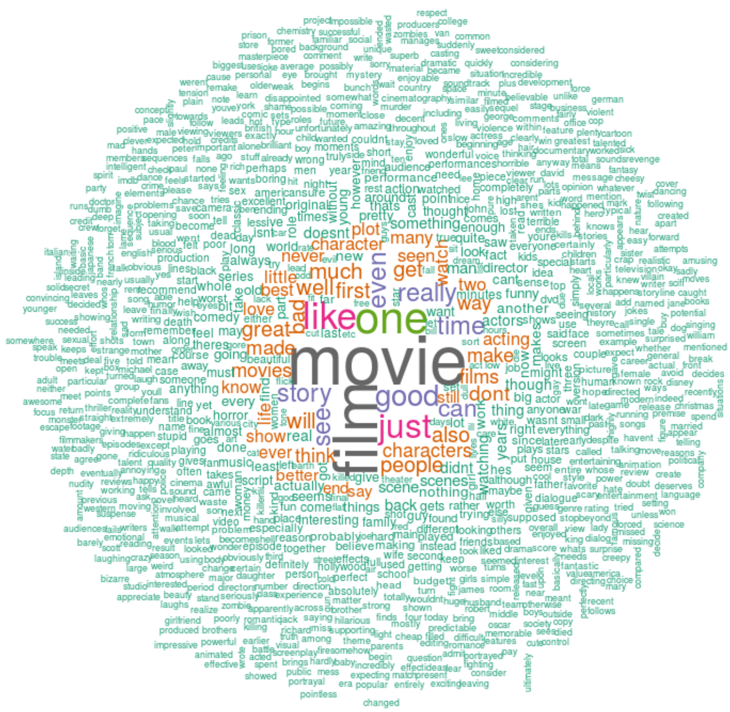
Sentiment Analysis and Text Mining
"Bag of Words Meets Bags of Popcorn"
A Quick R Demonstration

John Koo

June 29, 2015



The Data

Kaggle hosts various data science challenges ranging from tutorials aimed at budding data scientists, to $100,000-prize competitions. One such challenge, dubbed "Bag of Words Meets Bags of Popcorn", aims to train an algorithm that predicts movie review sentiments.

The data consists of 25,000 IMDB reviews attached by a binary sentiment score: 0 if the user rated the movie < 5 or 1 if > 6. Neutral reviews are omitted. The goal is to attach sentiment scores to 25,000 additional unlabeled reviews, and the resulting prediction is scored using the AUC metric. Another 50,000 unlabeled reviews are also provided to aid in NLP feature extraction.

The reviews are first cleaned of punctuation and HTML tags.

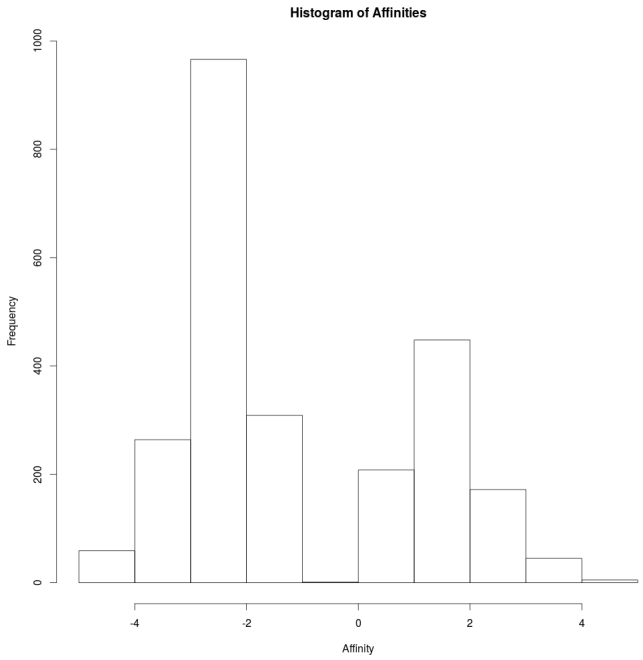
```
# read the data
train.labeled <- read.delim('~/Documents/bag of words/labeledTrainData.tsv',
                            header = T, quote = '', stringsAsFactors = F)
train.unlabeled <- read.delim('~/Documents/bag of words/unlabeledTrainData.tsv',
                              header = T, quote = '', stringsAsFactors = F)
test <- read.delim('~/Documents/bag of words/testData.tsv',
                   header = T, quote = '', stringsAsFactors = F)

# combine the data
all.data <- rbind(train.labeled[, -2], train.unlabeled, test)
all.data$sentiment <- c(train.labeled$sentiment,
```

```
                        rep(NA, nrow(train.unlabeled) + nrow(test)))

# index
train.labeled.ind <- 1:nrow(train.labeled)
train.ind <- 1:(nrow(train.labeled) + nrow(train.unlabeled))
test.ind <- (nrow(train.labeled) + nrow(train.unlabeled) + 1):nrow(all.data)

# clean up
# remove HTML tags
all.data$review.clean <- gsub('<.*?>', ' ', all.data$review)
# remove grammar/punctuation
all.data$review.clean <- tolower(gsub('[[:punct:]]', '', all.data$review.clean))
```

AFINN List

The University of Denmark provides a list of 2,477 words and phrases labeled with sentiment ratings between -5 and 5 (called the AFINN word list).

```
        word score
  invincible     2
    mirthful     3
       flops    -2
hypocritical    -2
       upset    -2
  overlooked    -1
   hooligans    -2
     welcome     2
      engage     1
        ache    -2
     defects    -3
  terrorizes    -3
      annoys    -2
     sulking    -2
   rejoicing     4
   dauntless     2
```



First, we read in the list and do a bit of cleaning:

```
# read in the AFINN list
afinn <- read.delim('~/Documents/bag of words/AFINN/AFINN-111.txt',
                    header = F, quote = '', stringsAsFactors = F)
names(afinn) <- c('word', 'score')

# spaces are denoted by "-", so we need to clean it up a bit
afinn$word.clean <- gsub('-', ' ', afinn$word)
# one phrase in the list includes an apostrophe.
afinn$word.clean <- gsub("[[:punct:]]", '', afinn$word.clean)
```

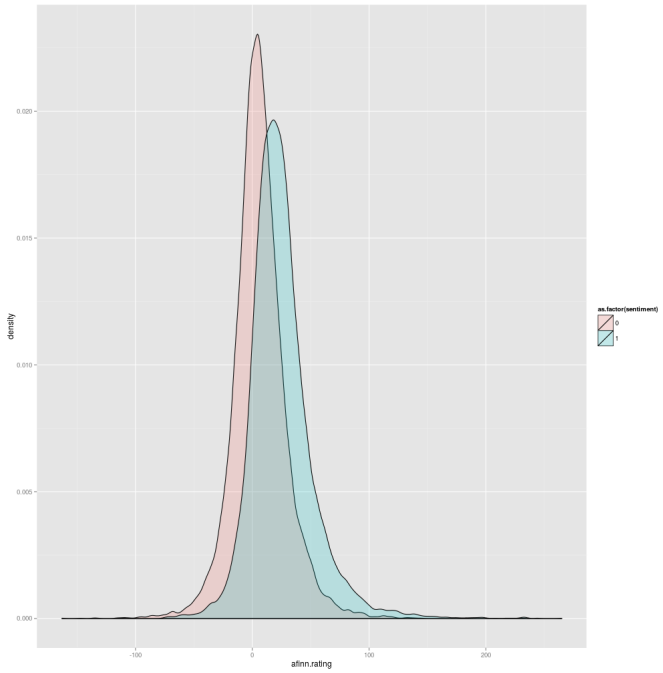
The AFINN list can be found at http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010.

A quick and dirty method would be to just count up the number of times each term occurs in each review and dot product that by the corresponding sentiment vector. Here, we use the str_count() function from the stringr package to count up the number of instances of each AFINN term in each review to obtain a text frequency matrix.

```
# find the text frequency matrix
# this may take a while ...
require(stringr)
tf <- t(apply(t(all.data$review.clean), 2,
              function(x) str_count(x, afinn$word.clean)))

# sentiment rating for each row
all.data$afinn.rating <- as.vector(tf %*% afinn$score)
```

The good thing about this method is it's very simple and takes in just one predictor. The bad thing is there's a huge overlap between the two classes.

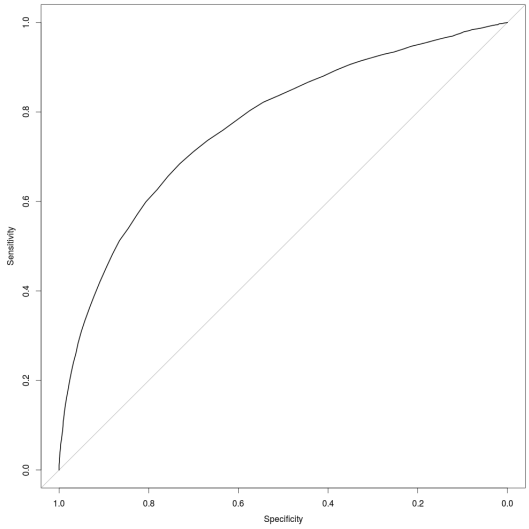
```
require(ggplot2)
ggplot(all.data[train.labeled.ind, ],
       aes(afinn.rating, fill = as.factor(sentiment))) +
    geom_density(alpha = .2)
```



Since the distributions are approximately normal (or at least unimodal), we'll try training a Naive Bayes classifier to the data.

```
require(e1071)
# fit a naive bayes classifier
nb.model <- naiveBayes(sentiment ~ afinn.rating,
                       data = all.data[train.labeled.ind, ])
# predicted values
nb.pred <- predict(nb.model,
                   newdata = all.data[test.ind, ],
                   type = 'raw')
# format results for submission
results <- data.frame(id = all.data$id[test.ind],
                      sentiment = nb.pred[, 2])
results$id <- gsub('"', '', results$id)
write.table(results,
            file = '~/Documents/bag of words/afinn_rating.csv',
            sep = ',', row.names = F, quote = F)
```

As expected, the results aren't all that great, giving us an AUC of .71516. But it's a start.



ROC curve constructed by predicting on the training data, AUC = .7701

Bag of Scores

This method is similar to the "bag of centroids" method using word2vec in Python, which can be found in the tutorial on Kaggle. But instead of learning features via word2vec and k-means clustering, we use the AFINN word list.

Instead of adding up the scores from the AFINN list, we use the scores as clusters. For each row, we say there are x_{-5} terms with the label of -5, x_{-4} terms with the label of -4, and so on.

```
# bag of scores
# scores range from -5 to 5
score.vectorizer <- function(sentence, words, word.scores)
{
  score.vector <- rep(0, length(table(word.scores)))
  k <- 0
  for (i in as.numeric(names(table(word.scores))))
  {
    k <- k + 1
    tempwords <- words[word.scores == i]
    score.vector[k] <- sum(str_count(sentence, tempwords))
  }
  return(score.vector)
}
score.data <- as.data.frame(t(apply(t(all.data$review.clean), 2,
                                    function(x) score.vectorizer(x,
                                                                 afinn$word.clean,
                                                                 afinn$score))))
# name columns
names(score.data) <- c('n5', 'n4', 'n3', 'n2', 'n1',
                       'zero',
                       'p1', 'p2', 'p3', 'p4', 'p5')
```

Taking a look at the data ...

```
  n5 n4 n3 n2 n1 zero p1 p2 p3 p4 p5
1  0  0 20 10 10    0  8 20  7  0  0
```

```
2  0  3  1  9  2    0  5 15  5  0  0
3  0  1 15 17 12    0  2 14  4  1  0
4  0  1  1 10  8    0  4 23  8  0  0
5  0  0  5 13 14    0  7 18  6  3  1
6  0  0  2  5  3    0  1 12  4  0  0
```

For now, we can just try fitting a simple random forest classifier to the data:

```
require(randomForest)
bag.of.scores.forest <- randomForest(score.data[train.labeled.ind, ],
                                     as.factor(train.labeled$sentiment))
bag.of.scores.forest.pred <- predict(bag.of.scores.forest,
                                     newdata = score.data[test.ind, ],
                                     type = 'prob')
results <- data.frame(id = all.data$id[test.ind],
                      sentiment = bag.of.scores.forest.pred[, 2])
results$id <- gsub('"', '', results$id)
write.table(results,
            file = '~/Documents/bag of words/bag_of_scores.csv',
            sep = ',', row.names = F, quote = F)
```

Unfortunately, this does not perform all that well, either (AUC $= .7858$). It seems that aggregating by AFINN score is not a good method.

Term Frequency–Inverse Document Frequency

A commonly used metric in scoring a phrase's importance in a sentence or document is the term frequency–inverse document frequency score, which compares the phrase's frequency in the document to the frequency of occurrence in a collection of documents.

The tf-idf score is calculated in two parts. The first part, which is the term frequency, is simply the frequency of a term in a document:

$$tf(t, d) = n(t|d)$$

The second part is the logarithm of the number of documents in the collection divided by the number of documents that contain the term:

$$idf(t, D) = \log \frac{|D|}{|\{d \in D | t \in d\}|}$$

The tf-idf score is simply the product of tf and idf:

$$tfidf(t, d, D) = tf(t, d) \times idf(t, d, D)$$

For example, if the word "popcorn" appears twice in a review, and 314 out of 75,000 reviews contain the word "popcorn", the tf-idf score of "popcorn" for that review would be $2 \times \log(75000/314) = 4.756$.

This is where the unlabeled training data comes into play. Typically, tf-idf gives us a better metric when the collection of documents is large.

The first thing we need to decide is which words we are going to use. Since we already have a tf matrix of AFINN words, we might as well start with that. But first, it's a good idea to convert it into a data frame.

```
tf <- as.data.frame(tf)
```

Next, we need our idf vector. For this, we're going to use both our labeled and unlabeled training data (we don't use the test data here).

```
idf <- log(length(train.ind)/colSums(sign(tf[train.ind, ])))
```

Finally, we obtain our tf-idf matrix:

```
tf.idf <- as.data.frame(t(apply(tf, 1, function(x) x * idf)))
```

Note that "absentees," an AFINN word, is not present in any of the reviews. This means the tf-idf score for "absentees" is undefined for all reviews. But since $lim_{x \to 0} x \log x = 0$, we can just overwrite the undefined values as 0. It's easier to manipulate the idf vector first before constructing the tf.idf data frame.

```
# remove infinite idf values since they go to zero when computing tf-idf
idf[is.infinite(idf)] <- 0

# compute tf-idf
tf.idf <- as.data.frame(t(apply(tf, 1, function(x) x * idf)))
```

Finally, we can fit a random forest classifier to the data.

```
# fit a classifier
tfidf.forest <- randomForest(tf.idf[train.labeled.ind, ],
                             as.factor(all.data$sentiment[train.labeled.ind]),
                             ntree = 100)

# compute predictions and write as csv
tfidf.forest.pred <- predict(tfidf.forest,
                             newdata = tf.idf[test.ind, ],
                             type = 'prob')
results <- data.frame(id = all.data$id[test.ind],
                      sentiment = tfidf.forest.pred[, 2])
results$id <- gsub('"', '', results$id)
write.table(results,
            file = '~/Documents/bag of words/tfidf_afinn.csv',
            sep = ',', row.names = F, quote = F)
```

We still don't get very good results (AUC $\approx$ .85, not to mention it takes forever to run). It appears that the AFINN word list does not give us good features. This is something we often see in simple NLP analyses–a priori feature extraction tends to give us poor predictors. The AFINN list is better suited for inferring sentiment from unlabeled data rather than as a part of a supervised algorithm. In the next part, we'll see how we can extract features using just the data themselves without any predefined labels.

Building a Term Frequency Matrix from the Corpus

In Kaggle's bag of words tutorial, we built predictors based on 5,000 of the most frequent words in the corpus of reviews. We can do something similar in R with the tm package.

```
require(tm)
```

We start by constructing a corpus, and from that we can construct a tf matrix of all the words in the corpus.

```
# construct corpus (using only training data)
corpus <- Corpus(VectorSource(all.data$review.clean[train.ind]))
```

```
# tf matrix using all words (minus stop words)
# list of stop words can be found at
# http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop
tf <- DocumentTermMatrix(corpus, control = list(stopwords = stopwords('english'),
                                                removeNumbers = T))
```

As you can imagine, even after removing stop words, we end up with a very large matrix (75000×213398). Many of these words occur very infrequently. We can remove sparse terms and cast as a dense matrix:

```
# only include words that occur in at least .1% of reviews
tf <- removeSparseTerms(tf, .999)
# convert to dense matrix for easier analysis
tf <- as.matrix(tf)
```

Now we have a more manageable 9,799 columns. Let's take a look at some of the words.

```
> head(colnames(tf))
[1] "aaron"     "abandon"   "abandoned" "abbott"    "abc"       "abducted"
```

Additionally, we can create a word frequency data frame:

```
# sum up the columns to find the occurrences of each word in the corpus
word.freq <- colSums(tf)
# put in nicer format
word.freq <- data.frame(word = names(word.freq), freq = word.freq)
rownames(word.freq) <- NULL
```

Let's take a look at the data:

```
> head(word.freq)
       word freq
1     aaron  145
2   abandon  146
3 abandoned  586
4    abbott  176
5       abc  204
6  abducted  118
```

Or if we want to look at the most frequent terms:

```
> head(word.freq[order(-word.freq$freq), ])
      word   freq
5779 movie 125307
3375  film 113054
6132   one  77447
5150  like  59147
4847  just  53132
3826  good  43279
```

The next approach aims to improve on the bag of words features by building features on words that occur more often in positive reviews than in negative reviews (or vice versa).

Normalized Difference Sentiment Index

The problem with using the N most frequent words as features is some words are very common regardless of sentiment. We expect the words ″movie″ and ″film″ to be common in both positive and negative reviews. One way to weed out such

words is by considering the difference of frequencies from positive and negative reviews. To begin, we construct a list of frequencies for both positive and negative reviews. Unfortunately, this means we can't use the 50,000 unlabeled training data.

```
word.freq <- function(document.vector, sparsity = .999)
{
  # construct corpus
  temp.corpus <- Corpus(VectorSource(document.vector))

  # construct tf matrix and remove sparse terms
  temp.tf <- DocumentTermMatrix(temp.corpus,
                                control = list(stopwords = stopwords('english'),
                                               removeNumbers = T))
  temp.tf <- removeSparseTerms(temp.tf, sparsity)
  temp.tf <- as.matrix(temp.tf)

  # construct word frequency df
  freq.df <- colSums(temp.tf)
  freq.df <- data.frame(word = names(freq.df), freq = freq.df)
  rownames(freq.df) <- NULL
  return(freq.df)
}

word.freq.pos <- word.freq(all.data$review.clean[all.data$sentiment == 1])
word.freq.neg <- word.freq(all.data$review.clean[all.data$sentiment == 0])
```

And let's take a look at the data:

```
> head(word.freq.pos)
        word freq
1    ability  214
2       able  717
3   absolute  153
4 absolutely  642
5    academy  190
6     accent  173

> head(word.freq.neg)
        word freq
1  abandoned   99
2    ability  231
3       able  523
4   absolute  196
5 absolutely  839
6     absurd  207
```

Next, we merge the two tables, do a bit of clean-up, and compute the difference in frequencies.

```
# merge by word
freq.all <- merge(word.freq.neg, word.freq.pos, by = 'word', all = T)

# clean up
freq.all$freq.x[is.na(freq.all$freq.x)] <- 0
freq.all$freq.y[is.na(freq.all$freq.y)] <- 0

# compute difference
freq.all$diff <- abs(freq.all$freq.x - freq.all$freq.y)
```

Now we can take a look at some of the more "important" words:

```
> head(freq.all[order(-freq.all$diff), ])
      word freq.x freq.y diff
1235 movie  23668  18139 5529
146    bad   7089   1830 5259
826  great   2601   6294 3693
1008  just  10535   7098 3437
604   even   7604   4899 2705
2115 worst   2436    246 2190
```

As expected, we see words like "great," "bad," or "worst," but we also see the word "movie" despite it not really carrying any sentimental meaning. This is probably because "movie" is so common any proportionally small difference is large. To remedy this, let's introduce a metric called the "normalized difference sentiment index," which takes this into account.

$$NDSI(t) = \frac{|n(t|0) - n(t|1)|}{n(t|0) + n(t|1)}$$

In other words, NDSI is the difference of frequencies normalized by their sum. NDSI values are between 0 and 1 with higher values indicating greater correlation with sentiment.

Before implementing this, note that some words that occur in positive reviews don't occur at all in negative reviews (or vice versa). In addition, we need to penalize infrequent words. If the word "steatopygous" occurs just once in a positive review and not at all in any of the negative reviews, we end up with a NDSI value of 1 even though we know it's not a great predictor of sentiment. To remedy this, we can add a smoothing term that penalizes infrequent words.

$$NDSI(t) = \frac{|n(t|0) - n(t|1)|}{n(t|0) + n(t|1) + 2\alpha}$$

Here, we set $\alpha = 128$.

```
# smoothing term
alpha <- 2**7

# ndsi
freq.all$ndsi <- abs(freq.all$freq.x -
                       freq.all$freq.y)/(freq.all$freq.x +
                                           freq.all$freq.y +
                                           2*alpha)
```

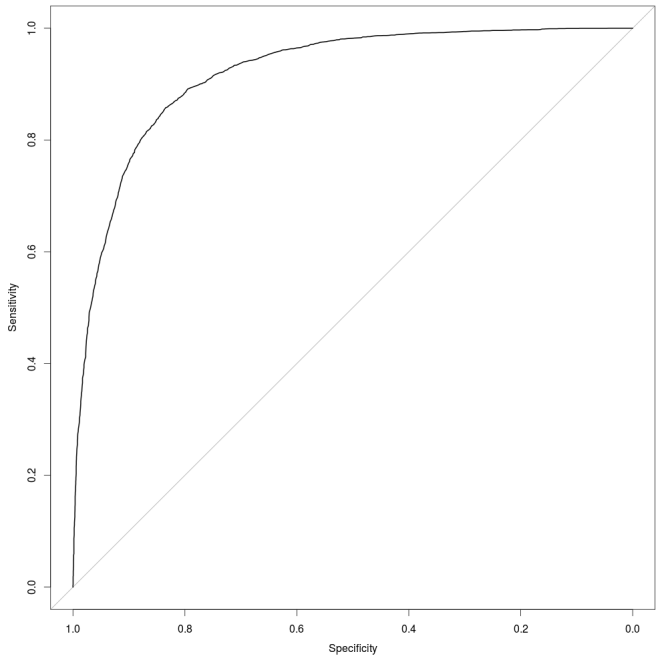
And we can take a look at which terms NDSI picked out as the best predictors:

```
> head(freq.all[order(-freq.all$ndsi), ])
       word freq.x freq.y diff      ndsi
2115  worst   2436    246 2190 0.7454050
2048  waste   1351     94 1257 0.7389771
1411 poorly    620      0  620 0.7077626
1040   lame    618      0  618 0.7070938
141   awful   1441    159 1282 0.6907328
1187   mess    498      0  498 0.6604775
```

Last but not least, we can use the N words with the highest NDSI values to build a tf-idf data frame, train a random forest classifier, and submit our results.

```
# number of words to consider in tf-idf matrix
num.features <- 2**10

# sort the frequency data
freq.all <- freq.all[order(-freq.all$ndsi), ]
```

```
# cast word to string
freq.all$word <- as.character(freq.all$word)

# build the tf matrix
tf <- t(apply(t(all.data$review.clean), 2,
              function(x) str_count(x, freq.all$word[1:num.features])))

# idf vector
idf <- log(length(train.ind)/colSums(sign(tf[train.ind, ])))
idf[is.infinite(idf)] <- 0

# tf-idf matrix
tf.idf <- as.data.frame(t(apply(tf, 1, function(x) x * idf)))
colnames(tf.idf) <- freq.all$word[1:num.features]

# train random forest classifier
ndsi.forest <- randomForest(tf.idf[train.labeled.ind, ],
                            as.factor(all.data$sentiment[train.labeled.ind]),
                            ntree = 100)

# predict and write output
ndsi.pred <- predict(ndsi.forest,
                     newdata = tf.idf[test.ind, ],
                     type = 'prob')
results <- data.frame(id = all.data$id[test.ind],
                      sentiment = bag.of.scores.forest.pred[, 2])
results$id <- gsub('"', '', results$id)
write.table(results,
            file = '~/Documents/bag of words/ndsi_tfidf.csv',
            sep = ',', row.names = F, quote = F)
```



ROC curve constructed on a fraction of the training data after having trained a model on another fraction, AUC = .9191

Submitting this result to Kaggle gives us an AUC of .91068, a huge improvement from our AFINN models using less than half the number of predictors!

Optimization

So far, we've been using inefficient methods and building the matricies ourselves (and waiting around quite a bit for our scripts to run), but thankfully there are some built-in methods that can help. The DocumentTermMatrix() function defaults to words present in the corpus, but if we want to use a custom list, we can simply restrict the columns of the tf matrix:

```
tf <- tf[, colnames(tf) %in% freq.all$word]
```

If we want to include n-grams in addition to single-word terms, the NLP package provides us with methods in doing so.

And finally, if we want to compute a tf-idf matrix with the tm package:

```
tf.idf <- DocumentTermMatrix(corpus,
                             control =
                               list(stopwords =
                                      stopwords('english'),
                                removeNumbers = T,
                                weighting = function(x) weightTfIdf(x,
                                                                    normalize = F)))
```

Conclusion

This is the end of this R demonstration, but there's still much to do. Adding additional predictors may improve our predictions at the cost of computational time (which is already expensive). This can be remedied by using additional packages and built-in functions. While tf-idf is a good metric for how important a word or phrase is to a document, it doesn't consider word meanings and similarities. For that, please refer to word2vec or similar models. Finally, none of these methods consider word order or context (e.g. "very awesome" and "not awesome" have opposite meanings despite sharing the word "awesome"). This can be accomplished by including n-grams in the term frequency matrix or by implementing doc2vec or similar models.

Links

1. "Bag of Words Meets Bags of Popcorn." https://www.kaggle.com/c/word2vec-nlp-tutorial
2. AFINN list. http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010
3. Wikipedia page on tf-idf. https://en.wikipedia.org/wiki/Tf%E2%80%93idf