Sentiment Analysis

Nakul Dawra, Nikola Kovachki, Alex Lew,
Walker Mills, Xiang Ni

April 20, 2015

1 Introduction

The Rotten Tomatoes movie review corpus is a collection of movie reviews collected by Pang and Lee in [2]. This corpus has been analysed in [3] where each sentence is parsed into its tree structure and each node is assigned a fine-grained sentiment label ranging from $1-5$ where the numbers represent very negative, negative, neutral, positive and very positive respectively. In this paper we use this data on an assortment of machine learning algorithms in an attempt to train a computer to accurately perform sentiment analysis. The corpus consists of about 150,000 phrases and all the methods in this paper are assessed by training on a random subset of phrases (and their subphrases) of size approximately $4/5$ of the data set and testing using the remaining $1/5$. The best known fine-grained sentiment analysis algorithms are able to perform with an accuracy of 80.7% (see [3]). Unfortunately no methods attempted in this paper are able to come close to this.

2 Bag of Words Approach

The main difficulty in running machine learning algorithms on natural language text is being able to represent it mathematically. There is a fairly naive and well documented approach to doing this known as Bag of Words. The idea is quite simple. We start with the dictionary of words in our language $\mathscr{D}$ and pick a subset $D = \{w_1, w_2, ..., w_n\} \subset \mathscr{D}$ to use as our effective dictionary. Since most languages are quite extensive in their vocabulary, we have $|D| \ll |\mathscr{D}|$. We will assume the data we want to analyze is given in sentences as the set $S = \{s_1, s_2, ..., s_k\}$ where each $s_i \subset \mathscr{D}$ is a finite multiset with counting function $m_i : s_i \longrightarrow \mathbb{N}_{\geq 1}$. S is known as the training set. Ideally we want our effective dictionary to contain almost all of the words in our sentences, but this is not strictly necessary. In fact, for practical purposes of computation, this will usually not be the case, and we will often even have that n is an order of maginitude lower than k. Now for a definition.

Definition 1. *The map* $\phi : S \times D \longrightarrow \mathbb{N}$ *defined by*

$$\phi(s, w) = \begin{cases} m(w) & : w \in s \\ 0 & : w \notin s \end{cases}$$

is called the counting bag of words function. If we replace $m(w)$ *with some constant* $c \in \mathbb{R}$ *in the above, then we will denote the map* ϕ_c *and call it the existence bag of words function.*

From now on, unless stated otherwise, if we refer to the bag of words function, we mean the counting bag of words function. This gives us just enough tools to represent each of our sentences numerically. For some $s \in S$, we can easily compute the sparse vector

$$(\phi(s, w_1), \phi(s, w_2), ..., \phi(s, w_n)).$$

We will call this a feature vector because, in machine learning, each dimension of the training data is called a feature in the model. This name motivates our next definition.

Definition 2. *The set* $\mathscr{F} = \{(\phi(s, w_1), \phi(s, w_2), ..., \phi(s, w_n)) : \forall s \in S\} \subset \mathbb{N}^n$ *is called the bag of words feature vector set of the training data* S *w.r.t. the dictionary* D*. In general, we will call any mathematical representaions of the sentences in* S*, the feature vector set* $\mathscr{F}$*.*

Hence each word in the sentence becomes a feature, so we weight the overall sentiment of the sentence based only on the sentiment of its words. This approach does not take into account the grammatical structure of the sentance or the meaning of interactions betweens it phrases. We can think of it as a first order approximation of sentiment because it only factors in the leaf nodes of the sentence's parse tree (i.e. the single words).

The above definitions outline a clear computational approach for the bag of words method. We need only obtain the feature vector set along with some pre-determined sentiments for our data and the machine learning algorithms will do the rest. We discuss the computational results from this method, in the subsequent sections which deal with our machine learning algorithms.

A natural extension to this simplistic model is a second order approach, where we extend our dictionary to also include phrases of length greater than one. This is much more computationally difficult, because we must first determine which parts of the sentence constitute phrases, and then determine how many need to be added to our dictionary. Neither of these issues can be ignored, since computation will become infeaseble if no effort is made to prune extraneous phrases from the dictionary. Consider a dictionary of size n. Adding only two- and three-word phrases extends its size to $n + \binom{n}{2} + \binom{n}{3}$, which for $n \sim 1000$ increases dictionary size by five orders in magnitude. Hence we must determine a method of choosing the phrases which contribute most to the setiment of our sentence, and thus have the biggest impact on their neighboring phrases, while ignoring ones that can stand on their own. If we could ignore

computational constraits, and perhaps sometime in the future problems in EXP will be tractable, this method can obviously be extended to take into account all possible combinations of words in a sentence, whereby a m-word sentence would produce $2^m - 1$ features capturing all possible linguistic interactions within the sentence.

3 Bag of phrases modified approach

One of the key weaknesses of the standard bag of words approach is that, when combined, words can have very different meanings. For instance, consider the exampe of the modifier "not". "Not" simply negates the meaning of the following word; the sentiment of the phrase (*not*, _____) can be either positive or negative; not itself shares no meaning about the sentiment of the phrase. The bag of words approach does not allow for wildcards and pattern matching, so the strategy developed here is an approximation of the meaning associated with two word phrases. The intention is to capture some essence of the modifier structure by considering the most popularly used two word phrases; possible examples are "not good", "strangely appealing", etc.

The modification to bag of words is as follows. If the set of all words appearing in the training data is W, then the possible elements in the dictionary is the set $W \cup (W \times W)$. In order to build our dictionary, we take the elements that appear most frequently in the training data out of this set.

Specifically, the methodology was the following:

1. Identify the words that appear the most commonly. This is simply the single word approach to bag of words.

2. Identify the modifiers. To find the most common modifiers, create a counter to identify the most common words used in the neighborhood of the most popular words and words with the strongest sentiment. The idea here is that the words that appear the most often or have the strongest sentiment either way will be qualifiers and thus the words that appear around them most often will include the set of modifiers. This is not an exact way of identifying modifiers, but the set of modifiers should be a subset of the set identified in this way.

3. Find the two word phrases that include a word from the possible set of modifiers that was previously identified. Clearly, modifiers can be applied to any range of words, but many qualifiers will be fairly common. Thus, it is a reasonable expectation that the two word phrases featuring a modifier might appear again in other data, and so we include these in the dictionary that we eventually use to train our bag of words classifier.

4. Build our bag of words by adding single words and two word phrases until the dictionary contains the desired number of elements.

We see that this approach helps capture the most used modifier-qualifier pairs in our dictionary. The potential downside of this approach is that the two word phrases will clearly appear less frequently in any test data than the one word phrases that compose the two word phrase. So, this approach may result in the exclusion of words from the dictionary which are more relevant that the included two word phrases. Below is the histogram of the occurrences of the 10 most frequent words vs. the occurrences of the 2-word phrase containing that phrase.

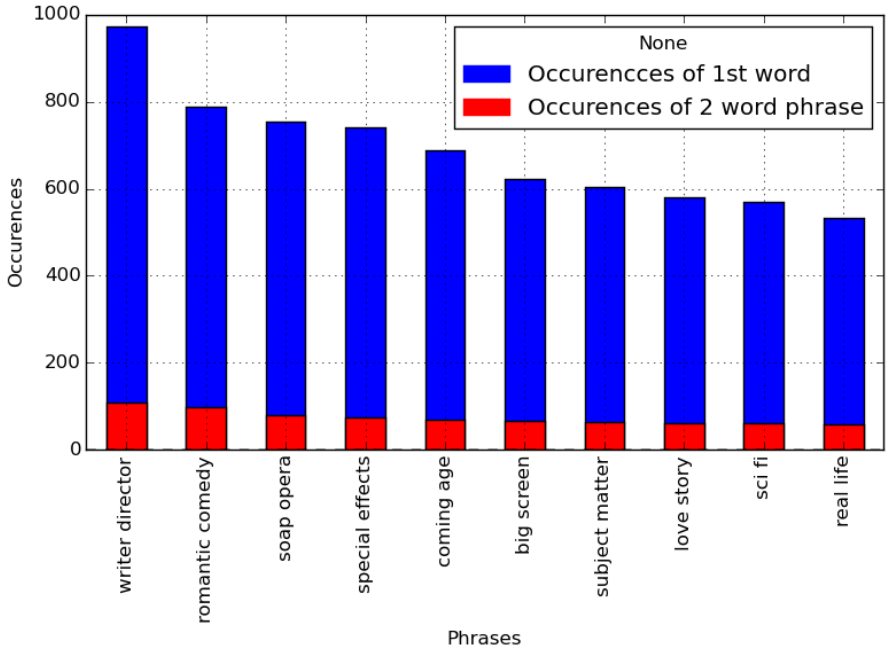


Figure 1: Frequency of 1 word vs. 2 word phrase

The frequency of the most frequent two-word phrase is clearly much lower. This is to be expected since the number of times a two word phrase including a word appears must necessarily be a fraction of the times the single word appears. However, we see that the proportion of the time that the two word phrase appears is roughly 5% to 10% of the times the single word appears. So, there is some justification for the two-word phrases being included in larger dictionaries of size 1000 to 10,000 but overall, not many of the two word phrases seems to appear often enough to justify their inclusion in the dictionary.

After training on our training set, we found the following values for E_{in} and E_{out} on our validation set.

These results are lower than the E_{in} and E_{out} that the one-versus-all clas-

Dictionary Size	1 Tree	5 Trees	10 Trees
10	.4964	.4959	.4962
1000	.5138	.5213	.5203
2500	.5074	.5201	.5245

Figure 2: Bag of Phrases

sifier implementation of bag of words produced, and thus we can conclude that unless we find a better way to identify important 2 word phrases, the inclusion of 2-word phrases in the dictionary does not improve performance.

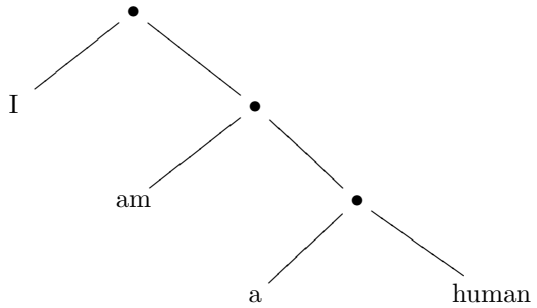
4 Adding the Tree Structure

A unique feature of the corpus of data provided is that sentiments are associated to each phrase and word in the tree structure of the sentences. In other words, each sentence is fully parsed into a tree with a "fine-grained" (1-5) sentiment associated with each node. One way to perform machine learning on data of this type is to construct a recursive neural network or RNN. We will now describe a toy version of this model. Assume for now, that all semantic trees are binary.

Suppose we are given a parse tree T with root r. We would like to construct a function S so that $S(T)$ returns a 5 dimensional real valued vector whose ith entry is the probability that the phrase r has sentiment i. If T consists of one node (i.e. is just one word) then $S(T)$ is defined from the test data given. For T consisting of more than one node we can define S recursively in terms of another function g. Let $g : \mathbb{R}^{10} \to \mathbb{R}^5$ and let;

$$S(T) = g(S(T_1), S(T_2)),$$

where T_1 and T_2 are the semantic subtrees of T whose roots are children of r. For example, consider the sentence, "I am a human" whose tree structure is shown below.



Then for some fixed choice of g the sentiment associated to this sentence is;

$$g(S(\text{ "I" }), g(S(\text{ "am" }), g(S(\text{ "a" }), S(\text{ "human" })))).$$

To find an appropriate g using a RNN, we pick a class of g which depend on some set of parameters $\Theta = (\theta_1, \ldots, \theta_n)$. Then we find Θ which minimizes the the error obtained by using g on every phrase of the test set (with respect to some cost function.)

Using a RNN type model for sentiment analysis is computationaly intensive, and has already been done extensively for the data set used in this paper (see [3]). Another, much less computationally intensive idea, is to somehow use the tree structure in the data to modify the feature vectors used as inputs in a "bag of words" type algorithm. Given an effective dictionary D, and a phrase with semantic tree T, we want to construct a function F which associates to each parse tree a vector in $\mathbb{R}^{|D|}$. For words, F will simply assign the i'th word in D to the i'th elementary vector in $\mathbb{R}^{|D|} = \mathscr{F}$. If T consists of more than one word, then we define F in terms of another function $f : \mathscr{F}^k \to \mathscr{F}$ as;

$$F(T) = f(F(T_1), F(T_2), \ldots, F(T_k))$$

where the T_i are subtrees whose roots are the children of the root of T and k is an upper bound on the number of children of any node.

Note that the function f is simply a choice, and we are optimizing to compute an ideal candidate for the function f. For any choice of f, F will generate vectors in $\mathscr{F}$ on which we can train. If we chooce f to simply be addition (i.e. f will simply add all the inputs together) then the feature vectors generated by F will be the same as those generated by the usual bag of words. If we want to take into account the tree structure at all, then our choice of f must be non-associative.

5 Computational Methods with Tree Structure

To use the methods of tree structure described above, we need an appropriate non-associate function and a method of appling it to the tree structures.

Lemma 1. *The function $f : \mathscr{F}^2 \longrightarrow \mathscr{F}$ defined by $f(V_1, V_2) = \frac{1}{2}(V_1 + V_2)$ is non-associative.*

Proof. A simple calculation shows,

$$f(f(V_1, V_2), V_3) = f(\frac{1}{2}(V_1 + V_2), V_3) = \frac{1}{4}(V_1 + V_2) + \frac{1}{2}V_3$$

$$f(V_1, f(V_2, V_3)) = f(V_1, \frac{1}{2}(V_2 + V_3)) = \frac{1}{2}V_1 + \frac{1}{4}(V_2 + V_3)$$

So indeed $f(f(V_1, V_2), V_3) \neq f(V_1, f(V_2, V_3))$ $\forall V_1, V_2, V_3 \in \mathscr{F}$ □

Lemma 2. *Let f be defined as in the previous lemma and B a sentence represented by a balanced binary tree of depth k (0-indexed). Then f recusevely applied to B produces the same feature vector as the existence bag of words function $\phi_{\frac{1}{2^k}}$.*

Proof. Let $V_1, ..., V_{2^k}$ be the leaf nodes of B, representing the identity feature vectors corresponding to the words in the sentence. Define $f_i^{(n)}$ to be the i-th step in the n-th reccursion of the application of f to B. Then since B balanced and binary, we have

$$f_1^{(1)} = f(V_1, V_2), ..., f_{2^{k-1}}^{(1)} = f(V_{2^k-1}, V_{2^k})$$

$$f_1^{(2)} = f(f_1^{(1)}, f_2^{(1)}), ..., f_{2^{k-2}}^{(2)} = f(f_{2^{k-1}-1}^{(2)}, f_{2^{k-1}}^{(2)})$$

$$\vdots$$

$$f_1^{(i)} = f(f_1^{(i-1)}, f_2^{(i-1)}), ..., f_{2^{k-i}}^{(i)} = f(f_{2^{k-i+1}-1}^{(i-1)}, f_{2^{k-i+1}}^{(i-1)})$$

$$\vdots$$

$$f_1^k = f(f_1^{(k-1)}, f_2^{(k-1)})$$

Evaluating these at each step, it is easy to see that

$$f_1^k = \frac{1}{2^k} \sum_{i=1}^{2^k} V_i$$

□

While the above Lemma is not particularly deep, its proof gives insight into a computational method for the feature vectors. We need only compute the feature vectors with the existence bag of words function ϕ_1 which is very is easy to do as we saw in the first section. Then we compute the depth, d, of each word of the sentence in its tree representation and divide the instance of the word in the feature vector by 2^d.
Using this function f to generate feature vectors for the random forests learning method generates the following results.

Figure 3: Accuracy percentage when $f(V_1, \ldots, V_k) = (V_1 + \ldots + V_k)/2$

Dictionary Size	1 Tree	5 Trees	10 Trees
1000	.511736	.517557	.520468
2000	.511230	.522745	.524454

6 Random Forests

The random forest classifier is a machine learning approach which builds decision trees based on random samples of the data. It is a basic and quite well understood approach for learning on large sets of data. Consider training data

with N feature vectors all with dimension k. Then the depth of each decision tree is $k+1$ where at each step we branch based on the feature and have the last node of the tree (its final leaf) be the assigned output, in our case, the sentiment. The machine learns based on minimizing a pre-determined threshold function for the given tree. As usual, we let $\mathscr{F}$ be the set of feature vectors and $\mathscr{Y}$ the set of outputs (sentiments), then the random forest algorithm works as follows:

1. Pick a number $T \in \mathbb{N}$ representing the desired number of trees.
2. For each $t \in \{1, ..., T\}$ uniformly sample N times from the data with replacement, creating samples (F_t, Y_t) where $F_t \subset \mathscr{F}$ and $Y_t \in \mathscr{Y}^N$.
3. For each pair $(F, Y) \in \{(F_t, Y_t)\}_{t=1}^T$ train a decision tree which outputs a predictor function $f_t : \mathscr{F} \longrightarrow \mathscr{Y}$.
4. Average over each predictor function to find our our final predictor $\hat{f} = \frac{1}{T}\sum_{t=1}^T f_t$

Running this algorithm on the standard bag of words feature vectors we obtain the following results.

Dictionary Size	1 Tree	5 Trees	10 Trees
1000	.4268	.4165	.4325
2500	.4753	.4766	.4956
5000	.4988	.5033	.5099

Figure 4: Random Forest applied to bag of words

7 Support Vector Machines

A linear support vector machine is type of binary classafier which tries to separate a data set into two categories with a maximum-margin hyperplane. For each $x \in \mathscr{F}$, a general feature vector set, we have an associated binary classification $y \in \{-1, 1\}$. We define our training set,

$$\mathscr{T} = \{(x_i, y_i) : x_i \in \mathscr{F}, y_i \in \{-1, 1\}\}$$

Then a hyperplane which statisfies the points of $x \in \mathscr{T}$ is defined as

$$w \cdot x - b = 0$$

We want all data points to fall outside of the margin of the hyperplane, so we impose the condition

$$y_i \cdot (w \cdot x_i - b) \geq 1$$

$\forall (x_i, y_i) \in \mathscr{T}$. So the problem becomes

$$\arg\min_{(w,b)} \frac{1}{2}\|w\|^2$$

under the above constratint. We multiply by $\frac{1}{2}$ take and the square of the L^2-norm to make the problem more easlily solvable by dynamic programming packages, as this has no effect on the minimum values.

This type of classifier is not a natural choice for sentiment analysis, as it is binary, while our values for sentiment are not. However, we can easily extend it to suit our purposes by using a multi-class SVM. In this approach, we create five x vs. all linear SVM classifiers where $x \in \{0, ..., 4\}$ is a sentiment. We change the data, to set all sentences with sentiment x to 1 and all other ones to -1. This essentially gives a "reject-accept" type classifier where if we predict a sentence to have output value 1 then it has sentiment x and otherwise it does not. Then we simply predict the sentiment of each sentence with all five classifiers to produce a final sentiment (if two or more classifiers accept then we uniformly pick one of them). Due to the huge computational overhead of this method, we were only able to run it once on a bag of words feature set with dictionary size 2000. The results are summarized below.

Classifier	0 v All	1 v All	2 v All	3 v All	4 v All	Overall
Accuracy	.952468	.820777	.550867	.783278	.937989	0.484888

Figure 5: Multi-class SVM classifier on bag of words

8 Logistic Regression

Logistic regression is a type of probabilistic statistical classification model. Generally, it is well suited for describing and testing hypotheses about relationships between a categorical outcome variable and one or more categorical or continuous predictor variables.

Define the logistic function as

$$\sigma(t) = \frac{1}{1 + e^{-t}}.$$

When using logistic regression, we model the conditional probability as:

$$p(y|x, \theta) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}},$$

Where x is the feature vector and y is the response and θ are the parameters we wish to learn.

Figure 6: Accuracy (%) of Logistic Regression

Dictionary Size	L_2 regularization	L_1 regularization
500	.496046	.495286
1000	.479026	.477950
2000	.620031	.460171

Thus we wish to maximize the following

$$\text{argmax}_\theta \sum_i \log p(y_i|x_i,\theta) - \alpha R(\theta),$$

where $R(\theta)$ is the regularization term,which forces the parameters to be small (for $\alpha > 0$). Generally, there are two poplular regularization methods which are called L_1 and L_2 regularizations. They takes the following forms:

$$L_1: \quad R(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$$

$$L_2: \quad R(\theta) = \|\theta\|_2^2 = \sum_i \theta_i^2$$

In our case the reason we used regularization is to avoid overfitting by not generating high coefficients for predictors that are sparse.

We run the logistic regression with different regularization methods (L_1 and L_2) on the bag of words feature vectors and summarize the results in Figure 6. In general, it is believed that L_1 regularization helps perform feature selection in sparse feature spaces. However, even though in our case the feature vectors are highly sparse, L_1 does not perform better than L_2 as shown in Figure 6.

9 Adaboost Classifier

AdaBoost is one of the most common ensemble algorithms. It basically combines several weak classifiers (e.g. a single split in Decisoin Tree) with different weights. One advantage of this algorithm is that it will improve the accuracy of weak learning models without over-fitting the data. Therefore, in our project, we combine Decision Tree with AdaBoost to train our learning model. The parameter we manipulated with AdaBoost is the number of estimators.

Given $(x_1, y_1), ..., (x_m, y_m)$, where $x_i \in X$ is the feature vectors and $y_i \in Y$ is the response., the Adaboost algorithm is as follows ([4]):

Step 1: Initialize $D_1(i) = 1/m$. For $t = 1, ..., T$:

- Train base learner using distribution D_t.
- Get base classifier $h_t : X \to \mathbb{R}$
- Choose $\alpha_t \in \mathbb{R}$.

Figure 7: Accuracy (%) of Adaboost Classifier

Dictionary Size	25 estimators	50 estimators
500	.494907	.496488
1000	.489718	.488263
2000	.493515	.489845

• Update:

$$D_{t+1}(i) = \frac{D_t(i)exp(-\alpha_i y_i h_t(x_i))}{Z_t},$$

where Z_t is a normalization factor (chosen so that will be a distribution)

Step 2: Output the final classifier.

We run the AdaBoost classifier using 25 and 50 estimators on the bag of words feature vectors and summarize the results in Figure 7. As the results show that the prediction accuracy is more or less the same for different dictionary sizes and estimators.

10 Gradient Boosting (GB)

In addition to AdaBoost, Gradient Boosting is the another ensemble algorithm that can be used on both classification and regression. It works like gradient descent in function space of weak classifiers, each classifier being a shallow Decision Tree in this case. The idea is to minimize the loss function along an axis of the function space at each iteration. The parameters with Gradient Boosting are the number of estimators, the learning rate, and the maximal depth of each estimator.

Input: Let $\{(x_i, y_i)\}_{i=1}^n$ be the training set. We seek to to minimize a differentiable loss function $L(y, F(x))$. Let M be the number of iterations.

The Gradient Boosting algorithm is as follows (by wikipedia):

Step 1: Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

Step 2: For m = 1 to M:

•: Compute pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$

•: Fit a base learner $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^n$.

•: Compute multiplier γ_m by solving the following one-dimensional

Figure 8: Accuracy (%) of Gradient Boosting

Dictionary Size	25 estimators	50 estimators
200	.496046	.496172
500	.497121	.497501
1000	.495223	.495919

optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

•: Update the model:

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$

Step 3: Output $F_M(x)$.

We run the Gradient Boosting algorithm using 25 and 50 estimators on the bag of words feature vectors and summarize the results in Figure 8. As the results show that the prediction accuracy is more or less the same for different dictionary sizes.

11 Conclusion

The goal of this paper was to identify optimizations and other approaches to sentiment classification of movie reviews. The approaches focused on attempting to improve accuracy of bag of words by appling concepts from linguistics, while incurring minimal computational overhead. One of the methods tested was to use the same model as bag of words, but include two-word phrases as well. This was designed to capture some notion of semantic meaning, in the form of common two-word phrases, theorized to contain comnmon negations and/or modifiers. This approach did not improve the accuracy of bag of words, most likely because the two word phrases didn't generalize well to other data sets. The next approach we tried was to cheaply incorporate tree structure into the bag of words feature vectors by scaling the identity vectors representing each word by the depth of the parse tree where that leaf node occurred. However, this approach, too, failed to improve the results of bag of words significantly. Finally, we tried a variety of classifiers, in an attempt to determine whether the choice of classifier effects the accuracy of the bag of words model. We found that the choice of classifier had a negligible effect on the accuracy of bag of words, likely becuase much of the information essential for sentiment analysis is lost by the first-order approximation.

References

[1] Abu-Mostafa, Yaser S., Malik Magdon-Ismail, Lin, Hsuan-Tien. Learning from Data: A Short Course. *United States: AMLBook.com*, 2012. Print.

[2] Bo Pang , Lillian Lee. Seeing stars: exploiting class relationships for sentiment categorization with respect to rating scales, *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, p.115-124, June 25-30, 2005, Ann Arbor, Michigan

[3] Socher, Richard, Perelygin, Alex, Wu, Jean Y., Chuang, Jason, Manning, Christopher D., Ng, Andrew Y., and Potts, Christopher. Recursive deep models for semantic com- positionality over a sentiment treebank. In *Conference on Empirical Methods in Natural Language Processing* , 2013b.

[4] Robert E. Schapire. The Boosting Approach to Machine Learning: An Overview. In *Nonlinear Estimation and Classification*, Springer, 2003.