Identifying Low-Quality YouTube Comments

Alex Trytko and Stephen Young

CS229 Final Project - Fall 2012

YouTube provides an unparalleled platform for sharing and viewing video content of every imaginable nature. However, the user experience is tempered by the erratic quality of discourse manifested in the comments posted in response to videos. Unfortunately, the anonymity of the Internet seems to foster a preponderance of highly offensive or derogatory comments which serve little apparent purpose other than to intentionally provoke and upset other users. YouTube combats such comments via a voting system whereby a comment is hidden from view once it has received a certain number of downvotes. Users can only see the contents of these comments by specifically requesting to do so. This feedback mechanism allows the YouTube community to collectively decide which comments are offensive, inflammatory, or otherwise low-quality, thus elevating (however marginally) the average level of discourse.

A drawback to YouTube's current comment-hiding strategy is the unavoidable delay between the time when a low-quality comment is posted and when it accrues enough downvotes to be hidden. In this project, we employ machine learning concepts to classify YouTube comments *without* relying on human involvement. We train and test three different algorithms toward this end: multinomial Naive Bayes, a support vector machine with a linear kernel, and logistic regression. It is easy to see how classifiers like the ones used in this project can help to reduce (or even eliminate) this latency. Even if it is undesirable to show or hide a comment based solely on the output of the classifier, the predictions can be used to indicate how predisposed a comment is to being low-quality, and adjust the downvote threshold for that comment accordingly. If adopted, this approach has the potential to significantly improve user experience at minimal cost to YouTube.

Data Acquisition

Data acquisition was a *much* greater hassle than we had originally anticipated. While Google does provide APIs to access YouTube data (indeed, we used these APIs for some steps of the data acquisition process), they do not provide information about the individual ratings of comments. Therefore, our only recourse was to scrape this information directly from YouTube comment pages. However, the structure of YouTube comment pages made this task non-straightforward. Hidden comments (i.e., the ones in which we were interested) do not appear in the page until the user clicks a "Show the comment" link, which issues an AJAX request to retrieve the comment and inserts it into the page. To circumvent this obstacle, we used a nifty third-party Python package called Selenium to automatically expand all hidden comments before scraping all of the comments from the HTML document. A major downside of using Selenium is inefficiency - it takes on the order of minutes to extract comments from a single comment page. Unfortunately, Selenium would fail (seemingly at random) to expand some comments in a manner that prevented all subsequent hidden comments on the page from being expanded. After several failed attempts at workarounds, we resigned ourselves to simply discarding all comments after the initial point of failure while processing a comment web page.

A final (and particularly vexing) difficulty with data acquisition was YouTube's inconsistency in displaying hidden comments. Comments can be displayed in two contexts: either as a top-level comment or as the parent of a reply to that comment. In some cases, a comment marked as hidden in the latter context will *not* be marked as hidden in the former context. This behavior is obviously problematic because it implies that the same comment could exist twice in our dataset with different labels. To address this issue, we added a post-processing step wherein we eliminated all duplicate comments for a particular video, keeping the positive label in the event of label disagreement. That is, if the same comment appeared as both visible and hidden, we labeled it has hidden.

We performed some standard pre-processing operations before feeding the data into our algorithm. First, we strip out all hyperlinks from comments. We then convert all words to lowercase, remove punctuation, and drop words that are one character long. We then apply the Porter stemming algorithm to each remaining word in order to reduce similar words to their common roots and avoid treating words with the same effective meaning as separate features. We also selected a conservative set of stop words ('the', 'to', 'is', 'it', 'and', 'on', 'of', 'in') and discarded all occurrences of those words in our dataset, as well as all numbers.

Spell correction is another preprocessing step commonly applied in text classification problems. We decided *against* performing spell correction because misspellings may themselves be indicative of comment quality, and hence valuable to keep separate from the correct versions. Running spell correction on a poorly written comment may drastically alter the overall quality of the comment, and may make a difference as to whether or not readers would decide to downvote the comment.

Dataset summary statistics

Videos:	**215**
Comments (total):	**29,473**
Comments (hidden):	**323**

Feature Selection

As with any text classification problem, not every word in our dataset is likely to be a useful feature with respect to predicting comment quality. First, some words simply do not occur often enough to provide any indication of one class or the other. Because YouTube comments are rife with misspellings and colloquialisms, our dataset contains a high proportion of rare terms - of the 20,147 unique words in our dataset, over 12,000 only occur once. Therefore, our first feature pruning step was to disregard all words that occurred below a certain threshold value (hereafter Minimum Word Frequency, or MWF) number of times.

Another feature pruning technique focuses on words with similar conditional class probabilities; i.e., words that occur with similar frequency in visible and hidden comments. For algorithms that assume feature independence (i.e. Naive Bayes), eliminating such words could reduce noise and improve classification performance. We do so by computing the probabilities of a word given the positive class and given the negative class, dividing the larger by the smaller, and disregarding the word if the quotient falls below a certain threshold value (hereafter Minimum Conditional Probability Difference Factor, or MCPDF).

Note that throughout this paper, we regard MWF and MCPDF not as constants but as configurable parameters. We tried to be conscientious about not over-aggressively pruning the feature set by setting these values too high, which could result in spuriously high performance. We discuss this problem more thoroughly in the section entitled “A caveat about feature selection”.

A note about performance metrics

Because our dataset features a heavily-skewed class distribution (only 1.10% of all comments were hidden), raw classification accuracy is not a terribly useful indication of performance. For example, consider a dumb classifier that blindly gives all comments the negative label. This classifier would have a 98.9% success rate, despite being obviously useless. Therefore, when discussing classifier performance, we usually refer instead to AUC, which denotes the area under the precision-recall curve. AUC is bounded between 0 and 1. Note that the aforementioned dumb classifier yields AUC of 0.011, which properly reflects its true performance.

Furthermore, all performance values are calculated using *k*-fold cross-validation, usually with *k* = 10.

Multinomial Naive Bayes

An obvious first approach to any text classification problem is multinomial Naive Bayes. We elected to implement the multinomial (as opposed to multivariate) variant. This variant takes term frequency into account. Without feature pruning (i.e., with MWF and MCPDF both set to 1), Naive Bayes performed very poorly with an AUC of 0.015 - not much better than the dumb classifier would have done. The following table demonstrates the changes in performance (measured in AUC) of our Naive Bayes classifier as we adjust the value of the MWF and MCPDF parameters. The grayed-out entries correspond to parameter values that resulted in over-aggressive feature pruning, which we define as a feature set containing fewer than 700 words (recall that the full feature set has over 20,000 words).

MWF	**MCPDF**

	1	2	3	4	5	6	7
1	0.015002	0.015738	0.014733	0.015688	0.019106	0.020037	0.019709
2	0.017157	0.022256	0.025929	0.031835	0.032632	0.038652	0.053693
3	0.017393	0.024125	0.031028	0.03953	0.062151	0.105764	0.078696
4	0.017894	0.027052	0.035474	0.048626	0.075824	0.125244	
5	0.018097	0.029874	0.04131	0.055801	0.080000		
6	0.018093	0.028907	0.048071	0.066744			
7	0.019041	0.03261	0.055562	0.078539			
8	0.018765	0.034616	0.055527				
9	0.019092	0.038693	0.069339				
10	0.019958	0.040864	0.080357				
11	0.020007	0.044995	0.078552				
12	0.020406	0.048105					

Note that performance generally increases with both parameters, indicating that MWF and MCPDF are valuable techniques for selecting features.

A caveat about feature selection

Unfortunately, we did not realize until very late in the project that we were performing feature selection at the wrong stage of the training/testing process. We prune the feature set with respect to the entire dataset, then separate the dataset into a training and test set. Instead, we should have have separated the dataset into training and test sets *first*, and pruned the feature set only with respect to the train set. This ensures that we are testing on examples for which we have absolutely no prior knowledge.

The consequence of this oversight is that, for very high values of MCPDF, we are essentially eliminating all terms except those which *happen* to only show up in one class in our dataset. This would grant our classifier a large advantage because, when calculating conditional probabilities for each term in a test example, it will be much more likely (and perhaps ONLY have the choice) to classify the example correctly. It is as if we are doing a certain 'learning' over the entire data set with this pruning.

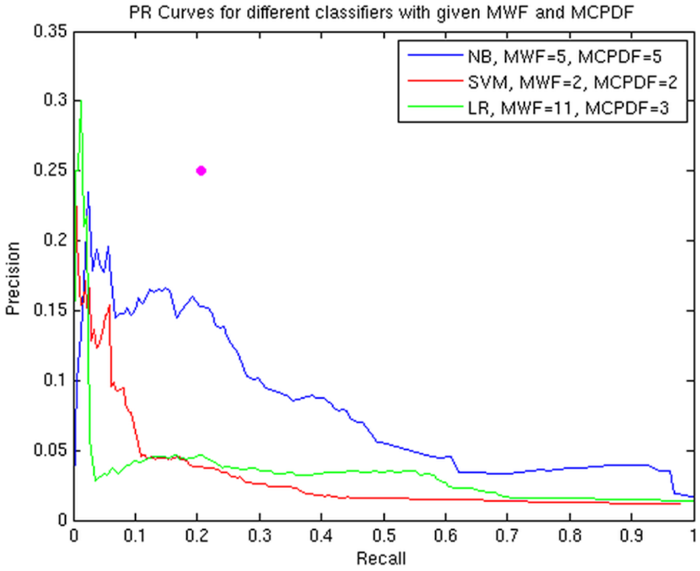
We acknowledge that, rather than feature pruning on the entire dataset and *then* doing k-fold cross validation, we should do the pruning *within* the k-fold cross validation, over only the portion of the data we are training on, so as not to implicitly learn anything unfairly about the testing data set. Unfortunately, this change would consist of a fair bit of restructuring of our entire process, as we would have to move the k-fold cross validation logic out into our python script. We tried putting the pruning logic in our Matlab code, but without data structures such as maps, our pruning step over such an enormous input matrix ran oppressively slowly.

The purpose of including this discussion at such length in this report is that we want it to be understood that we recognize the unfortunate implications of the way we implemented feature pruning, and that we acknowledge that the steps should have been done in a different order. However, we do try to avoid exploiting any significant artificial performance gains resulting from this error by not using extremely large values for this MCPDF. In doing so, hopefully we reap the intended benefits of this pruning technique (i.e., reducing noise stemming from features with similar conditional class probabilities) without reaping the unearned advantage described above. However, it is difficult to know how much of the performance gains in our findings and results are due to calming of noise, and how much are due to this mentioned implicit learning on the entire data set.

Experiments with other Algorithms

In addition to Naive Bayes, we applied a linear SVM and a logistic regression classifier toward our problem. Unfortunately, neither yielded promising results. The SVM exhibited poor performance for low values of the MCPDF and MWF parameters. When we used very high values for MCPDF, we achieved near-perfect performance. However, these MCPDF values corresponded to extremely limited feature sets, which lead to spurious performance results as discussed in the caveat above. In fact, these were the results which first really made us aware of this issue. Furthermore, unfortunately the SVM is not more honed, because we believed ourselves to be making forward progress with the NB classifier, but didn't realize until too late that the pruning we were doing, and the gains we were seeing, may not have been valid.

The following graph shows the performance of our three classifiers with values for MWF and MCPDF that work toward optimizing performance without over-aggressively pruning the feature set. While none of the three exceed the level of performance corresponding to human classification (as indicated by the magenta dot), Naive Bayes comes fairly close and all three perform better than random guessing, which would be represented by a horizontal line with precision equal to the positive class prior, or 0.011.



Classifier	MWF	MCPDF	AUC	Size of Vocab
NB	5	5	0.0800	994
SVM	2	2	0.0306	6302
LR	11	3	0.0335	701

● Human Performance (see below)

Difficulty of the Problem / Conclusions

One very important thing to keep in mind is that classifying YouTube comments is inherently a very difficult problem. Even a human, given an unlabeled comment, cannot consistently predict whether it would be downvoted heavily. Thus, it is unrealistic to expect this algorithm to perform as well as in canonical text classification problems such as spam classification.

To illustrate the difficulty of the problem, we estimated human classification performance by subjecting ourselves to the task of classification. We had a script give us a set of n random examples, where a certain percent of the examples (between 5 and 10 percent) were positive examples. The reason for making sure to give such a percentage of positive examples was that, as we are humans, we could not go through all 30,000+ comments looking for positive examples, and it was likely that a randomly chosen 100 or 200 would not contain *any* positive examples otherwise, because of the skew of the data. Note that this meant that the probability of seeing a positive example was *much* higher for our human testing - if anything this should give our human precision results an advantage. However, we still only saw low numbers for precision and recall (an average of .25 precision and .2063 recall).

Another factor that contributes to the difficulty of this problem is that whether a comment will be hidden is not merely dependent upon the content of the comment. After all, human voting behavior is nondeterministic - a given person on a given day may or may not decide to downvote a comment, regardless of how inflammatory the comment is. Thus, the exact same comment posted in response to the same video at two different times might be hidden in one case but not the other.

Although our classifiers did not exhibit stellar performance, they could still serve a purpose in real-world application. YouTube is clearly not likely to block comments based solely on an algorithmic prediction. Given the difficulty of accurately classifying comments, it is important to err on the side of caution - i.e., high precision is more important than high recall. Thus, the service that the algorithm provides is not necessarily to classify comments outright, but to provide a signal of comment quality that could be employed in a more sophisticated vote-based hiding mechanism. For example, rather than specifying an absolute voting threshold below which a comment is hidden, YouTube could maintain a dynamic threshold for each comment based on the algorithm’s predicted label. This could result in truly bad comments being hidden more quickly, thus reducing exposure to low-quality comments and improving overall user experience.