Subset both the training and test datasets to include only the columns corresponding to the frequent words identified earlier. This reduces the dimensionality of our data and ensures that we focus only on the relevant features for training and testing our classifier.

```r
# Define a function to convert counts into categorical variables
convert_counts <- function(x) {
  # Use ifelse to assign "Yes" if count is greater than 0, otherwise "No"
    x <- ifelse(x > 0, "Yes", "No")
}
```

To prepare the data for training a Naive Bayes classifier, we define a function convert_counts that converts numeric counts into categorical variables indicating whether a word appears in a message or not. Words with counts greater than 0 are labeled as "Yes", while words with counts equal to 0 are labeled as "No".

```r
# Convert counts to Yes or No for training data
sms_train <- apply(sms_dtm_freq_train, MARGIN = 2, convert_counts)
# Convert counts to Yes or No for test data
sms_test <- apply(sms_dtm_freq_test, MARGIN = 2, convert_counts)
```

Finally, applied the convert_counts function to each column (word) of both the training and test datasets using the apply function. This transforms the counts of word occurrences into categorical variables ("Yes" or "No") for each message, making the data suitable for training and evaluating our classifier.

Step 3 – training a model on the data

```r
# Load the e1071 package for Naive Bayes classifier
library(e1071)
# Train a Naive Bayes classifier on the preprocessed SMS data
sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

In this step, training a Naive Bayes classifier on our preprocessed SMS data. First load the e1071 package, which contains the naiveBayes function for building Naive Bayes models. Using the naiveBayes function, passED the training data (sms_train) and their corresponding labels (sms_train_labels) to create the classifier object sms_classifier. This classifier will be used to predict whether new SMS messages are spam or ham.

```r
# Check for any warning messages generated during training
warnings()
```

Checked for any warning messages that may have been generated during the training process. These warnings can provide valuable insights into potential issues or irregularities in the data or model. By running the warnings() function, we can view the warning messages that have been generated.

```r
# Re-train the Naive Bayes classifier after checking for warnings
sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

Following the warnings check, re-train the Naive Bayes classifier using the naiveBayes function once again. This step ensures that the classifier is properly trained and ready for use in the classification task. We use the same training data (sms_train) and labels (sms_train_labels) as before to train the model.

Step 4 – evaluating model performance

```r
# Generate predictions using the trained Naive Bayes classifier on the test data
sms_test_pred <- predict(sms_classifier, sms_test)
```

The trained Naive Bayes classifier to predict the classes of the test data. These predictions will be compared with the actual class labels to evaluate the performance of the model.

```r
# Load gmodels library and create contingency table
library(gmodels)
CrossTable(sms_test_pred, sms_test_labels,
    prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
    dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  1390
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
##          ham |      1200 |        20 |      1220 |
##              |     0.863 |     0.014 |           |
## -------------|-----------|-----------|-----------|
##         spam |         9 |       161 |       170 |
##              |     0.006 |     0.116 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1209 |       181 |      1390 |
## -------------|-----------|-----------|-----------|
##
##
```

This contingency table summarizes the performance of a Naive Bayes classifier on a test dataset consisting of 1390 observations. The rows represent the predicted classes, while the columns represent the actual classes. In the table, there are 1200 instances where the classifier correctly predicted messages as "ham" (non-spam) out of a total of 1220 actual "ham" messages, resulting in a high accuracy rate of approximately 86.3%. However, there are 20 instances where "ham" messages were incorrectly classified as "spam." For "spam" messages, the classifier correctly predicted 161 out of 170 actual "spam" messages, with an accuracy rate of approximately 91.6%. There are 9 instances where "spam" messages were incorrectly classified as "ham." Overall, the classifier demonstrates good performance, particularly in identifying "spam" messages, although there is a small number of misclassifications for both classes.

Step 5 – improving model performance

```r
# Train Naive Bayes Classifier with Laplace Estimator
sms_classifier2 <- naiveBayes(sms_train, sms_train_labels,
    laplace = 1)
```

In this step, a new Naive Bayes classifier is trained with a Laplace estimator set to 1, which helps address issues caused by zero probabilities in the training data.

```r
# Generate Predictions with Laplace Estimator
sms_test_pred2 <- predict(sms_classifier2, sms_test)
```

The newly trained classifier is used to generate predictions on the test data, incorporating the Laplace estimator for improved performance.

```r
# Evaluate Model Performance with Laplace Estimator
CrossTable(sms_test_pred2, sms_test_labels,
    prop.chisq = FALSE, prop.c = FALSE, prop.r = FALSE,
    dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  1390
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
##          ham |      1182 |        10 |      1192 |
##              |     0.850 |     0.007 |           |
## -------------|-----------|-----------|-----------|
##         spam |        27 |       171 |       198 |
##              |     0.019 |     0.123 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1209 |       181 |      1390 |
## -------------|-----------|-----------|-----------|
##
##
```

Again, the CrossTable function is utilized to evaluate the performance of the updated model by comparing predicted and actual classes in a contingency table. This allows us to observe any improvements made by adjusting the Laplace estimator. This contingency table summarizes the performance of a classification model on a test dataset comprising 1390 observations. The rows represent the predicted classes, while the columns represent the actual classes. Out of 1390 instances, the model correctly predicted 1182 "ham" (non-spam) messages and 171 "spam" messages. However, there were 37 misclassifications in total, with 27 "ham" messages incorrectly classified as "spam" and 10 "spam" messages incorrectly classified as "ham." Overall, the model demonstrates a relatively high accuracy for predicting "ham" messages (approximately 85.0%) but a slightly lower accuracy for predicting "spam" messages (approximately 91.3%).