Scoring Short Answer Essays

Luis Tandalla

September 2012

Introduction

This article thoroughly presents the techniques I used in my final model submission for the Kaggle competition "Automated Student Assessment Prize, Phase Two – Short Answer Scoring." The William and Flora Hewlett Foundation sponsored this competition "in hopes of discovering new tools to support schools and teachers." Kaggle competitors had the challenge to "develop a scoring algorithm for student-written short-answer responses." These responses consisted of essays of approximately fifty words which were written by 10th grade students answering questions that "cover a broad range of disciplines (from English Language Arts to Science)." Kaggle provided a training data of approximately 17,000 essays with two scores graded by two different people. They also provided two data sets as test data which consists of approximately 6,000 essays each, but these ones did not have their scores. I, as a competitor, developed a system which predicts the score given by the first human grader for these two test data sets. This paper describes in detail how I developed the system and how it currently works.

Preamble

Before reading procedure, the following information is needed:

- The paper makes reference to the source code of the system which is available at https://github.com/luistp001/LT-Autograder.
- The data have a column called SET, which is an identifier to the different questions asked to the students. There are 10 sets.
- In all of the supervised learning methods, the essay sets were trained individually and treated as regression. To calculate the score of a new essay, the algorithm predicts a real value which is later rounded off and this number is the score predicted.
- In all the supervised learning methods, the number of words, the number of sentences and the sentence average size are included as features even if it is not explicitly stated.

Preprocessing Stage

As a first step for cleaning the data, all non-alphabetic characters, with the exception of periods, are replaced by spaces. In the case of essays of set 10, non-alphanumeric characters are replaced by spaces. The special treatment for set 10 is done because question 10 makes reference to experimental numerical data, so there are some specific numbers that influence the score of the essays. Later, all capital letters are replaced by lower case letters.

The next step that helps to clean the data is to correct misspelled words. I wrote a spelling corrector based on Peter Norvig's spelling corrector, which can be found at http://norvig.com/spell-correct.html. However, my spelling corrector has some features specific to this data.

Spelling Corrector

This section describes how the spelling corrector works.

First, it needs a list of all American English words. I used the file ‘American-English’ which is available at ‘/usr/share/dict/’ in any Ubuntu distribution. From that list of words, I used the words that contain only lower case letters. The spelling corrector uses this list of words as the ‘right words’ (words with no spelled errors, in the source code is called “lexicon”).

For each set, there are certain words that are 'special' to that set. They are special in the sense that they are uncommon in general or are not included in the list of 'right words,' but are very common to a set. For example, trna is not a common word in general but it is common in sets 5 and 6. 'Vinegar' is a ‘right word’ but it is significantly common in set 1. Thus, the spelling corrector has lists of ‘special’ words for each set. In addition, there is a list 'extra' for words that are not in the list 'right' but have no spelled errors. Words that are slightly more common in this data than in general English are also included in the ‘extra’ list. For example, the word “hasnt” is not included in the 'right' list but is very common for the essays because “hasn't” was reduced to “hasnt” in the first step.

The spelling corrector also has counts of words and bigrams to choose the better correction for a word. These counts are obtained by counting the number of occurrences of each word and bigram in the file ‘big.txt,’ available at http://norvig.com/big.txt. In the source code, these counts are saved as dictionaries called ‘nwords’ and ‘nwords2.’

In some cases, the spelling error is that two words are written together. When one of the words has two letters, the spelling corrector would usually eliminate this word. For example, the spelling corrector would return 'play' when correcting 'toplay'. To avoid eliminating a legitimate word, the spelling corrector checks if a ‘right’ is going to be eliminated before returning a word, and if so, both words are returned. Thus, the spelling corrector returns 'to play' when correcting 'toplay.'

When correcting a word, the set to which it belongs, and optionally the previous and next word, is given to the spelling corrector. Whenever a correction is made, the correction is saved in dictionaries called 'caches,' which avoid recalculation of the right word for a misspelled word.

After the steps described above, the process to correct each word is:

1. The spelling corrector checks if the word is in the list of 'right' words. If it is, the same word is returned.
2. There are specific rules for some words such as
   if word == 'alot': return 'a lot'

   To avoid ambiguity in the following steps, the word to correct will be called 'misspelled.' 'Previous' is the word that is written before 'misspelled' in the essay, and 'next' is the word written after 'misspelled.'

3. A list of words that have edit distance of 1 to misspelled and that are 'right' is created as 'candidates.'

4. A list of words that have edit distance of 2 to misspelled and that are in 'nword' is created as 'candidates2.'
5. If any of the words in candidates or in candidates 2 is 'special,' this word is returned.
6. 'Previous' with each word in candidates2 form bigrams. If one or more of these bigrams is in nwords2, the word in candidates2 that produced the bigram with the highest count is returned.
7. The word in candidates with the highest count in 'nwords' is returned if this word is 'right'.

Steps 8 through 10 apply for words with 4 letters or more.

8. A list 'splits' is created with pair of words that come from splitting misspelled in two parts. As a condition, each word of the pair must have at least two letters.
9. If both words are 'right' in any of the splits, the split is returned. For instance, in the case of 'thevinegar,' 'the vinegar' is returned.
10. If there are one or more splits in which only one word is right, the split with the longest 'right' word is chosen. The misspelled word of the chosen split is corrected recursively. Then, the split is returned. For example, 'thevinigar' would be split in 'the vinegar,' and after correcting 'vinigar' with 'vinegar,' the spelling corrector returns 'the vinegar.'
11. If until this point the script has not returned a word, 'misspelled' is returned.

The following steps only apply to set 10.

12. If 'mispelled' is a number, it is returned.
13. Before proceeding to step 2, 'next' word is added to 'misspelled' word. If the combined word is 'right,' it is returned, and 'next' word is removed from the essay. For example, if 'misspelled' word is 'dogho' and 'next' word is 'use,' 'doghouse' is returned and 'use' is removed from the essay.

After correcting the misspelled words, all of the words are stemmed using the Porter Stem Algorithm. The implementation used is available at http://tartarus.org/martin/PorterStemmer/python.txt .

Testing of usefulness of cleaned data

With the new data obtained I trained a Random Forest to predict the scores, using the counts of words as features and using default parameters. Then, I trained two more Random Forests in which I included the counts of bigrams as features. In the third one, also the counts of trigrams was used as features. Comparing them to the Benchmark Bag of Words Model provided by Kaggle, they produced the following results.

	kappa
Benchmark Bag of Words	0.64554
My Bag of Words	0.67863
Bag of Words and Bigrams	0.68789
Bag of Words, Bigrams and Trigrams	0.68514

Furthermore, the Benchmark Bag of Words was also trained using a Random Forest. The difference of this model with my models is that they used cleaner data. The Bag of Words and Bigrams had the best performance of all. Also, it is noted that including the trigrams as features slightly decreases the performance of the model but also it considerably increases the training time. Consequently, I no longer used the counts of trigrams as features for training the scores of the essays.

Using Term Frequency, Inverse Document Frequency

Instead of using the raw counts, I also trained Random Forests by changing the counts to term frequency – inverse document frequencies normalized. This number is described in *Introduction to Information Retrieval* by Manning, Raghavan and Schütze . It is calculated as follows for each word and bigram in each essay:

Defining:

tf-idf: term frequency – inverse document frequency
tf: term-frequency.
idf: inverse document frequency
N: number of essays
dfw: document frequency of a term

tf = the raw counts of a word in a given essay
dfw = the number of essays in which a word appears

$$tfidf = \left(1 + log(tf)\right) \cdot log\left(\frac{N}{dfw}\right)$$

To normalize, for each essay:

lend = sum of the squares of all tf-idfs
$tfidf_{normalized} = \frac{tfidf}{lend}$ for each word.

I expected to get higher accuracy using the tf-idf weighting, but I got the following results:

	Kappa	
	With raw counts	With tf-idf weighting
Bag of Words	0.67863	0.65481
Bag of Words and Bigrams	0.68789	0.67005

As a conclusion, raw counts give a better performance than tf-idf weighted counts.

Choosing relevant words and bigrams

To avoid over-fitting the training data and reduce training time and noisiness in the data, I did not use all of the words that appear in each set. First of all, I only included the counts of words and bigrams that appear at least 10 times. After that, I tried an approach similar to Latent Semantic Analysis. I used principal component analysis (PCA) to reduce the number of words in the vocabulary of each set to a

lower number of variables that retained 95% of the variance. I trained Random Forests with the variables that PCA calculated using the raw counts of words. This approach produced a lower kappa. I tried again retaining 99% of the variance and later using the tf-idf weighted counts, but the result was the same, a decrease of performance.

Another approach to use was choosing the relevant features with the Boruta algorithm implemented in the Boruta package. The algorithm selects features as important if they are better predictors than random features, so it eliminates features that do not help to predict the scores. More information can be found at http://www.jstatsoft.org/v36/i11/paper. I ran the Boruta algorithm in each set using the raw counts of words as variables. The Boruta algorithm produced a subset words as 'important' words. I trained Random Forests with these subset of words and obtained a slightly increase of performance and a considerable reduction of training time. I repeated the procedure to obtain the 'important bigrams.' After this point, I only used the 'important words' and the 'important bigrams' for the training and development of the model.

Finding specific answer for each set

The following section was one key factor for the performance of my model.

To get higher scores, human graders look for specific answers (ideas or concepts) in the essays. For example, the prompt of set 1 is "Draw a conclusion based on the student's data. Describe two ways the student could have improved the experimental design and/or validity of the results" and the possible answers are:

- You need to know how much vinegar was used in each container.
- You need to know what type of vinegar was used in each container.
- You need to know what materials to test.
- You need to know what size/surface area of materials should be used.
- You need to know how long each sample was rinsed in distilled water.
- You need to know what drying method to use.
- You need to know what size/type of container to use.
- Other acceptable responses.

My goal was that my model does the same, so I wrote regular expressions that look for the specific answers in the essays. However, each of these answers can be written in several different ways. Therefore, I wrote various regular expressions for each of the answers. For instance, to find if one essay contains the idea 'to know what type of vinegar,' my python script searches for if any of the following regular expressions are present:

```
((type)|(kind)|(brand)) of vinegar
((concentr)|( \w?h)|(acid)) (\w+ ){0,5}vinegar
vinegar (\w+ )?((concentr)|(acid))
know (\w+ )?vinegar (\w+ )?((us)|(need))
```

Also, I read some of the essays to find what the 'other acceptable responses' can be. Thus, the script searches for each of the regular expressions for each answer and returns a list containing zeros or ones:

ones if the essay has a regular expression for the answers, or 0 if it does not. For instance, the essay, "For the students to be able to make a replicate, they would need to tell use how much vinegar is used and what tipe of materials is needed for the expirement", has the following answers:

Answer (idea or concept) for set 1	Does the essay have the answer?
How much vinegar	1
Size of container	0
Type of container	0
Type of material or samples	1
Size of material or samples	0
Type of vinegar	0
Location of container	0
Surface area of sample	0
Room temperature	0
How to rinse	0
Is container covered?	0
Are samples completely submerged?	0
Type of climate	0

Running Random Forests using only these lists as features produced a kappa of over 0.70 in the public leaderboard.

Later, from those lists, I extracted more features specific for each set.

Set	Features Added in the essays
2	Feature 1: does it have a 'conclusion answer' such as 'type a stretch the least?'
	Feature 2: number of 'ways to improve' answers it has.
3	Feature 1: According to the regular expressions, does it have any of the following answer? Alligator Specialist Generalist Have trouble adapting
	Feature 2: According to the regular expressions, does it have any of the following answer? Eat exclusively one type of food Can live in many different places Eat different kinds of food Live only in one type of place
	Feature 3: Does it have two or more of the answers of feature 2.

4	Feature 1: According to the regular expressions, does it have at least one answer?
	Feature 2: According to the regular expressions, does it have at least four of the answers?
7	Feature 1: does it have a 'trait' answer such as 'hard working?'
	Feature 2: does it have a 'detail' answer such as 'help Paul with his college expenses?
10	Feature 1: does it have any of the following answers? Blank Light gray Dark gray Black
	Feature 2: does it have one of the 'answers' that describe the effect of the color?
	Feature 3: does it have one of the 'asnwers' that use results from the experiment?

In addition to the features described above, I also included the number of answers that the regular expressions found. Let me call 'answers' to the list produced by the regular expressions and the features I extracted from them.

Using the counts of words, the counts of bigrams, and answers as features for Random Forests I obtained a kappa of approximately 0.73 in the public leaderboard.

Measuring similarities

Although the answers found by the regular expressions increased the performance of my model, there are many false positives and negatives. An answer can be written in more ways than the regular expression can find, so I would need much more regular expressions to find all these different ways. Also, the regular expressions return 1or 0, meaning they are completely certain whether an answer is present or not. Thus, a measure similar to a probability that an answer is present would be better.

My first approach to finding the probability of an answer was to find the similarity between the essay and the answer. Each essay has already calculated a vector of tf-idf weighted counts of the words. I also had to find this vector for the answers. When an answer was found by the regular expression, there was a match. After searching the essays of all the training data, I found there were several matches for each answer. Then, one vector for each answer was built with the matches that have the answer. For example:

The following regular expressions find the answer ‘type of vinegar,’

```
(\w+ ){0,4}\w*((type)|(kind)|(brand)) of vinegar\w*( \w+){0,4}
(\w+ ){0,4}\w*((concentr)|( \w?h)|(acid)) (\w+ ){0,5}vinegar\w*( \w+){0,4}
(\w+ ){0,4}\w*vinegar (\w+ )?((concentr)|(acid))\w*( \w+){0,4}
(\w+ ){0,4}\w*know (\w+ )?vinegar (\w+ )?((us)|(need))\w*( \w+){0,4}
```

And the following essays contain that answer,

Essay Id	Match found by regular expressions
12	need to know the kind of vinegar
20	thei should includ what type of vinegar or what brand so
26	need to know what type of vinegar thei were us because
68	would need the same kind of vinegar in each contain
117	also state the mass of ph label of the vinegar

Those matches converted to vectors of words would be,

EssayId	12	20	26	68	117	In all the matches
need	1	0	1	1	0	3
to	1	0	1	0	0	2
know	1	0	1	0	0	2
the	1	0	0	1	1	3
kind	1	0	0	1	0	2
of	1	1	1	1	2	6
vinegar	1	1	1	1	1	5
thei	0	1	0	0	0	1
should	0	1	0	0	0	1
includ	0	1	0	0	0	1
what	0	2	1	0	0	3
type	0	1	1	0	0	2
or	0	1	0	0	0	1
brand	0	1	0	0	0	1
so	0	1	0	0	0	1
thei	0	0	1	0	0	1
were	0	0	1	0	0	1
us	0	0	1	0	0	1
because	0	0	1	0	0	1
would	0	0	0	1	0	1
same	0	0	0	1	0	1
in	0	0	0	1	0	1
each	0	0	0	1	0	1
contain	0	0	0	1	0	1
also	0	0	0	0	1	1
state	0	0	0	0	1	1
the	0	0	0	0	1	1

mass	0	0	0	0	1	1
ph	0	0	0	0	1	1
label	0	0	0	0	1	1

I used the vector of words in all the matches with tf-idf weighting as the vector of words for an answer. Then, the similarity between an essay and an answer is calculated using cosine similarity described in *Introduction to Information Retrieval* by Manning, Raghavan and Schütze. The formula for its calculation is:

$$similarity1 = \frac{v_e \cdot v_a}{|v_e||v_a|}$$

with v_e as the vector of words for the essay and v_a as the vector of words for the answer.

The similarity calculated would have a value between -1 and 1, so that is better than only 0 or 1.

Another form of similarity was calculated later. This time, instead of having calculated the similarity between the vectors of words of the essay with the vectors of words of the answer, I calculated the similarity between the essay and each of the matches of an answer. Then, the highest value was taken as the similarity between the essay and the answer. That would be,

$$similarity2 = max\left(\frac{v_e \cdot v_m}{|v_e||v_m|}\right)$$

with v_e as the vector of words for the essay and v_m as the vector of words for each of the matches.

I used these similarities as features to train new Random Forests for predicting the score of the essays, but they had poor performance. Using the answers just as they are found by the regular expressions gives a higher kappa than using these measures of similarities.

Predicting the probability of an answer

This section explains another key element for the performance of the model.

I still wanted a measure of probability for the answers, and I knew that predicting if an essay has a specific answer would be easier and more accurate than predicting the score. Consequently, with the help of a python program, I manually labeled if the essays of the training data have the 'answers' or not. The python script and the labels are included in the source code. This was not done for set 4. Then, using these 'labels, I ran a Random Forest that predicts the probability that an essay has an answer. Later, those probabilities are used as features to predict the scores of the essay.

Before running the Random Forests, I ran the Boruta algorithm with the counts of words as features to choose the relevant words needed to predict if an essay has a specific answer. I repeated this procedure using the counts of bigrams and later using the counts of trigrams. Finally, I used the counts of relevant words, bigrams and trigrams given by the Boruta algorithm as features for the Random Forests to predict the 'labels.'

I chose Random Forests because it is fast to train and accurate enough. Ten-fold-cross-validation was used and each set of the 10 sets has approximately 15 possible answers, so I needed to train approximately 1,500 different models. Therefore, I needed a fast algorithm. Since the predictions of these models would be used as features, and I labeled the data by looking at the training essays. These

new features tend to over-fit the training data. For this reason, I also needed a supervised learning algorithm that is not prone to over-fitting. Thus, I chose Random Forest with default parameters for these models. I chose words, bigrams, and trigrams as features because after running a few Random Forests, I found that the three together give the best performance.

After having the relevant words, bigrams and trigrams, I ran Random Forests as described above. Then, the probabilities predicted by them are saved for later use. For the calculation of these probabilities in the Public and Private Leaderboards, one Random Forest is trained for each answer in each set using all the training data. For the probabilities in the training data, 10 fold cross validation is used to calculate them. The cross validation is run five times and the probabilities are then averaged. Thus, 50 Random Forests are run for each answer in each set. Reducing the number of cross validations trained and the number of folds would reduce the number of Random Forests trained, and so the training time, but also they may reduce the performance of the overall model by over-fitting to the training data. The Random Forests were trained as classification with "mtry" equal to a third of the number of variables. The other parameters used the default values.

A special type of 10 fold cross validation is done when no more than 30 essays have a certain answer. Usually, with 10-fold-cross- validation, the whole training data is split into 10 parts, and then 9 of them are used as the cross validation training data. For these particular answers, the training data with labels of 1 is split into 10 parts, and separately, the training data with labels of 0 is split into 10 parts. Then 9 parts of the training data with positive labels and 9 parts with negative labels are combined to be the cross validation training data. The remaining positive and negative parts come to be the cross validation test data. This is done to maintain the distribution in the cross validation training data.

I obtained a kappa of approximately 0.75 with Random Forests that predict the scores of the essays using the counts of the important words, the counts of the important bigrams, the answers found by the regular expressions and the probabilities of having the answers as features.

Choosing the relevant answers

The Boruta algorithm was used also to determine the relevant answers that help to predict the score. The same was done to find the relevant probabilities. Running Random Forest with the relevant words, bigrams, answers, and probabilities produced a kappa of approximately 0.76 in the public leaderboard. I trained several Random Forest with different combinations of these features to predict the score and concluded that using the four gives the best performance.

Using the score of the second grader

The goal of the competition was to predict the score of the first grader. However, I also used the score of the second grader in the training.

The scores of the two graders were occasionally different, which means that any of them was wrong, so the scores for those essays will have noise. One way to eliminate this noise would be to only use the essays in which both graders agree for training. However, this still loses information because if an essay can have grades from 0 to 3 and if the grades given are 2 and 3, then the supervised learning algorithm still can learn that the essay deserves a high grade. Therefore, instead of training to predict the scores given by the first grader, I train the model to predict the scores given by both graders.

I had two copies of the training data with the score of the first grader as the score to predict in one copy, and the score of the second grader to predict in the second copy. I joined the two copies together and the combined data was used for training.

For example, for the first 10 essays the original data is:

Id	Score1	Score2	EssayText
1	1	1	Some additional ...
2	1	1	After reading the ...
3	1	1	What you need ...
4	0	0	The student should ...
5	2	2	For the students ...
6	1	0	I would need ...
7	1	0	The information ...
8	3	3	You would need ...
9	3	3	Some additional ...
10	2	2	Inorder to ...

My training data instead is changed to

Id	Score	EssayText
1	1	Some additional ...
1	1	Some additional ...
2	1	After reading the ...
2	1	After reading the ...
3	1	What you need ...
3	1	What you need ...
4	0	The student should ...
4	0	The student should ...
5	2	For the students ...
5	2	For the students ...
6	1	I would need ...
6	0	I would need ...
7	1	The information ...
7	0	The information ...
8	3	You would need ...
8	3	You would need ...
9	3	Some additional ...
9	3	Some additional ...
10	2	Inorder to ...
10	2	Inorder to ...

In this way, the model gives preference to the essays in which both graders agree and know when an essay is hard to grade or any of the graders is wrong. Thus, this approach is used to reduce over-fitting. When I used training Random Forests with the double data, there was an increase of performance.

With the last features described above (relevant, words, bigrams, and probabilities) and with the combined data I trained four supervised learning algorithms:

Algorithm	Parameters
Random Forest	Default
Random Forest	Default
Gradient Boosting Machine	Interaction depth = 5 Bag fraction = 0.5 shrinkage = 0.001 number of trees = 5000
Gradient Boosting Machine	Bag fraction = 0.5 Bag fraction = 0.5 shrinkage = 0.01 number of trees = 500

Finally, to get higher kappa I averaged the predictions of each of the four models. However, I averaged the predictions before they are rounded. Then, I round this average and that is the final score given by the model.

Summary of final model for training

Misspelled words are corrected.
All words are stemmed.
Boruta Algorithm chooses relevant words and bigrams to predict scores.
Regular expressions look for answers.
Boruta Algorithm chooses relevant words, bigrams and trigrams to predict labels.
Random Forests are trained to predict probabilities of labels using the relevant words, bigrams, and trigrams.
10 fold cross validation is trained 5 times to predict probabilities of labels for the training essays.
Boruta Algorithm chooses relevant answers and probabilities to predict scores.
The training data is doubled using the score of the second grader.
2 Random Forests and 2 Gradient Boosting Machine are trained in each set to predict the scores of the essays using relevant words, bigrams, answers, and probabilities as features.
The average of the real values predicted by the last 4 models is rounded and becomes the score given by the whole model.

Summary of final model for testing

Misspelled words are corrected.
All words are stemmed.
Regular expressions look for answers.
The probabilities of labels are predicted using the already trained Random Forests.
The scores are predicted using the 2 Random Forests and the 2 Gradient Boosting Machine already trained in each set.
The average of the real values predicted by the last 4 models is rounded and becomes the score given by the whole model.

Additional Comments and Observations

I also trained Support Vector Machines and Neural Networks to predict the scores and to blend models. They constantly produced higher cross validations kappas but lower public leaderboard kappas. I suppose they give more importance to the variables I created (answers and probabilities), and because these variables are already fit to the training data, those algorithms overfit it. Thus, Support Vector Machines and Neural Networks did not work using my calculated features. Also, training a supervised learning algorithm to blend models did not work. However, by simply averaging four models, I reduced the variance of my models, and, consequently, the kappa ia higher.

To autograde the essays of set 6 was the easiest of all. This is due because answers for question 6 are concrete facts. My cross validation errors presented that the kappa for this set was usually 0.10 greater than for the combined kappa of then 10 sets. My highest public leaderboard kappa was 0.771, so I expect the kappa for only set 6 will be around 0.871. With some more time to improve the model, it would be possible to achieve the human benchmark kappa for set 6 and sets with prompts similar to it. On the other hand, scores for essays of sets 3, 4, 7, and 8, are relatively difficult to predict. This is due because their answers are not concrete facts that can be written in few similar words. The answers are more subjective and more diverse. My model treats all sets in the same way, but there could be a way in which those sets can be treated differently to get higher kappas.

References

A. Liaw and M. Wiener (2002). Classification and Regression by randomForest. R News 2(3), 18—22.

Brierley, Phil. "AusDM Analytic Challenge." *AusDM Analytic Challenge*. Tiberius Data Mining, n.d. Web. 10 July 2012. <http://www.tiberius.biz/ausdm09/index.html>.

Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schutze. "Scoring, Term Weighting and the Vector Space Model." *Introduction to Information Retrieval*. N.p.: Cambridge UP, 2008. N. pag. Print.

Miron B. Kursa, Witold R. Rudnicki (2010). Feature Selection with the Boruta Package. Journal of Statistical Software, 36(11), 1-13. URL http://www.jstatsoft.org/v36/i11/.

Norvig, Peter. "How to Write a Spelling Corrector." *How to Write a Spelling Corrector*. N.p., n.d. Web. 10 July 2012. <http://norvig.com/spell-correct.html>.

Porter, Martin. "Porter Stemming Algorithm." *Porter Stemming Algorithm*. N.p., Jan. 2006. Web. 10 July 2012. <http://tartarus.org/martin/PorterStemmer/>.

Ridgeway, Greg. "Generalized Boosted Models: A Guide to the Gbm Package." N.p., 3 Aug. 2007. Web. <http://cran.r-project.org/web/packages/gbm/vignettes/gbm.pdf>.