Final Report

May 7, 2020

Intro

The goal of this project was to provide a package that would provide someone with the capability to analyze text data. We found a data set with many tweets in it. Each tweet was categorized as part of a disaster or not part of a disaster. We wanted to train a model that would classify each tweet into a "Yes" or "No" response. After the tweet was classified, we would then test it out to see what the misclassification rate of our model would be. In order to perform this task, we included functions from Python and R into one package.

The data that we used comprised of 7000 tweets. One column was the document id, another column was the actual text, and a third column was the response variable classifying the tweet as a 1 or a 0. In this case, tweets about a disaster were classified as a 1 and tweets that were not about a disaster were classified as a 0. An excerpt of what some of the tweets from the original data can be seen below:

```
## [1] "Forest fire near La Ronge Sask. Canada"
## [2] "All residents asked to 'shelter in place' are being notified by officers. No other evacuation o
## [3] "13,000 people receive #wildfires evacuation orders in California "
## [4] "Just got sent this photo from Ruby #Alaska as smoke from #wildfires pours into a school "
## [5] "#RockyFire Update => California Hwy. 20 closed in both directions due to Lake County fire - #CA
```

Cleaning the Data in R

In order to work with the data, we found that turning the list of tweets into a "corpus" would be a good starting point to improve the computation time. The package includes functions that turn the initially turns the data into a corpus. Once we had our corpus, we began to clean the data. Some of these cleaning steps included turning the words to lower case, removing extra spaces, removing numbers, removing puncuation, and removing non-English words. Probably one of the most interesting tasks when cleaning the text data was to "stem" the words. Consider words like "compute", "computed", "computes", and "computing." To consider all of these words as different would be doing our model a disservice. The idea of steming words is that we chop off the last part and turn them all into a similar word such as "compute." After this cleaning process, we turned the data into a document term matrix. A document term matrix is a matrix where every row represents a document (tweet) and every column represents a word. The number of times that each word occurs in each document is listed in this document term matrix. A brief excerpt of what a document term matrix looks like can be seen below:

```
##      Terms
## Docs aba abandon abc abcnew abl ablaz absolut abstorm abus access
##    1   0       0   0      0   0     0       0       0    0      0
##    4   0       0   0      0   0     0       0       0    0      0
##    5   0       0   0      0   0     0       0       0    0      0
##    6   0       0   0      0   0     0       0       0    0      0
##    7   0       0   0      0   0     0       0       0    0      0
```

Notice that the excerpt used contains many zeros. The matrix is mostly sparse. In fact, there are only about 50,000 out 17,600,000 non-zero entries. It can be difficult to visualize the data when analyzing text so we decided that using a word cloud would be really helpful. The function `make.word.cloud` creates a word cloud of the 50 words that occured the most. This ended up being a lot easier task then we orignially anticipated

since there was a package that already did most of the work upfront. We thought it was a useful tool though so we decided to aggregate it into our final project for text analysis. For a word cloud, the size of each word represents how many times it occurs. We decided to use color as a random variable because we were trying to create a really striking data visualization. An example of a word cloud from our training data can be seen below:

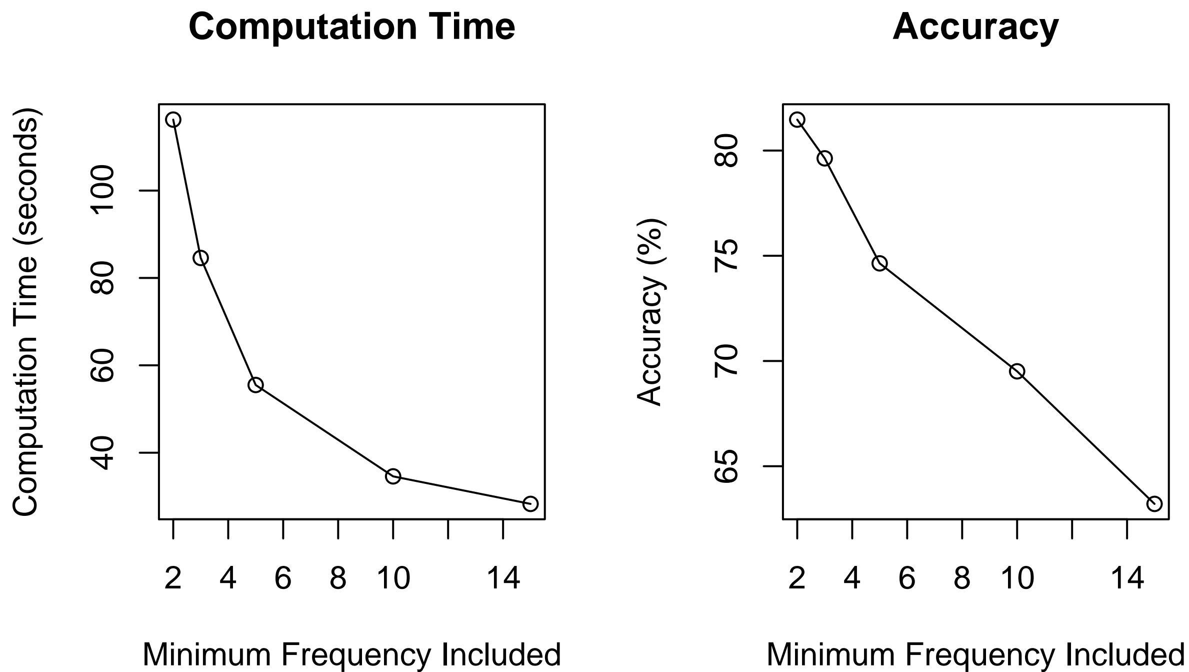
Cleaning the Data in with previous Python script

One goal of this project was to integrate python code with R. This was primarily done using the `reticulate` package. In this project, we primarily utilized the abilities of directly importing a python module and smoothly converting between python and R data types. This allowed us to use previously a previously written python script (author is Jessie), which cleaned text data, and stored counts with words in a python list. Although R has similar capabilities with it's own libraries to handle text, it is important to not redo work for already written code. The python module utilized the NLTK (Natural Language Toolkit) and gives options of removing digits, removing stop words, removing hyperlinks, and using the Porter Stemming Algorithm to stem words. However, when calling it in R, all those options are forced to be true. One difference between this and what was described above is the format of data returned. With R, the data is initially drawn in as a DocumentTermMatrix, but here it is loaded in as a large matrix in which the rows represent the sentences/tweets and the columns represent a word. Upon loading, a global variable `importM` is created which houses all the python functions.

```
tweetsFromPython = readAndCleanWords(dataFile,3)$counts
labels = getLabels(dataFile, 4)
```

Analyzing the Data in R

As far as analyzing the data in R goes, we used Naive Bayes Spam Filtering. We originally anticipated using a combination of the EM algorithm for analyzing our text but after submitting our topic proposal, we noticed that you suggested Naive Bayes Spam Filtering. After doing a little bit of reading online, we realized that this is regularly regarded as being a very quick and accurate way to go about doing text analsis so we built a function based on that method. The text data was cleaned using our other functions and then we were able to do Naive Bayes Spam Filtering while tweaking two parameters. We have the ability with our function to tweak how much training and how much test data is used. We also have the ability to tweak what the minimum frequency of each word to occur is included. For our own curiousity, we decided to compare accuracy of prediction and computation time based on the minimum frequency used in Naive Bayes Spam Filtering. Notice there are two plots — one is based on computation time and the other is based on accuracy.



As expected, lowering the minimum frequency has a profound impact on computation time and accuracy. As the amount of data decreases, accuracy and computation time both go down. This project was really unique for us because previously, we had no experience with text analysis and we both got a lot of practice with looking at strings and figuring out how to analyze them. We think it is really interesting that without sentence structure or word order, the content of a sentence can be classified correctly up to eighty percent of the time. Thanks for the opportunity to explore such an interesting topic!

Home Grown Naive Bayes

Another option to using the `naiveBayes` function in the `e1071` package was to write our own. For this implementation, we only consider the binary case. Naive Bayes works by calculating conditional probabilities of a data point occuring given a particular class $P(x_1, x_2...|Class_i)$. However, the goal is achieve the reverse, $P(Class_i|x_1, x_2, ...)$. We know by from Bayes Rule that $P(x_1, x_2...|Class_i) * P(Class_i) \propto P(Class_i|x_1, x_2...)$, and if the assumption of independence is made the calculations needed are mainly the multiplication of probabilities. However, we know that we must take sums of logs instead of products. In order to avoid the problem of having probabilities of 0, a small constant was added (1e-5). In this dataset, that probability is

less than just seeing the word once. Because of the large size of the data, this calcuation can sometimes be slow, so given a cluster created from the `parallel` package can be given as input to speed up calculation.

```
cl = makeCluster(2)
system.time(testNB.binary(tweetsFromPython[1:40,], tweetsFromPython[41:7000,], labels[41:7000]))
```

```
##    user  system elapsed
##  27.786   2.568  30.372
```

```
system.time(testNB.binary(tweetsFromPython[1:20,], tweetsFromPython[21:7000,], labels[21:7000], cl))
```

```
##    user  system elapsed
##   1.578   0.232   9.854
```

```
stopCluster(cl)
```

Cross Validation for different models

One way to choose a model is through cross-validation. By testing different models on different folds of the data, we are able to get a clearer picture of which model generalizes best. One way to reduce the model is by selecting words based on their frequency. Here we can choose a range of frequencies and compare the mean squared error (which in this case is just accuracy).

```
mses = CVtoChooseModel(c(4,4,4), c(4,5,6), tweetsFromPython, labels, N = 5)
mses
```

```
##            mses
## 4 - 4 0.4133721
## 4 - 5 0.4082496
## 4 - 6 0.3976091
```

Here we can see that having more words in the model does slightly better in accuracy. Alternitively, we can just pick a word frequency to narrow down our dataset and do cross-validation on the whole model

```
sixwords = reduceWords(6, tweetsFromPython, 6)
sixcv = NBcv(sixwords, labels, 10)
sixcv$aMSE
```

```
## [1] 0.4104782
```

When using Naive bayes with all words in the model, we get near 78% accuracy with our function. This is validated with 10-fold-cross validation and is in line with what is seen with the naiveBayes function in the previously used package.