PH125.9x - Predicting "Fake" News Using Machine Learning

Brendan Hawk

2020-06-01

Introduction

Overview

According to a cursory search on Google Trends[1], peak interest in the search term "Fake News" was reached in October of 2016. The same data reported from January of 2004 through September of 2016 shows a relative average interest of only 4.1%, while relative average interest since October of 2016 is 38%. While the concept of "real" vs "fake" news has certainly been as long-standing as the news itself, the buzz-word "Fake News" has brought it to new light in the public's eye.

As the sheer amount of information increases day by day, differentiating the truthful media available to us from stories that are poorly sourced, exaggerated, or even outright falsified is more important than ever. Unfortunately, it is also more difficult than ever; individuals and organisations crafting disinformation are doing so using targeting techniques backed by huge amounts of data and advanced machine learning. Propaganda on a given subject can be aimed at and delivered to individuals that are already on the cusp of believing it, relying on their confirmation bias to reinforce loosely held opinions into firm beliefs.

This same approach can be used to help distinguish such attacks from truer information. With the help of classifications from places like Politifact[2] and other non-partisan, object fact-checking sources, we can begin to train machine learning algorithms to help us sort the deluge of news and pull out what's worth keeping and what can be rejected.

The data used in this report comes from real-world news sources: Places like Reuters[3] website for a source of verified news, and places flagged by Politifact and Wikipedia as unreliable news sources for a source of "fake" news. The majority of these articles are focused on US and World Politics. This dataset was sourced from its original collectors, at University of Victoria ISOT Research Lab[4], where researches Ahmed, Traore, and Saad Published them in conference notes from the first ISDDC[1] conference in 2017 as well as a subsequent paper in the Journal of Security and Privacy[2].

Machine Learning Approaches

To accomplish the goal of identifying "fake" news, we first need to train our machine learning algorithms to read. Modern day approaches to this involve a computational marvel called Natural Language Processing, where that goal is more literally applied. Here, we have taken a more rudimentary approach and applied a technique called Sentiment Analysis. This technique involves breaking down parts of the text at hand into *tokens*, then associating each token with some metric of sentiment. For example, the token "foolish" is given a simple negative sentiment score of -2, or the token "sage" is attributed to multiple sentiments of "positive" and "trust". These associations are introduced to the data through *lexicons*: dictionaries of correlations between words in a given language and sentiments associated with them.

[1] https://trends.google.com/trends/explore?date=2004-01-01%202020-01-01&geo=US&q=fake%20news
[2] https://www.politifact.com/
[3] https://www.reuters.com
[4] https://www.uvic.ca/engineering/ece/isot/

Lexicons

Lexicons are the foundation of Sentiment Analysis, and are a great labour to produce. All three of the lexicons used in the following analysis were produced using some form of crowdsourcing, where respondents were asked to make these associations to given words using a provided scale or set of options. They each use a different form of association, and have all provided differing results when used in analysis.

Afinn The Afinn Sentiment Lexicon[3] presents a single scale of values to measure sentiment. This scale ranges from -5 to 5 and is rated in integers. This lexicon has associations for 2477 tokens.

NRC The NRC Word-Emotion Association Lexicon[4] presents each token with one or more associated emotions. This is a combination of the two sentiments "negative" and "positive", and eight emotions: "anger", "anticipation", "disgust", "fear", "joy", "sadness", "surprise", and "trust". This lexicon is considerably larger than the Afinn lexicon, with associations for 6468 tokens.

NRC VAD The NRC Valence, Arousal, and Dominance Lexicon[5] is a creative application of influential factor analysis and Best-Worst scoring to compute a score for each token on three different dimensions. The "valence" dimension is positive-negative or pleasure-displeasure scale The "arousal" dimension is a descriptor of the excited-calm or active-passive sentiment. The "dominance" dimensions shows the powerful-weak sentiment. Each dimensional score is provided a as decimal value ranging from 0 to 1. This makes it the most fine-grained lexicon. It is also the largest, with sentiment associations for all three dimensions associated with 20,007 tokens.

Analysis

The Data

The Fake News Dataset from Ahmed *et al* comes in the form of two csv files: Fake.csv and True.csv. To begin our analysis we can read these into a single dataset and assign a new column differentiating the rows:

```
# Read True.csv into memory
real.news <- fread(file.path("data", "True.csv"))
# Assign a new column denoting this is NOT "fake" news
real.news[, is_fake := FALSE]

# Read Fake.csv into memory
fake.news <- fread(file.path("data", "Fake.csv"))
# Assign a new column denoting this is "fake" news
fake.news[, is_fake := TRUE]

# Combine the tables
all.news <- rbind(real.news, fake.news)

# Remove intermediary objects
remove(real.news, fake.news)
```

Immediately on perusing the dataset, we can see that the sources have some shoddy character encoding. Reading down the list of titles, there are noticeable issues such as apostrophes mis-encoded as `â€`, and a few examples of a special form of double quote shown as `"`. It is easiest to deal with this up front, and re-encode everything into the same standard. We can also trim out leading and trailing whitespace from our titles and texts at the same time.

```
all.news[, `:=`(
    title = str_trim(iconv(title, from = "utf8", to = "latin1")),
    text = str_trim(iconv(text, from = "utf8", to = "latin1"))
)]
```

We should also inspect for missing data.

```
# Empty title cells
all.news[is.na(title) | title == "", .N]
```

```
## [1] 9
```

```
# Empty text cells
all.news[is.na(text) | text == "", .N]
```

```
## [1] 651
```

```
# Empty text cells aggregated by is_fake
all.news[
  is.na(text) | text == ""
][
  ,
  .N,
  by = is_fake
]
```

```
##    is_fake   N
## 1:   FALSE  21
## 2:    TRUE 630
```

We can see that there are missing data. 9 Titles are empty, as well as 630 "fake" and 21 "real" news texts. In the final analysis, we combine these two into a single text column to parse, so as long as we at least have a title we can continue with a given observation.

```
all.news <- all.news[!is.na(title) & title != ""]
```

Exploration

Knowing the source of the data, I was aware that there were some built-in biases. The most glaring of these is easily seen when scanning through texts from the True.csv file: almost all of them mention or lead in with the word "Reuters".

```
# Create a column showing whether or not the word "Reuters" is in the text
all.news[, has_reuters := grepl("reuters", text, ignore.case = TRUE)]

# Summarize this column disaggregated by is_fake
all.news[, .N, by = list(has_reuters, is_fake)]
```

```
##    has_reuters is_fake     N
## 1:        TRUE   FALSE 21357
## 2:       FALSE   FALSE    59
## 3:       FALSE    TRUE 23151
## 4:        TRUE    TRUE   322
```

This phenomenon is so prevalent, in fact, that if this were the foundation of our approach, we would already be finished with our analysis. If we simply guessed "If the article does *not* mention Reuters, then it must be fake news." we can achieve an unreasonable accuracy.

```
reuters.guess <- data.table(
    prediction = as.factor(!all.news$has_reuters),
    observation = as.factor(all.news$is_fake)
)

table(reuters.guess$prediction, reuters.guess$observation)
```

```
##
##         FALSE  TRUE
##   FALSE 21357   322
##   TRUE     59 23151
```

This accuracy being greater than 99% is an obvious pitfall, however none of the lexicons used in the sentiment analysis below contain the word "Reuters" in any form. This will naturally preclude this from confounding our results later.

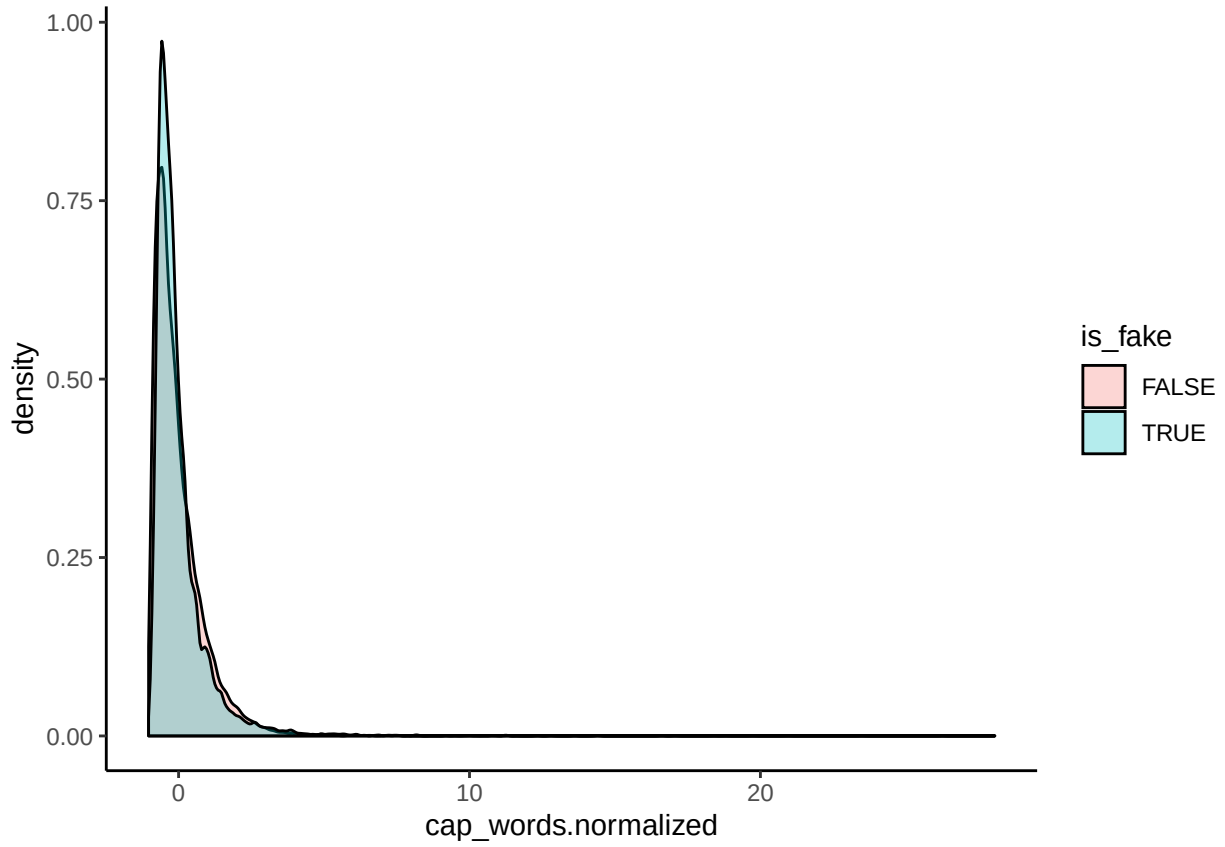
Further scanning of the data reveals that some articles have a preponderance of capitalized words, while others use only proper casing. We can count the number of whole words that are entirely capitalized, and visualize this on a plot. We will normalize the values within the realms of fake and true news, so that we can see if there is a differing correlation.

```r
# Join the title and text of each observation into a column called full_text
all.news[, full_text := paste(title, text, sep = " ")]

# Count occurrences of whole words that are capitalized in each full_text
all.news[, cap_words := str_count(full_text, "[^a-z][A-Z]+[^a-z]")]

# normalize both sets of data separately, then plot
all.news[
  is_fake == TRUE,
  cap_words.normalized := (cap_words - mean(cap_words))/sd(cap_words)
][
  is_fake == FALSE,
  cap_words.normalized := (cap_words - mean(cap_words))/sd(cap_words)
]

# plot
all.news %>%
  ggplot(aes(x = cap_words.normalized, fill = is_fake)) +
  geom_density(alpha = 0.3)
```

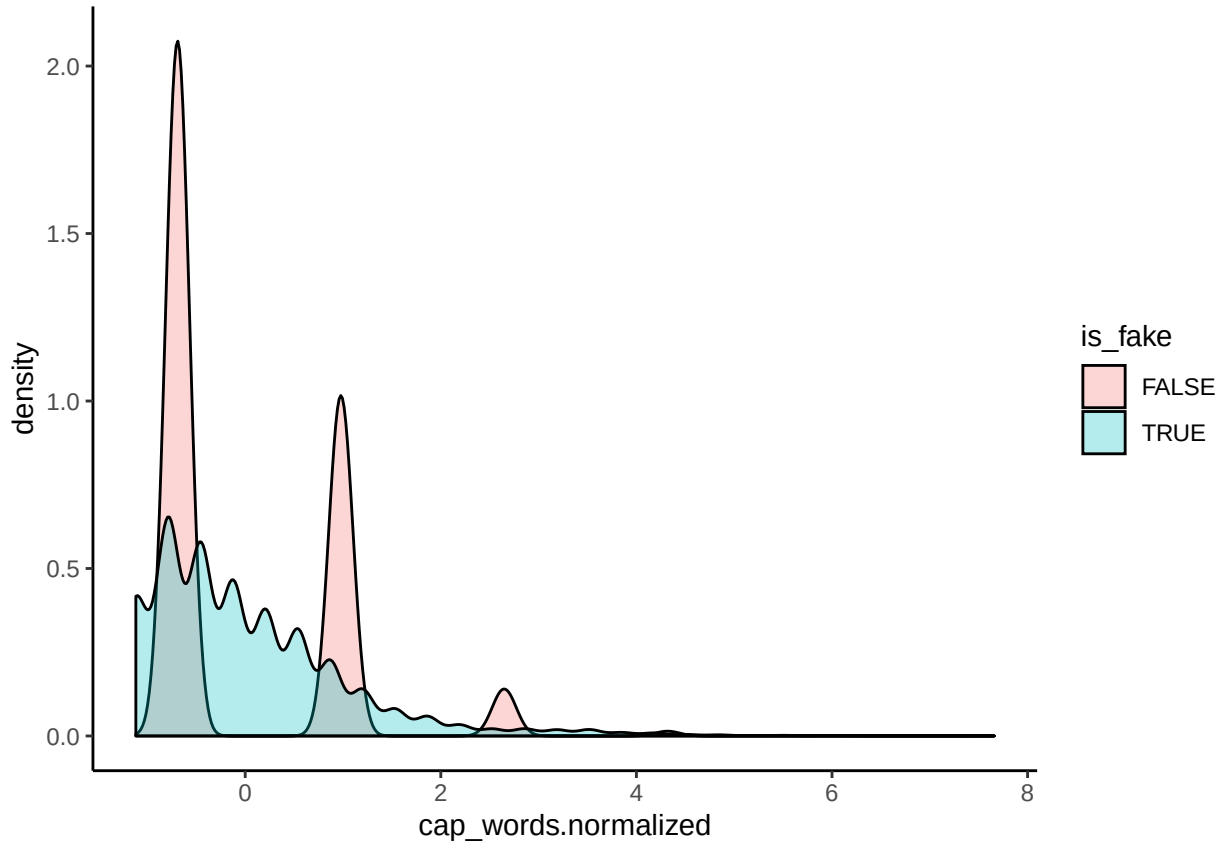


There certainly seems to be a noticeable difference in magnitude between these two, however their shape is incredibly similar. We can also perform the same exploration using just the titles.

```r
# Count occurrences of whole words that are capitalized in just the title
all.news[, cap_words := str_count(title, "[^a-z][A-Z]+[^a-z]")]
```

```r
# normalize both sets of data separately, then plot
all.news[
  is_fake == TRUE,
  cap_words.normalized := (cap_words - mean(cap_words))/sd(cap_words)
][
  is_fake == FALSE,
  cap_words.normalized := (cap_words - mean(cap_words))/sd(cap_words)
]

# plot
all.news %>%
  ggplot(aes(x = cap_words.normalized, fill = is_fake)) +
  geom_density(alpha = 0.3)
```



Perhaps due to titles being fewer words by a large margin, we end up with some strange divergent density plots. We will hold on to this idea, and use a count of capitalized words in the title as one of our predictors.

These data sources also came with a column of subjects. However it is clearly evident that there is another built-in bias here; the subjects found in the `Fake.csv` data are not found in the `True.csv` data and vice versa.

```
# Viewing all Subjects
all.news[, .N, by = subject]
```

```
##            subject     N
## 1:    politicsNews 11272
## 2:       worldnews 10144
## 3:            News  9049
## 4:        politics  6837
## 5: Government News  1569
## 6:       left-news  4457
## 7:         US_News   783
## 8:     Middle-east   778
```

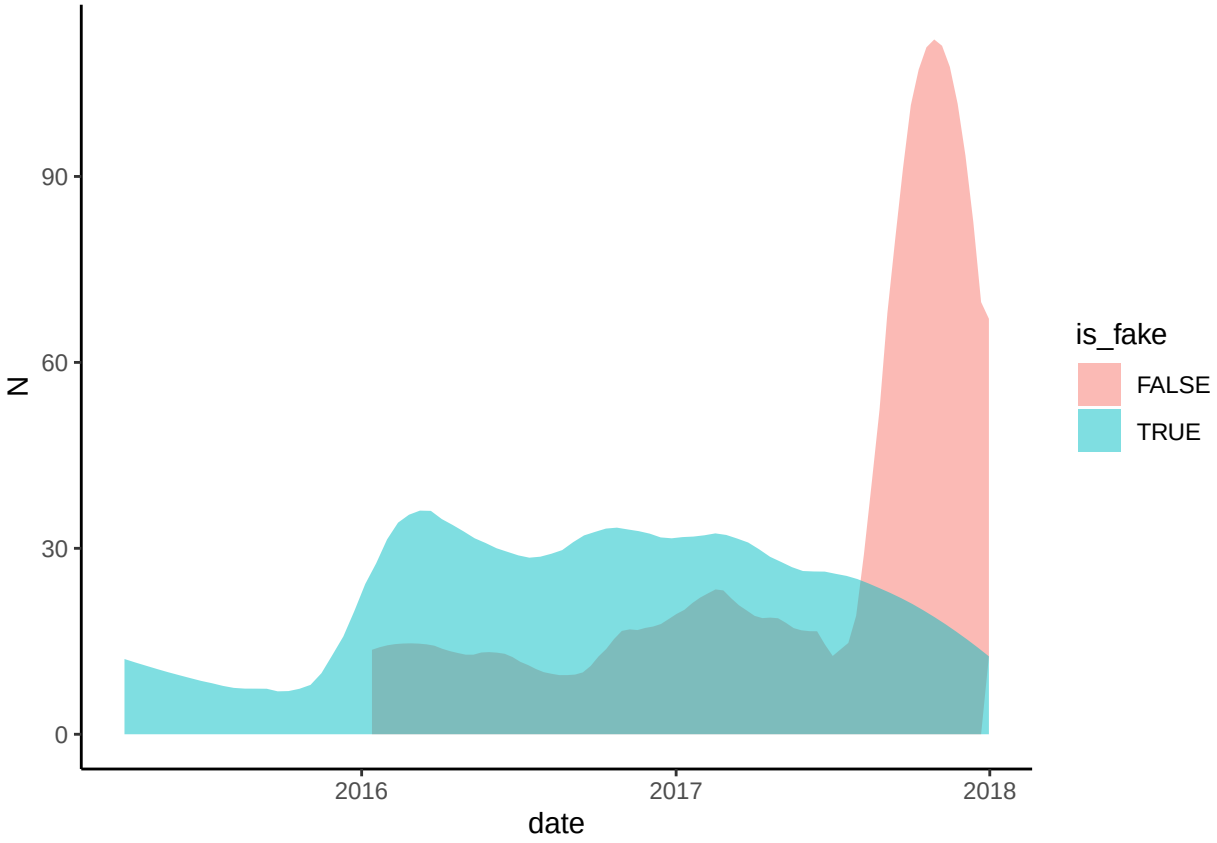
```
# Viewing subjects disaggregated by is_fake
all.news[, .N, by = list(subject, is_fake)]
```

```
##            subject is_fake     N
## 1:    politicsNews   FALSE 11272
## 2:       worldnews   FALSE 10144
## 3:            News    TRUE  9049
## 4:        politics    TRUE  6837
## 5: Government News    TRUE  1569
## 6:       left-news    TRUE  4457
## 7:         US_News    TRUE   783
## 8:     Middle-east    TRUE   778
```

Lastly, each observation comes with a date stamp of some form. The majority of this data is from 2016-2018, but as we will see the spread of dates is not even. This step produces a warning about several dates failing to parse. Inspecting these failures (not shown here) reveals that 35 observations use a different date format, and 10 observations have URLs listed in the date column. If we are to use this data later, we would have to consistently impute these. Many of the URLs have dates in them, and could be corrected by hand, however after this point in exploration the date column is not leveraged in the following analysis.

```r
# Parse the date in an R date object
all.news[, parsed_date := mdy(date)]

# plot counts over time
all.news[, .N, by = .(is_fake, date = date(parsed_date))] %>%
  ggplot(aes(x = date, y = N, fill = is_fake)) +
  stat_smooth(
    geom = 'area',
    span = 0.25,
    alpha = 0.5,
    method = 'loess',
    position = "stack"
  )
```



Approaches and Models

Data Preparation To begin our analysis, we will need to clean the data, tokenize the texts, bind the tokens to their respective sentiments in each of the three lexicons, and finally accumulate the resulting sentiments for each article.

Tokenization for this dataset will occur effectively per word. Commonly, more in-depth analysis utilize tokenization by `ngrams` or sets of n words. Commonly this takes the shape of bigrams ie pairs of words, allowing a difference to be detected between tokens such as "successful" and a matching but inverse sentiment from the bigram "not successful".

Another important cleaning step is applied to the tokenized values: removal of stop words. Stop words, such as "and", "is", and "so". These words do not add meaningful sentiment to analyse and are thusly removed.

It is also important to note that merging tokens with a sentiment set is done with `inner_join`, which inherently will reduce the size of the tokenized data list to only rows where there is a matching sentiment for that token in the lexicon. It is for this reason that the size of a lexicon is an important factor when choosing one for an analysis.

We will also split the data into a training set to build our models with, and a test set to verify the models' final accuracy.

```
# Clean and tidy dataset
all.news <- lazy_dt(all.news) %>%
  mutate(
    title = str_trim(iconv(title, from = "utf8", to = "latin1")),
    text = str_trim(iconv(text, from = "utf8", to = "latin1"))
  ) %>%
  filter(!is.na(title) & title != '') %>%
  mutate(
    full_text = paste(text, title),
    is_fake = as.factor(is_fake),
    title_caps = str_count(title, "[^a-z][A-Z]+[^a-z]")
  ) %>%
  select(title, full_text, is_fake, title_caps) %>%
  as.data.table()

# Mark training and test datasets.
train.index <- createDataPartition(
  all.news$is_fake,
  p = 0.8,
  times = 1,
  list = FALSE
)

all.news$set <- "testing"
all.news[train.index, set := "training"]

remove(train.index)

# split tokens for joining sentiment, remove stop words.
# This takes a few moments
tokenized <- all.news %>%
  unnest_tokens(token, full_text) %>%
  lazy_dt() %>%
```

```
  anti_join(data.table(token = stop_words$word), by = "token") %>%
  as.data.table()
```

Now we will join each of the three lexicons to these tokens separately. Each lexicon provides a different measure of sentiment, and so will need differing aggregation.

The Affin lexicon provides an integer column, and can simply be summed.

```
# Read the data from file
afinn <- fread("./data/afinn.csv")

# Change names of columns for joining
setnames(afinn, c("token", "sentiment"))

# setting data.table keys make joins a lot faster
setkey(afinn, token)
setkey(tokenized, token)

# data.table syntax for doing an inner join on keyed data.tables
afinn <- afinn[tokenized, nomatch = NULL]

# aggregate total sentiment for each article
afinn <- afinn[
  ,
  list(sentiment = sum(sentiment)),
  by  = list(title, is_fake, title_caps, set)
]
```

The NRC lexicon is far more work to aggregate. First, we join the tokens to the dataset, which filters the data to only those words that exist in the NRC lexicon. This same join, however, multiplies many of the rows by however many sentiments are attached to that token. When we aggregate them, we first need to make an intermediary process to spread these multiple rows into columns, showing whether or not a given token has any of the attached sentiments. Then, we can aggregate these into per-article totals.

```
nrc <- fread("./data/nrc.csv")

# Change column names for joining
setnames(nrc, "word", "token")

# Set keys for data.table join
setkey(nrc, token)
setkey(tokenized, token)

# data.table syntax for doing an inner join on keyed data.tables
nrc <- nrc[tokenized, nomatch = NULL]

# Tibbles with pivot_wider is a much easier-to-read approach here,
# but there are other more performant ways of doing this if our
# dataset was very large. See `reshape2` package
nrc <- as_tibble(nrc) %>%
  pivot_wider(
    names_from = sentiment,
    values_from = sentiment,
```

```
    values_fn = list(sentiment = length),
    values_fill = list(sentiment = 0)
  )

# View the wide output of this table
head(nrc, 3)
```

```
## # A tibble: 3 x 15
##   token title is_fake title_caps set   trust  fear negative sadness anger
##   <chr> <chr> <fct>        <int> <chr> <int> <int>    <int>   <int> <int>
## 1 abac... Cana... FALSE            0 trai...     1     0        0       0     0
## 2 aban... Spy ... FALSE            0 test...     0     1        1       1     0
## 3 aban... Schu... FALSE            0 trai...     0     1        1       1     0
## # ... with 5 more variables: surprise <int>, positive <int>, disgust <int>,
## #   anticipation <int>, joy <int>
```

```
# Roll up counts of all sentiments for all tokens for each article, ie Anger: 5, Joy 0, Negative: 3
nrc <- nrc %>%
  group_by(title, is_fake, title_caps, set) %>%
  summarize_at(vars(-token), list(sum)) %>%
  as.data.table()

head(nrc, 3)
```

```
##                                                                    title
## 1:     '#1 In Bigotry': Twitter EVISCERATES Mississippi Gov. Over Anti-Gay Law
## 2: '60 MINUTES': 9/11 REPORT Could Incriminate Saudi Arabia With These Details
## 3:          'A better future' - Britain's May tries to rally her Conservatives
##    is_fake title_caps      set trust fear negative sadness anger surprise
## 1:    TRUE          1 training    21   15       17      13    14        2
## 2:    TRUE          2 training    12    9       10       6     6        3
## 3:   FALSE          1 training    23   10       20      10    11        4
##    positive disgust anticipation joy
## 1:       18      11           10   7
## 2:       16       1           10   6
## 3:       38       5           10  11
```

Finally, the NRC VAD lexicon is aggregated like the Afinn lexicon, but with a column aggregated for each dimension.

```
# Read the NRC VAD data from file
vad <- fread("./data/nrc_vad.csv")

# This lexicon comes with Title-cased columns
setnames(vad, tolower(names(vad)))

# change this column name for joining
setnames(vad, "word", "token")

setkey(vad, token)
setkey(tokenized, token)
```

```r
# data.table syntax for an inner join on keyed data.tables
vad <- vad[tokenized, nomatch = NULL]

# Aggregate dimensions by summing across articles
vad <- vad[
  ,
  list(
    valence = sum(valence),
    arousal = sum(arousal),
    dominance = sum(dominance)
  ),
  by = list(title, is_fake, title_caps, set)
]
```

We can now separate the test and training sets. Because the above data preparation relies so heavily on merges, the data sets were kept together for easier processing. Here we will separate them and remove the training/testing identifier.

```r
afinn <- split(afinn, by = "set", keep.by = FALSE)
afinn.training <- afinn$training
afinn.testing <- afinn$testing

nrc <- split(nrc, by = "set", keep.by = FALSE)
nrc.training <- nrc$training
nrc.testing <- nrc$testing

vad <- split(vad, by = "set", keep.by = FALSE)
vad.training <- vad$training
vad.testing <- vad$testing

# clean up unneded objects
remove(tokenized, afinn, nrc, vad)
```

Naive Baseline It is always instructive to use the most naive approach as a baseline. Here, we can summarize our intended inference by effectively tossing a coin and guessing accordingly. Given an approximately normal distribution of binary categorical outcomes in the observed data, we can predict that this would be correct approximately half of the time. As we see here, this holds true and we achieve exactly 50% accuracy.

```r
# Replicate the naive approach 10000 times
mean(replicate(10000, {
  # Guess True or False for is_fake randomly
  predictions <- sample(c(TRUE, FALSE), nrow(all.news), replace = TRUE)

  # Return the accuracy of this replication
  mean(predictions == all.news$is_fake)
}))
```

```
## [1] 0.5
```

Random Forests Given the nature of our predictors being numeric and continuous, decision trees are a very common approach to classification algorithms. We will create and train a model for each set of data

bound to the three sentiment lexicons. Here, this analysis is accomplished more quickly using `parRF`, a parallelized implementation of Random Forests. The `caret` training method for this package only has one tuning parameter, `mtry`, and for data this small I have found the defaults work well enough.

I have implemented this here on a relatively powerful consumer PC, with a roughly 4Ghz processor running 16 threads.

```
# We can leveraging matrix-based function signatures for all the models we build
# This helper function will create a matrix of all predictors from a given data.table
makePredictors <- function(dt) {
  # drop our title, used only as an identifier column
  dt$title = NULL

  # drop the response column
  dt$is_fake = NULL

  # return the rest of the columns as a matrix
  as.matrix(dt)
}

# Start and register parallel threads
# NB: Never set this number higher than your computer
# can handle!
nThreads <- 16
cl <- makeSOCKcluster(nThreads)
registerDoSNOW(cl)

# A 'caret' trainControl object, using parallized
# 5-fold cross-validation.
rf.trainControl <- trainControl(
  method = "cv",
  number = 5,
  allowParallel = TRUE
)

# RF model for Afinn sentiments
afinn.rf.model <- train(
  makePredictors(afinn.training),
  afinn.training$is_fake,
  method = "parRF",
  trControl = rf.trainControl
)
```

```
## note: only 1 unique complexity parameters in default grid. Truncating the grid to 1 .
```

```
# RF NRC
nrc.rf.model <- train(
  makePredictors(nrc.training),
  nrc.training$is_fake,
  method = "parRF",
  trControl = rf.trainControl
)

# RF VAD
```

```
vad.rf.model <- train(
  makePredictors(vad.training),
  vad.training$is_fake,
  method = "parRF",
  trControl = rf.trainControl
)

# Stop and de-register parallel computing
stopCluster(cl)
registerDoSEQ()
remove(cl)
```

Now that we've built our models, we can begin to measure their accuracy with the training data used to produce them. Obviously this will be overfit, however it does begin to show us whether or not these attempts were in anywhere near successful.

```
## Accuracy measures against training datasets
# Afinn RF
confusionMatrix(fitted(afinn.rf.model), afinn.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 14847  4057
##      TRUE   1523 10859
##
##                Accuracy : 0.822
##                  95% CI : (0.817, 0.826)
##     No Information Rate : 0.523
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.64
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.907
##             Specificity : 0.728
##          Pos Pred Value : 0.785
##          Neg Pred Value : 0.877
##              Prevalence : 0.523
##          Detection Rate : 0.475
##    Detection Prevalence : 0.604
##       Balanced Accuracy : 0.817
##
##        'Positive' Class : FALSE
##
```

```
# 82.2%

# NRC RF
confusionMatrix(fitted(nrc.rf.model), nrc.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 16724    81
##      TRUE     12 15039
##
##                Accuracy : 0.997
##                  95% CI : (0.996, 0.998)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.994
##
##  Mcnemar's Test P-Value : 1.77e-12
##
##             Sensitivity : 0.999
##             Specificity : 0.995
##          Pos Pred Value : 0.995
##          Neg Pred Value : 0.999
##              Prevalence : 0.525
##          Detection Rate : 0.525
##    Detection Prevalence : 0.528
##       Balanced Accuracy : 0.997
##
##        'Positive' Class : FALSE
##
```

```
# 99.7%

# NRC VAD RF
confusionMatrix(fitted(vad.rf.model), vad.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 16726    35
##      TRUE     17 15133
##
##                Accuracy : 0.998
##                  95% CI : (0.998, 0.999)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.997
##
##  Mcnemar's Test P-Value : 0.0184
##
##             Sensitivity : 0.999
##             Specificity : 0.998
##          Pos Pred Value : 0.998
##          Neg Pred Value : 0.999
##              Prevalence : 0.525
```

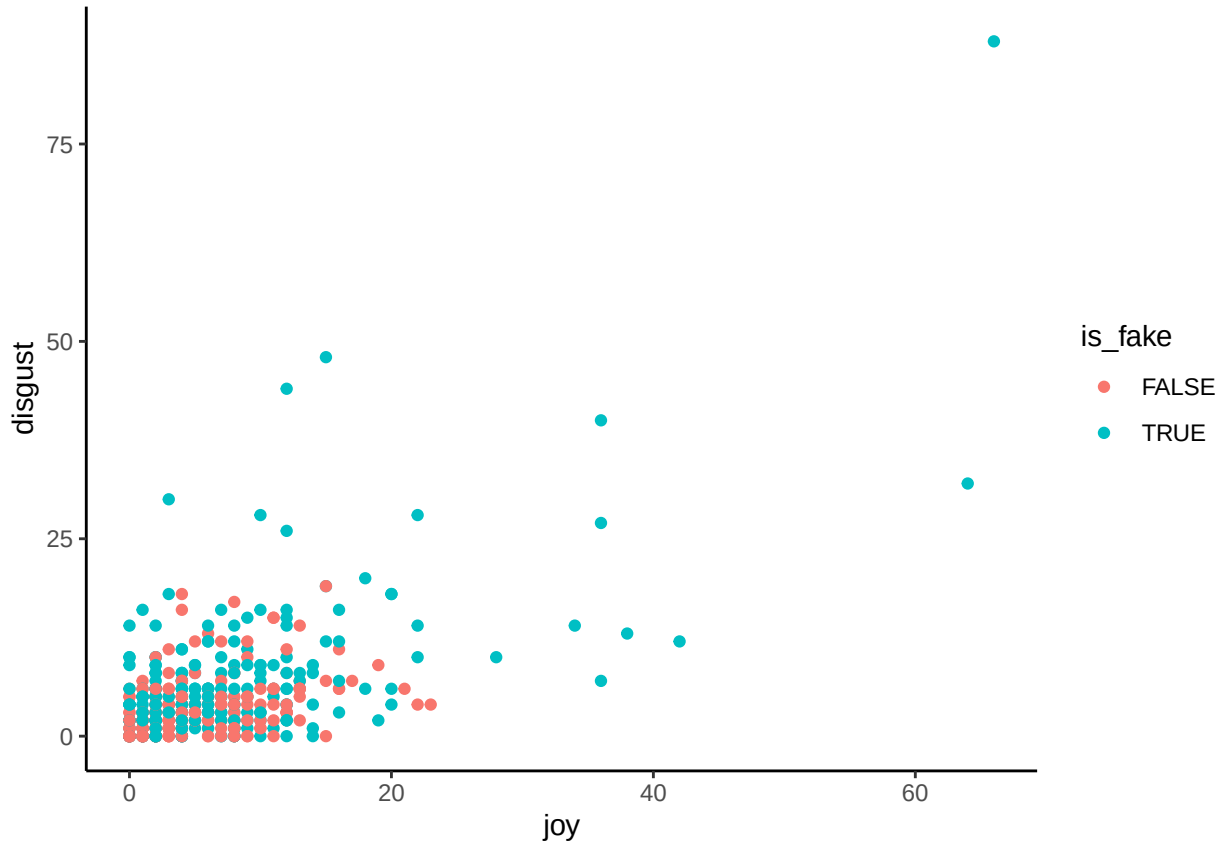
```
##          Detection Rate : 0.524
##    Detection Prevalence : 0.525
##       Balanced Accuracy : 0.998
##
##        'Positive' Class : FALSE
##
```

```
# 99.8%
```

Support Vector Machines In essence, Support Vector Machines allow a model to mathematically draw boundaries around groups of data within higher-order planes. The `ksvm` function used here and in the analysis at large is a type of SVM that leverages something called the *kernel trick*: an ingenious application of mathematical assumptions that allows SVMs to function well in higher dimensions. This, however, is not a thesis on the mathematics of SVMs, so here I will demonstrate visually how they are able to separate clusters of related data.

```
# Grab a very small chunk of data to demonstrate with
# 500 rows, with only "joy" and "disgust" dimensions as predictors
data.small <- nrc.training[
  sample.int(nrow(nrc.training), 500),
  list(is_fake, joy, disgust)
]

# Scatterplot of the predictors, color coded by outcome
data.small %>%
  ggplot(aes(x = joy, y = disgust, color = is_fake)) +
  geom_point()
```



As we can see, there does seem to be a general visual trend that one color is more prevalent in the upper-right of the plot, while the cluster in the lower left seems to be more focused on the other color. Now we can train a model on this data.

```
# Grab a matrix of predictors and a response vector
predictors <- as.matrix(data.small[, .(joy, disgust)])
response <- data.small$is_fake

# Train a basic KSVM model
ksvm.model <- ksvm(
  x = predictors,
  y = response
)

# Make predictions and view accuracy
ksvm.predictions <- predict(
  ksvm.model,
  as.matrix(nrc.training[, .(joy, disgust)])
)
confusionMatrix(ksvm.predictions, nrc.training$is_fake)
```

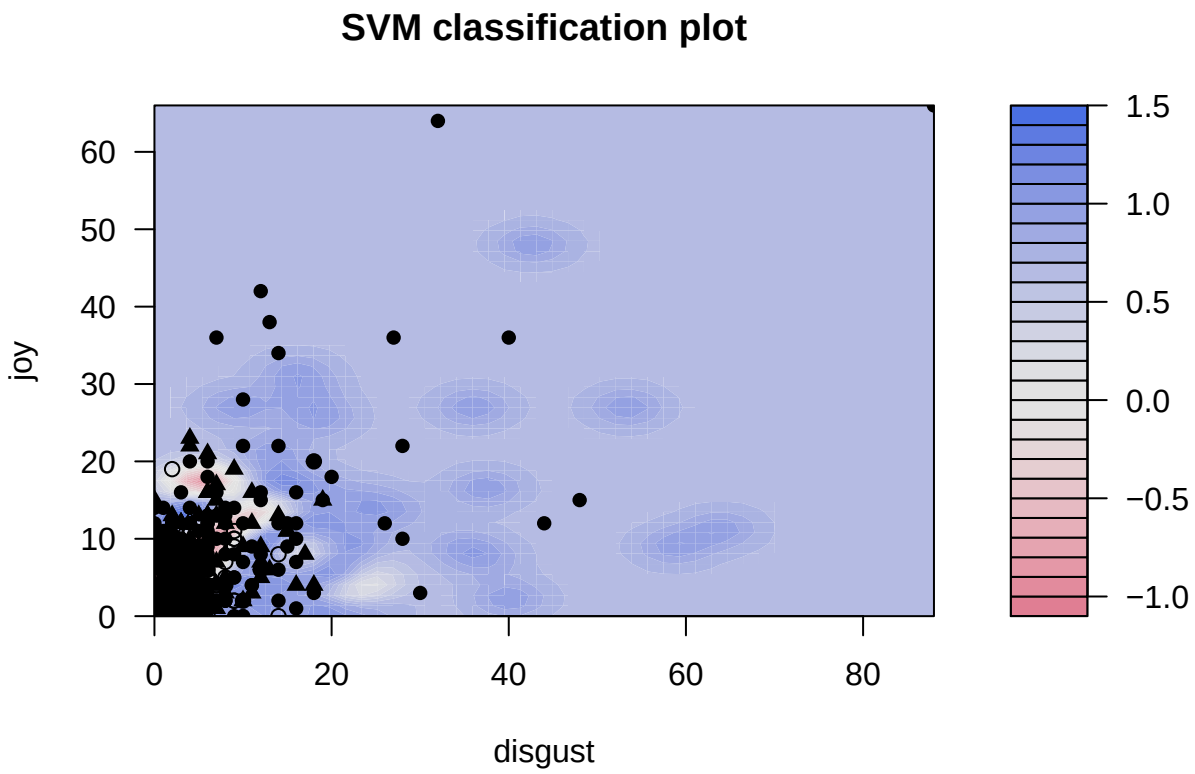
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 12219  7480
```

```
##      TRUE   4517  7640
##
##                Accuracy : 0.623
##                  95% CI : (0.618, 0.629)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.238
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.730
##             Specificity : 0.505
##          Pos Pred Value : 0.620
##          Neg Pred Value : 0.628
##              Prevalence : 0.525
##          Detection Rate : 0.384
##    Detection Prevalence : 0.618
##       Balanced Accuracy : 0.618
##
##        'Positive' Class : FALSE
##
```

```r
# 62.3%
```

With this little data, and only two dimensions, it is unsurprising that the accuracy of this model is not great. We can show the boundaries drawn by this model in a similar scatterplot.

```r
# Visualize the Boundaries our KSVM model has created.
plot(ksvm.model, data = predictors)
```



While the model clearly has a lot of learning still to do, we can see that generally the boundaries drawn make sense. In any machine learning problem it is the befuddled area in the middle that is always hardest to predict, so with more data, more dimensions, and a tuned algorithm we should be able to achieve much better results.

Now we are ready to begin training these KSVM models. The one parameter we need to train is `sigma`, which controls how linear or flexible the decision boundary becomes. The other tuning parameter `C` is a cost parameter, used to penalize large residuals after normalization. Due to the relatively small size of our data, I have chosen to leave this as the default of 1.

The tuning values for `sigma` will come from a helper function in the `kernlab` package called `sigest`, that estimates the range of these values based on a given fraction of the training data. I have chosen to expand this range by 25% on either side, and cover a very large number of possible parameters. This step is done for each model, as each model has it's own lexicon-bound training data to contend with.

```
# Create and register threads for parallel computing
cl <- makeSOCKcluster(nThreads)
registerDoSNOW(cl)

# Create a trainControl object for all KSVM models
ksvm.trainControl <- trainControl(
  method = "cv",
  number = 5,
  allowParallel = TRUE
)

# Create the matrix of predictors
```

```
afinn.training.predictors <- makePredictors(afinn.training)

# Here, and below, sigmas are going to come from an estimation function provided
# in the 'kernlab' package. The range of these will be expanded by 25% so that
# we can try a broader set of values for sigma.
afinn.training.sigmas <- sigest(afinn.training.predictors, frac = 1)
afinn.training.sigmas <- seq(
  afinn.training.sigmas["90%"] * 0.75,
  afinn.training.sigmas["10%"] * 1.25,
  length.out = 10
)

# train the model
afinn.ksvm.model <- train(
  afinn.training.predictors,
  afinn.training$is_fake,
  method = 'svmRadial',
  trControl = ksvm.trainControl,
  tuneGrid = data.table(
    sigma = afinn.training.sigmas,
    C = 1
  )
)

# Create the matrix of predictors
nrc.training.predictors <- makePredictors(nrc.training)

# Create set of values for tuning sigma
nrc.training.sigmas <- sigest(nrc.training.predictors, frac = 1)
nrc.training.sigmas <- seq(
  nrc.training.sigmas["90%"] * 0.75,
  nrc.training.sigmas["10%"] * 1.25,
  length.out = 10
)

# Train the model
nrc.ksvm.model <- train(
  nrc.training.predictors,
  nrc.training$is_fake,
  method = 'svmRadial',
  trControl = ksvm.trainControl,
  tuneGrid = data.table(
    sigma = nrc.training.sigmas,
    C = 1
  )
)

# Create the matrix of predictors
vad.training.predictors <- makePredictors(vad.training)

# Create the set of values for tuning sigma
vad.training.sigmas <- sigest(vad.training.predictors, frac = 1)
vad.training.sigmas <- seq(
```

```
  vad.training.sigmas["90%"] * 0.75,
  vad.training.sigmas["10%"] * 1.25,
  length.out = 10
)

# Train the model
vad.ksvm.model <- train(
  vad.training.predictors,
  vad.training$is_fake,
  method = 'svmRadial',
  trControl = ksvm.trainControl,
  tuneGrid = data.table(
    sigma = vad.training.sigmas,
    C = 1
  )
)

# Stop and de-register parallel computing
stopCluster(cl)
registerDoSEQ()
remove(cl)
```

Now with our models built, we can check the accuracy achieved with the training set.

```
# Afinn SVM
confusionMatrix(fitted(afinn.ksvm.model$finalModel), afinn.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 14804  4107
##      TRUE   1566 10809
##
##                Accuracy : 0.819
##                  95% CI : (0.814, 0.823)
##     No Information Rate : 0.523
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.634
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.904
##             Specificity : 0.725
##          Pos Pred Value : 0.783
##          Neg Pred Value : 0.873
##              Prevalence : 0.523
##          Detection Rate : 0.473
##    Detection Prevalence : 0.604
##       Balanced Accuracy : 0.814
##
##        'Positive' Class : FALSE
```

```
##
```

```
# 81.9%

# NRC SVM
confusionMatrix(fitted(nrc.ksvm.model$finalModel), nrc.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 15830  2594
##      TRUE    906 12526
##
##                Accuracy : 0.89
##                  95% CI : (0.887, 0.894)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.779
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.946
##             Specificity : 0.828
##          Pos Pred Value : 0.859
##          Neg Pred Value : 0.933
##              Prevalence : 0.525
##          Detection Rate : 0.497
##    Detection Prevalence : 0.578
##       Balanced Accuracy : 0.887
##
##        'Positive' Class : FALSE
##
```

```
# 89%

# NRC VAD SVM
confusionMatrix(fitted(vad.ksvm.model$finalModel), vad.training$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 15542  3228
##      TRUE   1201 11940
##
##                Accuracy : 0.861
##                  95% CI : (0.857, 0.865)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.72
```

```
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.928
##             Specificity : 0.787
##          Pos Pred Value : 0.828
##          Neg Pred Value : 0.909
##              Prevalence : 0.525
##          Detection Rate : 0.487
##    Detection Prevalence : 0.588
##       Balanced Accuracy : 0.858
##
##        'Positive' Class : FALSE
##
```

```
# 86.1%
```

These accuracies are not as good as the RF models above, which truthfully surprised me. That said, it's possible that these accuracies are more consistent and will hold out better when using the testing set later.

Ensemble Model There are some limitations to building an ensemble model, specifically with this dataset. Due to the relatively small size of the dataset, and the nature of how limiting lexicons can be, not every model built above can make predictions for every article in the original data. If an article, once broken into tokens, contains no matching rows to join to a given lexicon, eg Afinn, then that article is inherently excluded from the dataset due to the joining process required to build the models. An ensemble model will have to account for there being sparse data - ie given 6 predicted outcomes for every article in the original data some of those outcomes will be `NA`.

To accommodate this, and to try to make our ensemble model more accurate, we will begin by grabbing all the unique articles in our training data along with its respective `is_fake` flag. Then we can make the predictions and use left joining to create a matrix of predicted outcomes. Here we will use a "voting" scheme - given 6 predictions (some of which may be `NA`) we will simply take whichever has the most votes. In the case of a tie, we can simply guess as we did for our naive approach.

```
# Create a container for our models, gathering all articles in the training set
ensemble <- rbind(
  afinn.training[, list(title, is_fake)],
  nrc.training[, list(title, is_fake)],
  vad.training[, list(title, is_fake)]
)

# Take only the unique articles
ensemble <- unique(ensemble)

# Create data.tables from the training data with a column for their respective
# predictions
afinn.rf <- cbind(
  afinn.training,
  afinn.rf = predict(afinn.rf.model, makePredictors(afinn.training))
)
nrc.rf <- cbind(
  nrc.training,
  nrc.rf = predict(nrc.rf.model, makePredictors(nrc.training))
```

```
)
vad.rf <- cbind(
  vad.training,
  vad.rf = predict(vad.rf.model, makePredictors(vad.training))
)
afinn.ksvm <- cbind(
  afinn.training,
  afinn.ksvm = predict(afinn.ksvm.model, makePredictors(afinn.training))
)
nrc.ksvm <- cbind(
  nrc.training,
  nrc.ksvm = predict(nrc.ksvm.model, makePredictors(nrc.training))
)
vad.ksvm <- cbind(
  vad.training,
  vad.ksvm = predict(vad.ksvm.model, makePredictors(vad.training))
)

# Remove columns not needed for this step
afinn.rf <- afinn.rf[, list(title, afinn.rf)]
nrc.rf <- nrc.rf[, list(title, nrc.rf)]
vad.rf <- vad.rf[, list(title, vad.rf)]
afinn.ksvm <- afinn.ksvm[, list(title, afinn.ksvm)]
nrc.ksvm <- nrc.ksvm[, list(title, nrc.ksvm)]
vad.ksvm <- vad.ksvm[, list(title, vad.ksvm)]

# Set keys
setkey(ensemble, title)
setkey(afinn.rf, title)
setkey(nrc.rf, title)
setkey(vad.rf, title)
setkey(afinn.ksvm, title)
setkey(nrc.ksvm, title)
setkey(vad.ksvm, title)

# a series of left-joins
ensemble <- afinn.rf[ensemble]
ensemble <- nrc.rf[ensemble]
ensemble <- vad.rf[ensemble]
ensemble <- afinn.ksvm[ensemble]
ensemble <- nrc.ksvm[ensemble]
ensemble <- vad.ksvm[ensemble]

# We can look at the matrix we've created
ensemble[, afinn.rf:vad.ksvm]
```

```
##        afinn.rf nrc.rf vad.rf afinn.ksvm nrc.ksvm vad.ksvm
##     1:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##     2:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##     3:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##     4:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##     5:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##    ---
```

```
## 31907:    FALSE  FALSE   TRUE      FALSE    FALSE    FALSE
## 31908:     <NA>  FALSE   TRUE       <NA>    FALSE    FALSE
## 31909:     <NA>   TRUE   TRUE       <NA>    FALSE    FALSE
## 31910:     <NA>   TRUE   TRUE       <NA>    FALSE    FALSE
## 31911:     TRUE   TRUE   TRUE       TRUE     TRUE     TRUE
```

```
# Take the columns of predictions, convert them to a matrix of
# boolean values, then take the mean of each row.
ensemble[
  ,
  ensemble.mean := rowMeans(do.call(
    cbind,
    lapply(ensemble[, afinn.rf:vad.ksvm], as.logical)
  ), na.rm = TRUE)
]

# Convert the means above to predictions as a factor
# Predictions > 0.5 align with predicting is_fake = TRUE
ensemble[
  ensemble.mean > 0.5,
  ensemble := "TRUE",
]

# Predictions < 0.5 align with predicting is_fake = FALSE
ensemble[
  ensemble.mean < 0.5,
  ensemble := "FALSE",
]

# If the prediction is exactly 0.5, use naive guessing
ensemble[
  ensemble.mean == 0.5,
  ensemble := sample(c("TRUE", "FALSE"), .N, replace = TRUE),
]

# Make this column a factor for use in confusionMatrix
ensemble[, ensemble := as.factor(ensemble)]

# See the results
confusionMatrix(ensemble$ensemble, ensemble$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE  TRUE
##      FALSE 16167  2426
##      TRUE    576 12742
##
##                Accuracy : 0.906
##                  95% CI : (0.903, 0.909)
##     No Information Rate : 0.525
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.81
```

```
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.966
##             Specificity : 0.840
##          Pos Pred Value : 0.870
##          Neg Pred Value : 0.957
##              Prevalence : 0.525
##          Detection Rate : 0.507
##    Detection Prevalence : 0.583
##       Balanced Accuracy : 0.903
##
##        'Positive' Class : FALSE
##
```

```
# 90.6%
```

Understandably, the ensemble model is about as accurate as the mean of the individual accuracies for our 6 models. It will remain to be seen if this continues to be as consistently accurate when applied to our testing dataset

Results

Random Forests The results for our Random Forest models suffer against the testing set when compared to the results from the training set. This is expected, but we are still getting very good results. In fact, as we'll see below, the RF models outperformed the SVM models entirely. These results show that the NRC lexicon gave us the best results for a RF approach.

```
## Make final predictions/measure Acc
# Afinn RF
confusionMatrix(predict(afinn.rf.model, afinn.testing), afinn.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3759 1099
##      TRUE    413 3286
##
##                Accuracy : 0.823
##                  95% CI : (0.815, 0.831)
##     No Information Rate : 0.512
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.648
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.901
##             Specificity : 0.749
##          Pos Pred Value : 0.774
##          Neg Pred Value : 0.888
##              Prevalence : 0.488
##          Detection Rate : 0.439
##    Detection Prevalence : 0.568
##       Balanced Accuracy : 0.825
##
##        'Positive' Class : FALSE
##
```

```
# 82.3%

# NRC RF
confusionMatrix(predict(nrc.rf.model, nrc.testing), nrc.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3921  575
##      TRUE    330 3867
##
##                Accuracy : 0.896
```

```
##                  95% CI : (0.889, 0.902)
##     No Information Rate : 0.511
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.792
##
##  Mcnemar's Test P-Value : 5.03e-16
##
##             Sensitivity : 0.922
##             Specificity : 0.871
##          Pos Pred Value : 0.872
##          Neg Pred Value : 0.921
##              Prevalence : 0.489
##          Detection Rate : 0.451
##    Detection Prevalence : 0.517
##       Balanced Accuracy : 0.896
##
##        'Positive' Class : FALSE
##
```

```r
# 89.6%

# NRC VAD RF
confusionMatrix(predict(vad.rf.model, vad.testing), vad.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3809  628
##      TRUE    442 3826
##
##                Accuracy : 0.877
##                  95% CI : (0.87, 0.884)
##     No Information Rate : 0.512
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.754
##
##  Mcnemar's Test P-Value : 1.55e-08
##
##             Sensitivity : 0.896
##             Specificity : 0.859
##          Pos Pred Value : 0.858
##          Neg Pred Value : 0.896
##              Prevalence : 0.488
##          Detection Rate : 0.438
##    Detection Prevalence : 0.510
##       Balanced Accuracy : 0.878
##
##        'Positive' Class : FALSE
##
```

```
# 87.7%
```

Support Vector Machines The KSVM model results predicting against the training set are far closer to their counterparts when predicting for the testing set. We again see below that NRC lexicon gave us the best results with this kind of model.

```
# Afinn SVM
confusionMatrix(predict(afinn.ksvm.model, makePredictors(afinn.testing)), afinn.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3758 1111
##      TRUE    414 3274
##
##                Accuracy : 0.822
##                  95% CI : (0.814, 0.83)
##     No Information Rate : 0.512
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.645
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.901
##             Specificity : 0.747
##          Pos Pred Value : 0.772
##          Neg Pred Value : 0.888
##              Prevalence : 0.488
##          Detection Rate : 0.439
##    Detection Prevalence : 0.569
##       Balanced Accuracy : 0.824
##
##        'Positive' Class : FALSE
##
```

```
# 82.2%

# NRC SVM
confusionMatrix(predict(nrc.ksvm.model, makePredictors(nrc.testing)), nrc.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3971  787
##      TRUE    280 3655
##
##                Accuracy : 0.877
##                  95% CI : (0.87, 0.884)
```

```
##     No Information Rate : 0.511
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.755
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.934
##             Specificity : 0.823
##          Pos Pred Value : 0.835
##          Neg Pred Value : 0.929
##              Prevalence : 0.489
##          Detection Rate : 0.457
##    Detection Prevalence : 0.547
##       Balanced Accuracy : 0.878
##
##        'Positive' Class : FALSE
##
```

```
# 87.7%

# NRC VAD SVM
confusionMatrix(predict(vad.ksvm.model, makePredictors(vad.testing)), vad.testing$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3961  874
##      TRUE    290 3580
##
##                Accuracy : 0.866
##                  95% CI : (0.859, 0.873)
##     No Information Rate : 0.512
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.733
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.932
##             Specificity : 0.804
##          Pos Pred Value : 0.819
##          Neg Pred Value : 0.925
##              Prevalence : 0.488
##          Detection Rate : 0.455
##    Detection Prevalence : 0.555
##       Balanced Accuracy : 0.868
##
##        'Positive' Class : FALSE
##
```

```
# 86.6%
```

Ensemble Model We can again make all 6 sets of predictions, and gather an ensemble model. The results are not much better than the results of the individual models, but this isn't unexpected since the accuracies of each individual model are not hugely different.

```
# Ensemble model for testing dataset
# Create a container for our models, gathering all articles in the training set
ensemble <- rbind(
  afinn.testing[, list(title, is_fake)],
  nrc.testing[, list(title, is_fake)],
  vad.testing[, list(title, is_fake)]
)

# Take only the unique articles
ensemble <- unique(ensemble)

# Create data.tables from the testing data with a column for their respective
# predictions
afinn.rf <- cbind(
  afinn.testing,
  afinn.rf = predict(afinn.rf.model, makePredictors(afinn.testing))
)
nrc.rf <- cbind(
  nrc.testing,
  nrc.rf = predict(nrc.rf.model, makePredictors(nrc.testing))
)
vad.rf <- cbind(
  vad.testing,
  vad.rf = predict(vad.rf.model, makePredictors(vad.testing))
)
afinn.ksvm <- cbind(
  afinn.testing,
  afinn.ksvm = predict(afinn.ksvm.model, makePredictors(afinn.testing))
)
nrc.ksvm <- cbind(
  nrc.testing,
  nrc.ksvm = predict(nrc.ksvm.model, makePredictors(nrc.testing))
)
vad.ksvm <- cbind(
  vad.testing,
  vad.ksvm = predict(vad.ksvm.model, makePredictors(vad.testing))
)

# Remove columns not needed for this step
afinn.rf <- afinn.rf[, list(title, afinn.rf)]
nrc.rf <- nrc.rf[, list(title, nrc.rf)]
vad.rf <- vad.rf[, list(title, vad.rf)]
afinn.ksvm <- afinn.ksvm[, list(title, afinn.ksvm)]
nrc.ksvm <- nrc.ksvm[, list(title, nrc.ksvm)]
vad.ksvm <- vad.ksvm[, list(title, vad.ksvm)]

# Set keys
```

```
setkey(ensemble, title)
setkey(afinn.rf, title)
setkey(nrc.rf, title)
setkey(vad.rf, title)
setkey(afinn.ksvm, title)
setkey(nrc.ksvm, title)
setkey(vad.ksvm, title)

# a series of left-joins
ensemble <- afinn.rf[ensemble]
ensemble <- nrc.rf[ensemble]
ensemble <- vad.rf[ensemble]
ensemble <- afinn.ksvm[ensemble]
ensemble <- nrc.ksvm[ensemble]
ensemble <- vad.ksvm[ensemble]

# We can look at the matrix we've created
ensemble[, afinn.rf:vad.ksvm]
```

```
##       afinn.rf nrc.rf vad.rf afinn.ksvm nrc.ksvm vad.ksvm
##    1:     TRUE   TRUE   TRUE       TRUE     TRUE     TRUE
##    2:     TRUE   TRUE   TRUE       TRUE     TRUE     TRUE
##    3:     <NA>   TRUE   TRUE       <NA>     TRUE     TRUE
##    4:     TRUE   TRUE   TRUE       TRUE     TRUE     TRUE
##    5:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
##   ---
## 8701:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
## 8702:     <NA>  FALSE  FALSE       <NA>    FALSE    FALSE
## 8703:     <NA>  FALSE  FALSE       <NA>    FALSE    FALSE
## 8704:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
## 8705:    FALSE  FALSE  FALSE      FALSE    FALSE    FALSE
```

```
# Take the columns of predictions, convert them to a matrix of
# boolean values, then take the mean of each row.
ensemble[
  ,
  ensemble.mean := rowMeans(do.call(
    cbind,
    lapply(ensemble[, afinn.rf:vad.ksvm], as.logical)
  ), na.rm = TRUE)
]

# Convert the means above to predictions as a factor
# Predictions > 0.5 align with predicting is_fake = TRUE
ensemble[
  ensemble.mean > 0.5,
  ensemble := "TRUE",
]

# Predictions < 0.5 align with predicting is_fake = FALSE
ensemble[
  ensemble.mean < 0.5,
  ensemble := "FALSE",
```

```
]

# If the prediction is exactly 0.5, use naive guessing
ensemble[
  ensemble.mean == 0.5,
  ensemble := sample(c("TRUE", "FALSE"), .N, replace = TRUE),
]

# Make this column a factor for use in confusionMatrix
ensemble[, ensemble := as.factor(ensemble)]

# See the results
confusionMatrix(ensemble$ensemble, ensemble$is_fake)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction FALSE TRUE
##      FALSE  3998  869
##      TRUE    253 3585
##
##                Accuracy : 0.871
##                  95% CI : (0.864, 0.878)
##     No Information Rate : 0.512
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.743
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.940
##             Specificity : 0.805
##          Pos Pred Value : 0.821
##          Neg Pred Value : 0.934
##              Prevalence : 0.488
##          Detection Rate : 0.459
##    Detection Prevalence : 0.559
##       Balanced Accuracy : 0.873
##
##        'Positive' Class : FALSE
##
```

```
#  87.1%
```

Conclusion

This analysis has been a very brief and surface-level exploration of classifying "fake" news using machine learning. There are a great many more techniques to be applied, such as bigram analysis and and Natural Language Processing. This work is not uncommon - many social media and news outlets are attempting to do exactly this kind of analysis on their own content so as to be responsive to the needs of their consumers and responsible for the platform they give to the people producing that media.

I believe the greatest limitation this report suffers is the source data itself. Any and all conclusions drawn from it should be taken with a grain of salt - the original data selection was skewed and biased, and the aggregated source data was not large enough to consider this analysis a thorough one. If there was a larger, more robust, and more objectively classified set of data one could apply these approaches as the starting point to further exploration of these and other techniques.

Citations

1. Ahmed H. SS Traore I. (2017) Detection of Online Fake News Using N-Gram Analysis and Machine Learning Techniques. In: Traore I, Woungang I, Awad A (eds) Intelligent, Secure, and Dependable Systems in Distributed and Cloud Environments. Springer, Cham, Switzerland, pp 127–138

2. Ahmed H SS Traore I (2018) Detecting Opinion Spams and Fake News Using Text Classification. Journal of Security and Privacy 1:

3. Nielsen F Årup (2011) A New ANEW: Evaluation of a Word List for Sentiment Analysis in Microblogs. CoRR abs/1103.2903:

4. Mohammad SM, Turney PD (2013) CROWDSOURCING a Word–Emotion Association Lexicon. Computational Intelligence 29:436–465. https://doi.org/10.1111/j.1467-8640.2012.00460.x

5. Mohammad SM (2018) Obtaining Reliable Human Ratings of Valence, Arousal, and Dominance for 20,000 Englishwords