Analysis Report for Project 1.

Spam Message Filter Using Bayes Algorithm

Programming Language: R

Presented by: Michael Owusu Asiamah

Problem Statement

The widespread use of mobile phones and computers has opened up a new avenue for electronic junk mail, known as SMS spam. SMS spam is particularly problematic because, unlike email spam, many cellular phone users pay a fee per SMS received. Developing a classification algorithm that could filter SMS spam would provide a valuable tool for cellular phone providers. This project aims to apply the Naive Bayes algorithm, which has been successfully used for email spam filtering, to SMS spam detection. SMS spam poses additional challenges for automated filters compared to email spam. SMS messages are often limited to 160 characters, reducing the amount of text that can be used to identify whether a message is junk. The limit, combined with small mobile phone keyboards, has led many to adopt a form of SMS shorthand lingo, which further blurs the line between legitimate messages and spam.

Problem Justification

By applying the Naive Bayes algorithm to SMS spam detection, this project seeks to provide a useful tool for cellular phone providers to combat the growing problem of SMS spam. The project will leverage the success of Naive Bayes in email spam filtering and adapt the algorithm to address the unique challenges posed by SMS spam.

Methodology

- Data Collection: This project used the Dataset sms_spam.csv from Collection at http://www.dt.fee.unicamp.br/~tiago/smsspamcollection/
- Data Preprocessing: This project was cleaned and processed by removing special characters, numbers, and punctuation marks. This step may also involve tokenization, stemming, and lemmatization to reduce the dimensionality of the feature space.
- Feature Extraction: Extracted relevant features from the preprocessed data, such as the frequency of words, character n-grams, or other textual features.

- Model Training: this project used Train the Naive Bayes classifier using the preprocessed data and labeled messages. The classifier will learn the probability distributions of the features in both the spam and ham classes.
- Model Evaluation: The Project was Evaluated by testing the performance of the Naive Bayes classifier using a separate test dataset.

Introduction

The Naive Bayes algorithm is based on Bayes' theorem, which is a fundamental theorem in probability theory. The theorem states that the probability of an event A given event B is related to the probability of event B given event A and the marginal probabilities of events A and B. Mathematically, Bayes' theorem is expressed as $P(A \mid B) = P(B)\ P(B|A) * P(A)$ In the context of the Naive Bayes algorithm, event A represents the class label (e.g., spam or not spam), and event B represents the features of the dataset (e.g., words in an email message). The algorithm assumes that the features are conditionally independent given the class label. This means that the probability of a feature given the class label can be estimated independently of other features. Mathematically, this assumption is expressed as $P(B|A)=\prod_{i=1}^{n} P(b_i|A)$ where n is the number of features in the dataset, and bi is the value of the i-th feature.

The Naive Bayes algorithm estimates the probabilities P(A|B) and P(B|A) using the observed data. It calculates the probability of a class label given the features by summing over all possible class labels: $P(A|B) = P(B) \sum_{c} P(B|c)P(c)$ The algorithm then predicts the class label with the highest estimated probability: $\hat{A} = \mathrm{argmax}_A P(A|B)$.

In summary, the Naive Bayes algorithm uses Bayes' theorem and the assumption of conditional independence of features to estimate the probabilities of class labels given the features of the dataset. It then predicts the class label with the highest estimated probability.

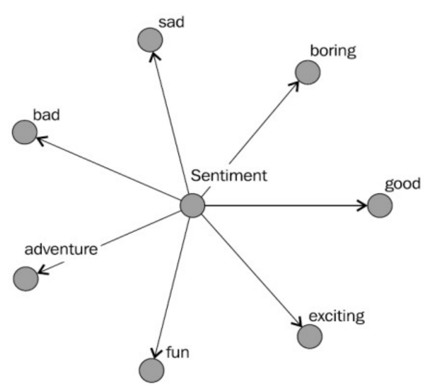
Classifiers based on Bayesian methods utilize training data to calculate an observed probability of each outcome based on the evidence provided by feature values. When the classifier is later applied to unlabeled data, it uses the observed probabilities to predict the most likely class for the new features. It's a simple idea, but it results in a method that often has results on par with more sophisticated algorithms. Bayesian classifiers have been used for:

- Text classification, such as junk e-mail (spam) filtering
- Intrusion or anomaly detection in computer networks
- Diagnosing medical conditions given a set of observed symptoms

Typically, Bayesian classifiers are best applied to problems in which the information from numerous attributes should be considered simultaneously to estimate the overall probability of an outcome. While many machine learning algorithms ignore features that have weak effects, Bayesian methods utilize all the available evidence to subtly change the predictions. If a large number of features have relatively minor effects, taken together, their combined impact could be quite large.

Naïve Bayesian Classifier:

A divergent Bayesian Network with multiple child nodes sharing a common parent node



Mathematically, let's assume that *Assume C* is the parent node and *Fi* are the children or feature nodes,

$$P(C \mid F_1,\ldots,F_n) = \frac{P(C)\cdot P(F_1,\ldots,F_n \mid C)}{P(F_1,\ldots,F_n)} \quad P(C \mid F_1,\ldots,F_n) = \frac{P(C)\cdot P(F_1 \mid C)\cdot\ldots\cdot P(F_n \mid C)}{P(F_1,\ldots,F_n)}$$

Our main objective is to choose the class *Ci* that maximizes the posterior probability *P(Ci|F1…Fn) which yields the following:*

$$Classify\, C_i : \underset{c}{\text{argmax}}\, P(C)\cdot\prod_{i=1}^{n} P(F_i \mid C)$$

From the above, it can be deduced that, for a given data, we can estimate the probabilities for different values of a feature. We can also estimate the proportion of observations assigned to a specific class. Example: For each feature (like words in a document), we can estimate the probability of its different values. We can also estimate the proportion of observations assigned to each class. This helps us understand the relationships between features and classes in the dataset.

Laplace Estimator

In statistics, the Laplace estimator, sometimes referred to as additive smoothing, is a method for smoothing count data and removing problems brought on by certain values having 0 occurrences. It is especially helpful when estimating the probabilities of class labels given the features of the dataset in the context of Naive Bayes classifiers. The basic idea of the Laplace estimator is to guarantee that each feature has a nonzero probability of occurring with each class by adding a small constant (usually 1) to each of the counts in the frequency table. This method assists in preventing circumstances in which certain words that have never before been used for a particular category suddenly surface, which would cause all of the computations to be zero.

Filtering Spam Messages via SMS:

Collecting and Cleaning Data

The first step in constructing our Naive Bayes classifier involves processing raw data into a bag-of-words representation, which simplifies text into a format that computers can understand. This representation ignores word order and provides a variable indicating whether a word appears in the text or not.

```
#Loading all the required packages
R version 4.3.1 (2023-06-16 ucrt) -- "Beagle Scouts"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> install.packages("e1071")
Installing package into ‘C:/Users/Kweku Cobbah/AppData/Local/R/win-
library/4.3’
(as ‘lib’ is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/e1071_1.7-
14.zip'
Content type 'application/zip' length 664712 bytes (649 KB)
downloaded 649 KB

package ‘e1071’ successfully unpacked and MD5 sums checked

The downloaded binary packages are in
```

```
        C:\Users\Public\Documents\Wondershare\CreatorTemp\Rtmp8kmyhZ\downloade
d_packages
> install.packages("tm")
Installing package into 'C:/Users/Kweku Cobbah/AppData/Local/R/win-
library/4.3'
(as 'lib' is unspecified)
trying URL 'https://cran.rstudio.com/bin/windows/contrib/4.3/tm_0.7-11.zip'
Content type 'application/zip' length 997770 bytes (974 KB)
downloaded 974 KB

package 'tm' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Public\Documents\Wondershare\CreatorTemp\Rtmp8kmyhZ\downloade
d_packages
> install.packages("snowballc")
Installing package into 'C:/Users/Kweku Cobbah/AppData/Local/R/win-
library/4.3'
(as 'lib' is unspecified)
Warning in install.packages :
  package 'snowballc' is not available for this version of R

A version of this package for your version of R might be available elsewhere,
see the ideas at
https://cran.r-project.org/doc/manuals/r-patched/R-admin.html#Installing-
packages
Warning in install.packages :
  Perhaps you meant 'SnowballC' ?
> install.packages("wordcloud")
Installing package into 'C:/Users/Kweku Cobbah/AppData/Local/R/win-
library/4.3'
(as 'lib' is unspecified)
trying URL
'https://cran.rstudio.com/bin/windows/contrib/4.3/wordcloud_2.6.zip'
Content type 'application/zip' length 447454 bytes (436 KB)
downloaded 436 KB

package 'wordcloud' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
        C:\Users\Public\Documents\Wondershare\CreatorTemp\Rtmp8kmyhZ\downloade
d_packages
> library(SnowballC)
> library(e1071)
Warning message:
package 'e1071' was built under R version 4.3.2
> library(tm)
Loading required package: NLP

Attaching package: 'tm'

The following object is masked _by_ '.GlobalEnv':
```

```
    stopwords

Warning message:
package 'tm' was built under R version 4.3.2
> library(wordcloud)
Loading required package: RColorBrewer
Warning message:
package 'wordcloud' was built under R version 4.3.2
> library(gmodels)
Warning message:
package 'gmodels' was built under R version 4.3.2
```
#The tm package refers to text mining which is used to remove numbers and punctuation; handle uninteresting words such as and, but, and or; and how to break apart sentences into individual words, "wordcloud" to build the wordcloud, and the "gmodels" to create the confusion matrix.

#Load the dataset and store it in a dataframe.
```
>data= read.csv("sms_spam.csv", stringsAsFactors = FALSE)
```

#examining the structure of the datasets: we can see 5559 observations(rows) and 2 variables in the spam dataset. SMS type has been coded as either ham or spam while the SMS text stores the full message content.
```
>str(data)
```

#since the type element is a character vector and is a categorical variable, we will convert it into a factor using the codes below:
```
>attach(data)
>data$type = factor(type)
```

#Examining this with the str() and table() functions,
we see that type has now been appropriately recoded as a factor.
Additionally, we see that 747 representing 13% of SMS messages in our data were labeled as spam, while 4,827 representing 87% were labeled as ham.
```
>str(data)
'data.frame':  5559 obs. of  2 variables:
 $ type: Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
 $ 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__ ...
> table(data$type)
 ham spam
4812  747
```

#The first step in processing text data is to build a corpus. we will therefore build a corpus to convert our dataset to a group of documents and they will be treated in corpus form. here we use the corpus() which means volatile corpus. The resulting corpus object is saved with the name data_corpus with the code indicated below:
```
>data_corpus =VCorpus(VectorSource(text))
> print(data_corpus)
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
```

```
Content:  documents: 5559
```

By printing the data_corpus, we see that it contains documents
for each of the 5,559 SMS messages in the training data:

#By showing the structure of our dataset, in the corpus object. We can use the inspect function to summarize the data structure as shown below to view a summary of the first and second SMS messages in the corpus:

```
>inspect =inspect(data_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
```

#view the actual message text, using the as. character() function the code below views 1 message:

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

#To view multiple messages, we'll use the lapply() function,
This function is part of the apply family, which includes apply(), sapply() etc.

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

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

#Now let's standardize words by removing punctuation and other characters. This process involves converting the text into a bag-of-words representation, which helps to simplify text into a format that computers can understand

#converting text to lowercase characters using R's tolower() function.

```
>data_corpus_clean = tm_map(data_corpus,content_transformer(tolower))
```

we can verify the first message in the original corpus and compare it to the same in the transformed corpus using the codes below:

```
>sms_corpus_clean = tm_map(data_corpus,content_transformer(tolower))
> as.character(data_corpus[[1]])
[1] "Hope you are having a good week. Just checking in"
```

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

Good, we can see uppercase letters are now replaced by small letters.

#Let's continue to remove numbers from the SMS messages using the tm_map() function.

```
>data_corpus_clean = tm_map(data_corpus_clean, removeNumbers)
```

#The command shown below is used to remove filler words by the use of the stopwords() function:

```
>data_corpus_clean = tm_map(data_corpus_clean,removeWords, stopwords())
```

#using the built-in removePunctuation() transformation to remove punctuations:

```
>data_corpus_clean = tm_map(data_corpus_clean, removePunctuation)
```

#function to replace punctuation

```
>replacePunctuation = function(x) {gsub("[[:punct:]]+", " ", x)}
>replacePunctuation("hello...world")
```

#Stemming helps to treat similar words representing the same information. To apply the wordStem() function to an entire corpus of text documents, the tm package includes a stemDocument() transformation. We apply this to our corpus with the tm_map() function exactly as done earlier:

```
> library(SnowballC)
> wordStem(c("learn", "learned", "learning", "learns"))
[1] "learn" "learn" "learn" "learn"
>data_corpus_clean = tm_map(data_corpus_clean, stemDocument)
```

#The final step of cleanup is removing whitespace from our dataset using stripWhitespace() transformation.

```
>data_corpus_clean = tm_map(data_corpus_clean, stripWhitespace)
```

#we can show the first 2 messages before and after data cleaning with the following codes:

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

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

> lapply(data_corpus_clean[1:2], as.character)
$`1`
[1] "hope good week just check"

$`2`
[1] "kgive back thank"
```

Data preparation – splitting text documents into words

#After the data is clean, the next step is to split the messages into individual components through a process called tokenization. now we can create sms_dtm object that contains the tokenized corpus using the default settings, which apply minimal processing.
data_dtm = DocumentTermMatrix(data_corpus_clean)

#prepossing incase we havent done it already:lets apply it to the unprocessed sms corpus as shown with this simple code:

```
>data_dtm1 = DocumentTermMatrix(data_corpus,
+control = list(tolower = TRUE,removeNumbers = TRUE,
+stopwords = TRUE,removePunctuation = TRUE,stemming = TRUE))
```

compare the two and check if they are the same

```
>data_dtm = DocumentTermMatrix(data_corpus_clean)
> data_dtm1 = DocumentTermMatrix(data_corpus,
+ control = list(tolower = TRUE,removeNumbers = TRUE,
+ stopwords = TRUE,removePunctuation = TRUE,stemming = TRUE))
> data_dtm
<<DocumentTermMatrix (documents: 5559, terms: 6967)>>
Non-/sparse entries: 56362/38673191
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
> data_dtm1
<<DocumentTermMatrix (documents: 5559, terms: 6961)>>
Non-/sparse entries: 43221/38652978
Sparsity           : 100%
Maximal term length: 40
Weighting          : term frequency (tf)
```

#correct these unmatched values by following code:

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

creating training and test datasets

#with our data prepared for analysis, we need to split it into training and test datasets. This ensures that our spam classifier can be evaluated on data it hasn't seen before. However, the split must occur after the data has been cleaned and processed, so the same preparation steps are applied to both datasets as shown below:

```
>data_dtm_train =data_dtm[1:4169, ]
>data_dtm_test  =data_dtm[4170:5559, ]
```

The code above will divide the data into two parts: 75%(4180) for training and 25%(1379) for testing. our DocumentTermMatrix object has also been split using standard [row, col] operations, as it acts like a data frame.

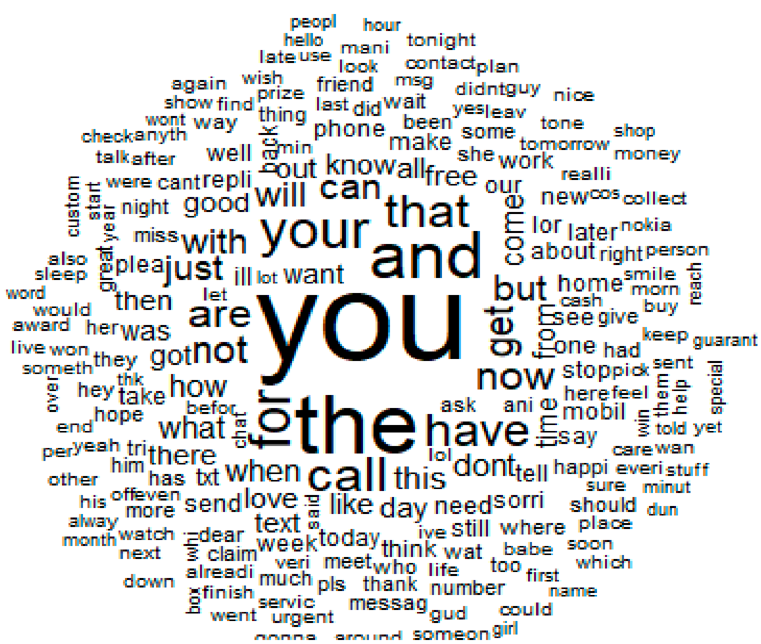
```
#We can save a pair of vectors with labels for each of the rows in the
training and testing matrices with the following codes:
>data_train_labels = data[1:4169, ]$type
>data_test_labels  = data[4170:5559, ]$type
```

```
#compare the proportion of spam in the training and test data frames to
confirm that the subsets are representative of the complete set of our data:
>prop.table(table(data_train_labels))
data_train_labels
      ham      spam
0.8647158 0.1352842
> prop.table(table(data_test_labels))
data_test_labels
      ham      spam
0.8683453 0.1316547
```

The training data and test data contain about 13 percent spam which depicts that the spam messages were divided evenly between the two datasets.

Visualizing text data – word clouds

```
#visuallize wordcloud
>cloud =wordcloud(data_corpus_clean, min.freq = 50, random.order = FALSE)
```

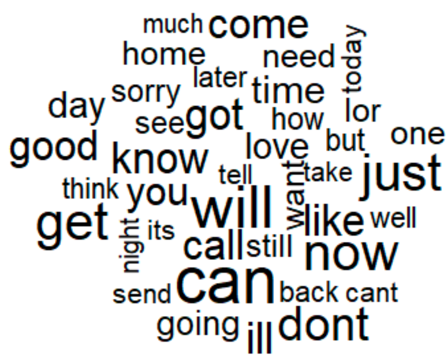



```
#we can now create a subset where the message type is labeled spam and ham
respectively using the code below:
>spam = subset(data, type == "spam")
```

```
>ham =subset(data, type =="ham")
```

#We can now visualize both spam and ham wordcloud by using the max.word parameter with the codes below:

```
>spam_cloud = wordcloud(spam$text, max.words=50, scale = c(3, 0.5))
Warning messages:
1: In tm_map.SimpleCorpus(corpus, tm::removePunctuation) :
  transformation drops documents
2: In tm_map.SimpleCorpus(corpus, function(x) tm::removeWords(x,
tm::stopwords())) :
  transformation drops documents
```




```
>ham_cloud = wordcloud(ham$text, max.words = 50, scale = c(3, 0.5))
Warning messages:
1: In tm_map.SimpleCorpus(corpus, tm::removePunctuation) :
  transformation drops documents
2: In tm_map.SimpleCorpus(corpus, function(x) tm::removeWords(x,
tm::stopwords())) :
  transformation drops documents
```

#From the cloud, it can be deduced that Spam messages include words such as now, free, reply, claim, and call; these terms do not appear in the ham cloud at all. Instead, ham messages use words such as can, sorry, need, and don't. These stark differences suggest that our Naive Bayes model will have some strong keywords to differentiate between the classes.

creating indicator features for frequent words

#The final step in the data preparation process is to transform the sparse matrix into a data structure that will train a Naive Bayes classifier. Since our spare matrix has over 6500 features and It's very unlikely that all of these are useful for our classification, we will eliminate any word that appears in less than five SMS messages, or less than about 0.1 percent of the records in the training data with the code below:

```
>sms_msg_freq_words = findFreqTerms(data_dtm_train, 5)
```

#view the character of the frequent words

```
>str(sms_msg_freq_words)
 chr [1:1139] "£wk" "€m̃" "€s̃" "abiola" "abl" "abt" "accept" ...
```

The above Code results show that 1,139 terms appear in at least five SMS messages:

#filter Document Term Matrix to include only the terms appearing in a specified vector. Since we want all the rows, but only the columns that represent the words in the mesg_freq_words vector, our commands are:

```
>sms_msg_dtm_freq_train = data_dtm_train[, sms_msg_freq_words]
>sms_msg_dtm_freq_test = data_dtm_test[, sms_msg_freq_words]
```

#since the cells in the sparse matrix are numeric and measure the number of times a word appears in a message and Naive Bayes only trained on categorical data, we will now change our spare matrix into categorical yes or No which indicates whether the word appears or not.
The following codes define a convert_counts() function to convert counts to Yes/No strings:

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

#now we can use convert_counts() for each of the columns in our sparse matrix for training and test matrices :

```
>sms_msg_train = apply(sms_msg_dtm_freq_train, MARGIN =2,  convert_counts)
>sms_msg_test = apply(sms_msg_dtm_freq_test, MARGIN =2,convert_counts)
```

Training a model on the data

#build our model on the sms_train matrix

```
>sms_msg_classifier = naiveBayes(sms_msg_train, data_train_labels)
```

The sms_classifier object now contains a naiveBayes classifier object that can be used to make predictions.

Evaluating the model performance

To evaluate the SMS classifier, we need to test its predictions on unseen messages in the test data. The classifier, named sms_msg_classifier while the class labels (spam or ham) are stored in a vector named data_test_labels, will be used to generate predictions and compare them to the true values. The predict() function will be used to make the predictions, and stores a vector named "sms_msg_test_pred".

```
>sms_msg_test_pred = predict(sms_msg_classifier, sms_msg_test)
```

#Compare the predictions to the true values, using the CrossTable() function from the gmodels package which helps us analyze the relationship between the predicted and true labels in our SMS spam dataset as follows:

```
>CrossTable(sms_msg_test_pred, data_test_labels,
+ prop.chisq = FALSE, prop.t = FALSE,
+ dnn = c('predicted', 'actual'))
```

```
   Cell Contents
|-------------------------|
|                       N |
|           N / Row Total |
|           N / Col Total |
|-------------------------|


Total Observations in Table:  1390


             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1201 |        30 |      1231 |
             |     0.976 |     0.024 |     0.886 |
             |     0.995 |     0.164 |           |
-------------|-----------|-----------|-----------|
        spam |         6 |       153 |       159 |
             |     0.038 |     0.962 |     0.114 |
             |     0.005 |     0.836 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
             |     0.868 |     0.132 |           |
-------------|-----------|-----------|-----------|
```

Results interpretation of confusion matrix

From the table above, Our email classification model has produced excellent results, achieving 93% overall accuracy. When we break it down, we find that spam detection is robust at 96.2%, while ham emails are correctly classified an amazing 97.6% of the time. These findings show how well our model can differentiate between unwanted and valid emails. We can with confidence suggest putting this model into use to improve email filtering and increase productivity."

improving model performance

we can improve our model by reducing the misclassification value of 30 using the Laplace estimator. To mitigate the frequency problem of 0, We can apply the Laplace estimator, smoothing, When the model doesn't find a word, Instead of putting a probability 0, it will put a counter 1 as shown in the code below:

```
>sms_msg_classifier2 =naiveBayes(sms_msg_train,data_train_labels,laplace = 1)
>sms_msg_test_pred2 = predict(sms_msg_classifier2, sms_msg_test)
CrossTable(sms_msg_test_pred2, data_test_labels,
prop.chisq = FALSE, prop.t = FALSE, prop.r = FALSE,dnn
=c('predicted','actual'))
```

Model improvement confusion matrix

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


Total Observations in Table:  1390


             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1202 |        28 |      1230 |
             |     0.996 |     0.153 |           |
-------------|-----------|-----------|-----------|
        spam |         5 |       155 |       160 |
             |     0.004 |     0.847 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
             |     0.868 |     0.132 |           |
-------------|-----------|-----------|-----------|
```

The number of false positives (ham messages mistakenly classified as spam) decreased to 5 with the addition of the Laplace estimator to the model, and the number of false negatives (spam messages mistakenly classified as ham) decreased to 28. This improvement, while seemingly small, is noteworthy given the model's high accuracy already. To avoid being too aggressive or too passive when filtering spam, care must be taken when making additional model adjustments. In general, users would rather have a small percentage of spam messages bypass the filter than an excessively aggressive filtering of ham messages.

Summary and conclusion

```
> confusionMatrix(table(sms_msg_test_pred, data_test_labels))
Confusion Matrix and Statistics

                 data_test_labels
sms_msg_test_pred  ham spam
             ham  1201   30
             spam    6  153

               Accuracy : 0.9741
                 95% CI : (0.9643, 0.9818)
    No Information Rate : 0.8683
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8801

 Mcnemar's Test P-Value : 0.0001264

            Sensitivity : 0.9950
            Specificity : 0.8361
         Pos Pred Value : 0.9756
         Neg Pred Value : 0.9623
             Prevalence : 0.8683
         Detection Rate : 0.8640
   Detection Prevalence : 0.8856
      Balanced Accuracy : 0.9155

       'Positive' Class : ham
```

In conclusion, Text classification frequently makes use of the Naive Bayes classifier. In the project, we used Naive Bayes on a spam SMS message classification task to demonstrate its efficacy. The utilization of specialized R packages for text processing and visualization was necessary to prepare the text data for analysis. In the end, the model successfully identified over 97% of all SMS messages as spam or ham.
When it comes to text classification, Naive Bayes is incredibly strong and even outperforms some advanced machine learning models.