Sentiment Analysis of Tweets using Machine Learning

Michael Lumetta
Swarthmore College
500 College Avenue
Swarthmore, PA
mlumett1@swarthmore.edu

Christopher Miller
Swarthmore College
500 College Avenue
Swarthmore, PA
cmiller5@swarthmore.edu

Abstract

This paper describes a series of systems used to analyze the SemEval-2015 Task 10 Subtask B. As a whole, the system is a sentiment analyzer that uses a variety of machine learning algorithms including the decision list, support vector machine, and Naive Bayes algorithms. The data set being analyzed is a series of Tweets. Various methods of pre-processing are used to allow the machine learning algorithms to extract more useful information out of the data. Scores ranged from 34.68 to 54.34 depending on the combination of pre-processing and machine learning algorithms.

1 Introduction

Twitter is a rapidly growing medium of communication. The website has over 284 million monthly active users with more than 500 million Tweets sent per day. Each Tweet is a snapshot of what a particular user is thinking about at a given time. The sheer amount of data Twitter generates is a powerful tool. For companies, it could allow for near-instantaneous public sentiment feedback on their products. For political researchers during a presidential election, it could be used to understand how public sentiment towards one candidate or another changes over time. However, extracting that sentiment is not an easy task. While it is fairly simple to estimate sentiment by hand on a single Tweet, or even a few hundred Tweets, it is inefficient (if not impossible) to try to estimate sentiment on thousands or even millions of Tweets. Therefore, we have designed a system that uses machine learning to use extracted features from a sentiment-labeled training set of Tweets to estimate the sentiment of a test set.

2 System Architecture

Our system uses a variety of pre-processing methods and multiple machine learning algorithms in an attempt to find the one that is most successful in labeling the test set of Tweets.

2.1 Pre-processing

To show the effects of pre-processing on a Tweet, we will use the following example Tweet:

OBVIOUSLY, i am not a fan of Dream High 2 and i haven't even watched it! well, i watched the 1st ep,that's where i heard that song!

2.1.1 Case Folding

First, we use case folding to make all characters in all Tweets lowercase. This loses us some information with regard to proper nouns, but it allows us to properly count other words, regardless of their capitalization. For example, the word *OBVIOUSLY* and *obviously* are obviously the exact same word, but unless we fold their cases they will be treated as two entire distinct words. This could cause us to lose valuable information regarding word counts. Therefore, our example tweet becomes:

OBVIOUSLY, i am not a fan of Dream High 2 and i haven't even watched it! well, i watched the 1st ep,that's where i heard that song!

↓

obviously, i am not a fan of dream high 2 and i haven't even watched it! well, i watched the 1st ep,that's where i heard that song!

2.1.2 Negation Detection

Second, we used negation detection to detect when a user was flip the sentiment polarity of a statement. For example, a user saying they like something suggests a positive sentiment; however, simply adding the word *not* (they do *not* like something) flips the polarity entirely. To account for this, we attach NOT_ as a prefix to each word following the word *not* in a Tweet until the next punctuation mark. Therefore, our example tweet becomes:

obviously, i am not a fan of dream high 2 and i haven't even watched it! well, i watched the 1st ep,that's where i heard that song!

↓

obviously, i am not NOT_a NOT_fan NOT_of NOT_dream NOT_high NOT_2 NOT_and NOT_i NOT_haven't NOT_even NOT_watched NOT_it! well, i watched the 1st ep,that's where i heard that song!

2.1.3 Stopword Processing

Third, we had the option to use stopword processing. Stopwords are defined as the most common words in a corpus–usually words like *that* or *I* or *if*. While it isn't possible to say exactly which words ended up as stopwords–it is calculated as words whose frequency exceeded a threshold, which could change depending on the corpus–these words may not provide meaningful information and may therefore be discarded. However, in this particular data set, we found using stopword processing to be detrimental to our results. So, while the option to use stopword processing is there, it is not utilized in any of the final tests.

2.2 Features

After the Tweet corpus was pre-processed, a variety of features were extracted.

2.2.1 Word ngrams

The most basic form of feature extracted was word ngrams. A unigram is a single word in a tweet. An ngram is a group of *n* contiguous words in a Tweet. For example, in our example tweet

obviously, i am not NOT_a NOT_fan NOT_of NOT_dream NOT_high NOT_2 NOT_and NOT_i NOT_haven't NOT_even NOT_watched NOT_it! well, i watched the 1st ep,that's where i heard that song!

the words *1st* and *ep* are contiguous and, together, form a bigram. The bigrams of our example tweet are:

(obviously, i) (i, am), (am, not), (not, NOT_a), (NOT_a, NOT_fan), (NOT_fan, NOT_of), (NOT_of, NOT_dream), (NOT_dream, NOT_high), (NOT_high, NOT_2), (NOT_2, NOT_and), (NOT_and, NOT_i), (NOT_i, NOT_haven't), (NOT_haven't, NOT_even), (NOT_even, NOT_watched), (NOT_watched, NOT_it), (NOT_it, well), (well, i), (i, watched), (watched, the), (the, 1st), (1st, ep), (ep, that's), (that's, where), (where, i), (i, heard), (heard, that), (that, song)

We used unigrams, bigrams and trigrams in our feature extraction.

2.2.2 Skipgrams

An addition to ngram features is to ignore the "contiguous" constraint to ngrams. This allows for skipgrams, which are collections of non-contiguous words separated by a set number of steps. In our example tweet, we could have a skipgram

(obviously, am)

or

(am, NOT_a)

if the skip amount *k* for the skipgrams are set to 1 and the number of words *n* is set to 2 (a value of *k* = 0 would just be a bigram). We generally used a *k* value of 3 in our solutions.

2.3 Machine Learning

We implemented four machine learning classifiers: a decision list, a naive Bayes classifier, a support vector machine, and a k-nearest neighbors classifier.

2.3.1 Decision List Classifier

The decision list classifier is the simplest of the systems that we tried. For each feature f and class c, it calculates the maximum likelihood estimates of $P(f|c)$. The maximum likelihood estimate is given as follows:

$$P(f|c_i) = \frac{count(f|c_i)}{\sum_j count(f|c_j)} \quad (1)$$

For each feature, the class with the maximum score of $P(f|c_i)$ is stored with the feature name and score in a list, which is sorted on the score. Instances of test tweets are then classified as the class of their strongest feature in the decision list. This metric is simple but still reasonably powerful for classifying tweets.

2.3.2 Naive Bayes Classifier

Unlike a decision list, the naive Bayes classifier uses all features in a test instance to make a classification. The naive Bayes classifier calculates the probability of a tweet having a certain sentiment classification as follows. First, it calculates $P(c_i|f_j)$ for all sentiments i and all features j in the test tweet. This calculation is performed by a modified version of Bayes rule:

$$P(c_i|f_j) = \frac{P(c_i) * P(f_j|c_i)}{P(f_j)} \quad (2)$$

As in the decision list algorithm, $P(c_i)$ and $P(f_j|c_i)$ are calculated as maximum likelihood estimates by counting all occurrences of c_i and all occurrences of f_j in c_i. The class c_i that maximizes $P(c_i|f_1, ..., f_m)$ is chosen as a label for the test tweet:

$$\begin{aligned} label &= \arg\max_i P(c_i|f_1, ..., f_m) \\ &= \arg\max_i P(c_i) \prod_{j=1}^{m} P(f_j|c_i) \end{aligned}$$

The denominator can be omitted because we are taking the *argmax* of all classes. To prevent underflow, we used the sum of the log probabilities rather than a product of the real probabilities.

The naive Bayes classifier is "naive" because it assumes a conditional independence of features - that is, the presence of one token has no influence on the other tokens. Obviously, in natural language, this assumption does not hold, but the naive Bayes classifier still performs reasonably well.

2.3.3 Support Vector Machine

Support vector machines (SVM) are useful for classifying text because they tend to still be effective when the number of features d, or the dimension of a feature vector, is high relative to the number of training instances n. The SVM divides the feature space in d dimensions with hyperplanes in $d - 1$ dimensions. For our SVM implementation, we used *scikit-learn*, an open-source machine learning framework for Python. *Scikit-learn* provides a LinearSVC class that implements an SVM, which we used with their CountVectorizer class to generate feature vectors.

2.4 K-Nearest Neighbors

K-nearest neighbors is generally used as an unsupervised clustering algorithm. However, it can also be used as a supervised classifier. The k-nearest neighbors algorithm assigns each tweet to the majority class of its k nearest neighbors, computed by some distance function. We implemented this algorithm using *scikit-learn*'s KNeighborsClassifier. We tried versions where the neighbors were weighted uniformly and where they were weighted according to the inverse of the distance from the tweet being classified.

3 Results

After we had processed our corpus, extracted relevant features, and built the machine learning algorithms, we combined them all in different ways with the goal of finding the most successful in tagging the test corpus. For each algorithm, we performed 5-fold cross-validation on the training data. The results in Table 1 are the averages of the official SemEval scores over the 5 cross-validation sections.

3.1 Description of Algorithms

1. Decision List 1: A decision list improved by case folding and negation processing. No stopwords. Features: bag of words.

2. Naive Bayes 1:Naive bayes classifier with case folding and negation processing, no stopwords. Features: bag of words, bigrams, trigrams, and 3-skipgrams.

3. Decision List 2: A decision list improved by case folding and negation processing. No stopwords. Features: bag of words, bigrams, trigrams, and 3-skipgrams.

4. Decision List/Naive Bayes Hybrid: A combination of algorithms 1 and 2. Breaks ties by

choosing the non-neutral result, if there is one, or the decision list.

5. Decision List 3: An extended decision list algorithm using the k most significant features as opposed to just the most significant. Features: bag of words, bigrams, trigrams, and 3-skipgrams.

6. Decision List 4: Same as algorithm 5 but uses bag of words as features.

7. Support Vector Machine: An SVM approach using LinearSVC from scikit-learn. For feature vectors, uses count vectors for bag of words (implemented by CountVectorizer).

8. Weighted K-Nearest Neighbors: Uses KNeighborsClassifier from *scikit-learn* to implement k-nearest neighbors. For feature vectors, use count vectors for bag of words. Weights neighbors by inverse of distance.

9. Uniform K-Nearest Neighbors: Same as algorithm 8 but does not weight neighbors.

10. SVM/Decision List Hybrid: A simple voting scheme between the SVM and decision list outputs. Breaks ties first by choosing the non-neutral result, if one exists, and second by choosing the SVM.

11. SVM/Decision List/Naive Bayes Hybrid: A voting scheme that gives each of the three algorithms a weighted vote. The weight is calculated by taking that algorithm's score by itself and normalizing it to 1.

3.2 Discussion

We found the best algorithms to be the support vector machine and, interestingly, the decision list. It makes sense that the support vector machine performed well. Given that it is a robust algorithm for high-dimensional spaces and that it does not make the same assumptions as naive Bayes or the decision list, we expected that it would perform well.

Decision list, however, was surprising. Because decision list focuses on the strongest feature rather than all features, we expected naive Bayes to outperform the decision list. Interestingly, the presence of many features seemed to dilute their usefulness. Also, decision lists performed better with just the strongest feature being used to select a class and with just the bag of words rather than bigrams, trigrams, or skipgrams. An interesting approach we did not try would be to use the naive Bayes algorithm on a subset of strong features, with strength being determined by the decision list.

Finally, we tried multiple voting schemes with the SVM. None of them proved to significantly enhance our score. Algorithm 11 achieved a marginally better score using weighted decision list and naive Bayes in addition to the SVM. An interesting area of further exploration would be what additional classifiers could make a more robust ensemble classifier.

Algorithm	**Score**
Decision List 1	46.02
Naive Bayes 1	34.68
Decision List 2	40.83
Decision List/Naive Bayes Hybrid	44.67
Decision List 3	42.19
Decision List 4	37.54
Support Vector Machine	54.34
Weighted K-Nearest Neighbors	29.58
Uniform K-Nearest Neighbors	23.83
SVM/Decision List Hybrid	54.32
SVM/Decision List/Naive Bayes	54.68

Table 1: Table of results over 11 experiments. Each score is the official SemEval score averaged over 5-fold cross-validation.

4 Conclusions

We proposed a variety of methods for extracting sentiment from Tweet data. We found that the Support Vector Machine was by far the most successful machine learning algorithm. Case folding and negation processing were useful pre-processing methods while stopword processing was not. Which features were most useful depended on multiple factors– some combinations worked better than others. In the case of Decision List 1, a simple bag of words was more effective than any other combination of features. While a score of 54.34 is not particularly high compared to the historical average for this task, it is significantly above the baseline (choosing the most

frequent sentiment).