Taylor Penner

Jonathan Stroud

Jacobus Harding

Probabilistic Sentiment Analysis of Movie Reviews

CS 175 Final Report (Group 5)

Introduction

In this project, we perform sentiment analysis on a subset of the IMDB movie review database using a number of probabilistic models. *Sentiment analysis* is the task of identifying from a piece of text some implicit information about its content. We approach this as a supervised learning problem, where we are given a large corpus of reviews, each labeled with a score between 1 and 10. When given a review, our models extract some salient features from its text, and then use this information to predict its rating. Our models are trained to make these predictions in a way that gives the best possible performance on the training corpus, and are then validated using a test corpus which our models never see during training.

We distinguish between two types of prediction tasks, *classification* and *regression*. In classification, we predict a binary variable representing the polarity of the review. In our labeled training data, reviews with a rating in the range 6 - 10 are considered “positive” and reviews with a rating in the range 1 - 5 are considered “negative”. We approach this binary classification problem using a number of common machine learning algorithms, such as Naive Bayes and Support Vector Machines, which are detailed in future sections. For regression, we make a real-valued prediction in the range [1, 10] using a form of logistic regression. We compare our models with various baselines and obtain near state-of-the-art performance.

In addition to these prediction tasks, we also build probabilistic generative models from which we can sample random positive or negative reviews. This proves to be an incredibly challenging task due to the complex structure present in natural language, but we see some success using a Markov Chain-like model. We also explore probabilistic context-free grammars as a generative framework, but initial experiments led to disappointing results. In the coming sections, we show some examples generated by each of these models and attempt to explain their strengths and weaknesses.

Related Work

Previous work on sentiment analysis has explored the use of many classical machine learning techniques to perform regression and classification tasks with textual input on data. Examples of related tasks might be to infer a student’s grade on an essay from its text, to determine how much a customer would be willing to pay for a product based on a written description, or to predict public opinion about a current event based on twitter posts (Pang and Lee, 2008). All of these tasks can be placed within a supervised learning framework if we are given a sufficient number of labeled examples.

A key component in all of these models is *feature selection*, where information is extracted from the text of a document and converted into a feature vector that can be effectively used to predict the target variable. This is particularly difficult to accomplish with text data because of the highly complex structure present in natural languages. A method commonly used in natural language processing is the bag-of-words representation, but other more complex approaches make use of bi- and tri-grams. Some sophisticated models use a hierarchical feature extraction approach using deep neural networks, with significant success (Bollen, Mao, and Zeng 2011). We make use of some of these approaches with our models, and explore the effectiveness of each on our dataset. Once features are extracted, we can use almost any off-the-shelf machine learning algorithm to perform sentiment analysis.

The most sophisticated approaches currently in use today make use of a generative probabilistic model, which models the text of a review with a graphical model similar to Latent Dirichlet Allocation (Blei, Ng, and Jordan 2003). This is a particularly interesting approach because it could allow us to take advantage of the generative structure to potentially sample unique reviews with a given sentiment . In this project, we use our own Markov Chain-like probabilistic model to make steps towards accomplishing this complex task. The models we develop for classification come very close to exceeding the state-of-the-art performance exhibited by this probabilistic model. When this model is used for classification, it performs with 89% accuracy, while our best model achieves an impressive 88.1% accuracy (Maas et. al, 2011).

Data Used

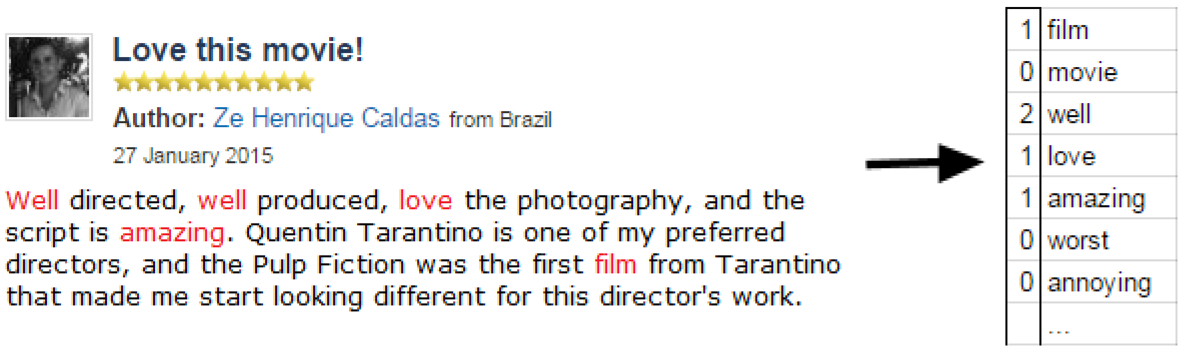
Our training and test datasets comes from the IMDB movie review dataset, provided by (Maas et. al.). We also have access to movie reviews from other sources through the Natural Language Toolkit, but benchmarks used in this paper are based on the IMDB dataset. This dataset contains 25,000 reviews for training and and an additional 25,000 for testing. Each set contains 12,500 “positive” reviews and 12,500 “negative” reviews. “Positive” reviews are those with ratings between 7 and 10 and “negative” reviews are those with ratings between 1 and 4. We ignore reviews with ratings of 5 and 6, because adding only very polar reviews to the dataset helps when training a discriminative model (Maas et. al.).

Textual data is very complex, so we first perform some data exploration to get a better sense of the difficulties that we will have to face when building our models. The first thing we notice is that many of the words only appear in a very small number of documents. Of the ~89,000 words present in the entire training corpus, approximately 44% of them only ever appear once, and 95% of them appear fewer than 100 times. Additionally, the 100 most common words make up a full 51% of the total corpus. This has some interesting implications when we convert to a bag-of-words representation (which we discuss in the next section). Namely, the data matrix 99.85% sparse, meaning that significant memory and computational savings can be obtained by using a sparse matrix representation.

Feature Selection

For many of our models, we use a common set of 89,000 bag-of-words features. Each of these features corresponds to a unique word found in the training corpus, and for a particular document, the value of each of feature is the number of times the corresponding word appeared in the document. This feature representation was introduced by (Joachims, 1996) and has since become a popular method for feature extraction in a number of natural language processing tasks.

To illustrate this feature representation, we give two highly polar example reviews as well as subsets of their feature vectors in in the bag-of-words representation.



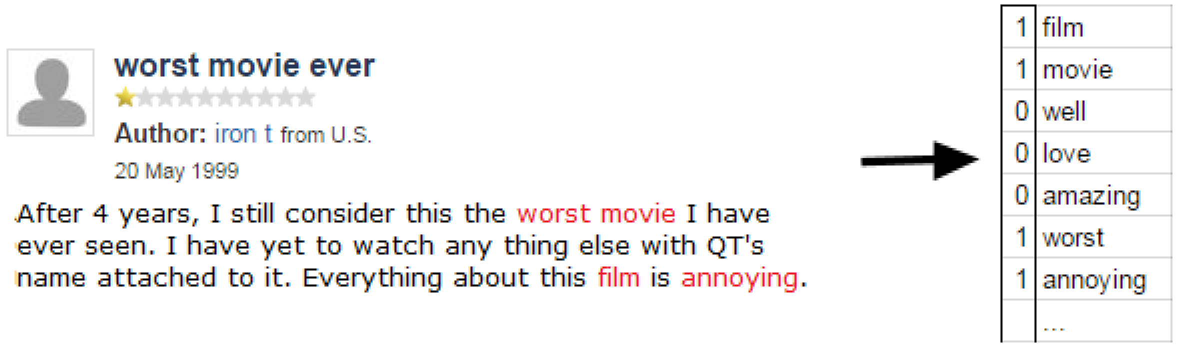
Love this movie!

Author: Ze Henrique Caldas from Brazil

27 January 2015

Well directed, well produced, love the photography, and the script is amazing. Quentin Tarantino is one of my preferred directors, and the Pulp Fiction was the first film from Tarantino that made me start looking different for this director's work.

1	film
0	movie
2	well
1	love
1	amazing
0	worst
0	annoying
	...



worst movie ever

Author: iron t from U.S.

20 May 1999

After 4 years, I still consider this the worst movie I have ever seen. I have yet to watch any thing else with QT's name attached to it. Everything about this film is annoying.

1	film
1	movie
0	well
0	love
0	amazing
1	worst
1	annoying
	...

Words like ‘love’ and ‘amazing’ have a strongly positive connotation, so we expect our sentiment analysis algorithms to assign some positive weight to this feature. Words like ‘worst’ and ‘annoying’, on the other hand, have a strong negative connotation, so we would expect our algorithms to assign their features a negative weight.

While this representation has seen great success in practice, it suffers from some severe limitations. This representation assumes that each word independently contributes to the sentiment of the review, and we can easily think of an example where this is not the case. A review which states “this movie is not good” would likely confuse the learner, because “good” will more often than not have a positive connotation, but this review clearly is negative. There is no way for this representation to capture the negation that occurs before “good”.

To combat this, we can introduce bi-gram features. These are similar to bag-of-words features, but instead they count the number of times a pair of words appear in a document. We would now have a feature for “not good” as well as “not” and “good”. While this enhances our ability to represent some complex relationships between words, it grows the number of features significantly. Some of our experiments will use these features.

Reducing the number of features to a representative subset is an important challenge. Each of our models approach this task in a slightly different way, giving diverse results.

Classification

We perform classification using two common models, Support Vector Machines and Naive Bayes.

L1-Regularized Support Vector Machines

A Support Vector Machine is a type of linear classifier, meaning that its prediction function is a thresholded linear combination of the input features. We can write this as a piecewise function of the dot product of the feature vector, x, and a weight vector, d.

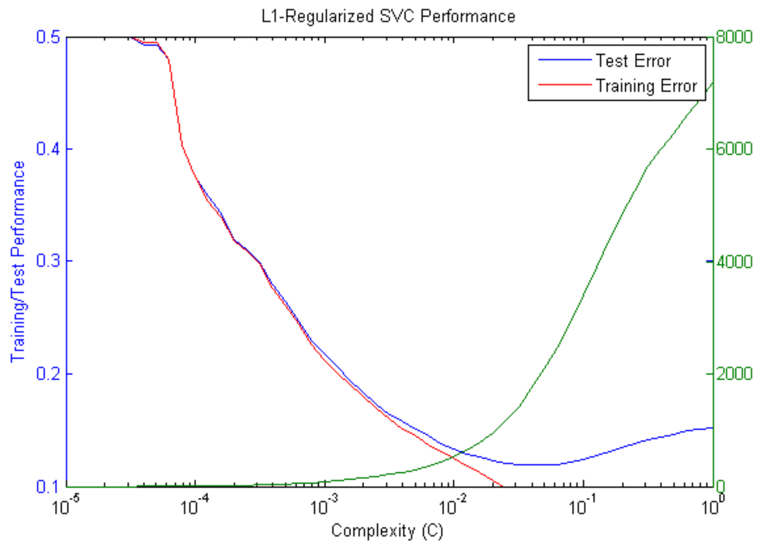
$$f(x;w) = \begin{cases} +1 & : w^T x \geq 0 \\ -1 & : w^T x < 0 \end{cases}$$

We classify a document as "positive" if a weighted sum of its bag-of-words features is greater than zero, and "negative" otherwise. The weights are chosen by minimizing the following objective:

The first term in the objective is the L1 regularization penalty, which penalizes the weight

$$w^* = argmin_w ||w||_1 + C \sum_i max(0, 1 - y_i w^T x_i)^2$$

vector if it grows too large. The second term penalizes the weights if they result in incorrect classifications. The constant term C balances these two penalties - a value close to zero will lead to a sparse solution and a value near infinity will result in perfect classification on the training set. We choose the complexity parameter C with cross-validation. We train models with increasingly large values of C, and choose the one that has the best test performance. As we raise C, we see that the error of the model on the training set gradually decreases, but eventually the error of the model on the test set starts to rise as our model starts to overfit the training set. The green line in the figure shows the number non-zero weights in the sparse solution.



Our optimal sparse solution, as validated by test performance, used only 1400 of the original 89,000 bag of words features, and achieved 88% test accuracy. Here, we give some of the most polar words learned by our model.

Positive Words	Weight	Negative Words	Weight	Neutral Words	Weight
excellent	.2880	waste	-.4599	line	< .0001

superb	.2277	worst	-.4380	dialogue	< .0001
perfect	.2221	awful	-.3615	actress	< .0001
amazing	.2103	poorly	-.3415	god	< .0001
wonderful	.2037	disappointment	-.2936	total	< .0001

It is important to note that we applied a few pre-processing techniques that helped achieve the best accuracy. First, each document's feature vector was scaled by the number of words that appeared in that document. This prevents reviews with incredibly high word counts from having a disproportionate effect on training. We also scale each feature by the number of times its corresponding word appeared in the corpus, which prevented common words like "the" from having too much of an effect on training. It is typical in the application of linear models to "zero-mean" each feature, but we found that this was inappropriate for this dataset because it eliminated all sparsity in the feature representation, leading to significantly increased computational cost.

Naive Bayes

Our initial classifier is a representation of the Naive Bayes model. This model takes the classic bag-of-words representation of a training corpus and assumes each feature contributes independently to the overall probability of a document being classified as positive or negative. By using this naive model we simplify our number of features as well as make the probabilistic calculations easier with a tradeoff that some useful sentiments can be misinterpreted or missed altogether.

We are focused on a problem of binary text classification, a document either fits into positive or negative classes. We use a Naive Bayes probabilistic equation to find the probability of a document belonging to one of these two classes:

$$c_{map} = argmax(P(c|d) = argmax(P(c)\Pi_{1 \leq i \leq n} P(x_i|c))$$

Where P(c|d) is the conditional probability of class c given document d, P(c) is the prior probability of c and $P(t_k |c)$ is the conditional probability of token k given class c. Since we use an equal number of negative and positive reviews the marginal class distribution P(c) is uniform, meaning it has no effect on the prediction. We must estimate the probability of each word of the document given a class(likelihood) multiplied by the probability of the particular class(prior). We calculate this probability for each of the two classes positive and negative.

We use the Multinomial Naive Bayes theorem which uses term frequency to classify documents. Term frequency is defined as the number of times a given term *t* appears in document *d*. We normalize term frequency by dividing the raw frequency by the document length.

$$normalized\ term\ frequency = \frac{tf(t, d)}{n_d}$$

tf(t, d): count of term *t* in document *d*
n_d: total number of terms in document d

The term frequencies are then used to compute the maximum-likelihood estimate(MLE) based on the training data to estimate class-conditional probabilities:

$$P(x_i|c_j) = \frac{\Sigma\, tf(x_i, d \,\varepsilon\, c_j) + \alpha}{\Sigma N_{d\varepsilon c_j} + \alpha * V}$$

This equations states that the probability of a word given a class is equal to the sum of raw term frequencies of x_i from all documents in training sample which belong to class c_j divided by the sum of all term frequencies in the training dataset for the same class c_i. Alpha is a parameter given a value of 1 for the case that a class and word never occur together in the training data. If this happens then the frequency based probability estimate would be zero, rendering the equation irrelevant. By adding in the Alpha no probability estimate will be exactly zero.

There are a few ways to improve a Naive Bayes classifier to try and capture more complex sentiments. We can introduce the concept of bi-grams, phrases that are two words long where the sentiment of the phrase is dependant on both words and is different than if you evaluated each word independently. Take someone saying “not good”. According to our model “good” would be evaluated as purely positive while if you introduce bi grams which look for a negation in front of a positive word “not good” would be recognized as negative.

You can also introduce the concept of Multiwords. This strategy utilizes Part of Speech(PoS) tagging to find sets of linguistically motivated terms. Word pairs such as Noun-Adj or Verb-Noun can have clearer sentiments when evaluated together(Garcia et. al)..

Classification Performance

Here we compare our two classification methods with the state-of-the-art results given by (Maas et. al.). We find that our relatively simple models are able to achieve nearly the same level of performance with very little computational cost.

	Random	Naive Bayes	Sparse SVM	Maas et. al.
Training Accuracy	50%	90%	95.0%	-
Testing Accuracy	50%	85.25%	88.1%	**88.89%**

Regression

In regression, we attempt to predict the rating of a review rather than just the polarity. Possibly the most common regression method is linear regression, but because our ratings are limited to the range [1, 10], we find that linear regression, which can predict any real value, leads to poor test performance. Instead, we perform logistic regression, which predicts a logistic function of the linear response, rescaled to be within the range (1, 10).

$$f(x; w) = 1 + \frac{9}{1 + e^{-w^T x}}$$

We then choose the weights to minimize the mean squared error.

$$w^* = argmin_w \frac{1}{N} \sum_i (y_i - f(x_i; w))^2$$

There is no closed-form solution for this optimization, but since the objective is convex, we can apply gradient descent to quickly converge on an optimal solution. We employ a commonly used variant of gradient descent called stochastic gradient descent, where, at each step, we compute the gradient using only a small number of random training examples. This works because the expectation of the stochastic gradient is equal to the full gradient, so over many iterations the solution will still converge.

Regression Performance

We compute the mean squared error (MSE) of our model on both our training and test sets for comparison. MSE is not easily interpretable on its own, so to measure the performance of our model we compare it against three different models. First, we predict "5" for all examples, which assumes no knowledge whatsoever of the training corpus. Second, we perform classification using the Sparse SVM and predict the mean rating within each class. Third, we perform simple linear regression and then clip the predictions that are outside of the [1, 10] range. We find that logistic regression performs the best out of the four models, achieving a MSE of 4.6362.

	5 Always	Linear Classification	Linear Regression	Logistic Regression
Training MSE	12.2442	4.6180	4.0203	3.1599
Testing MSE	12.4490	5.6903	5.0264	**4.6362**

Review Sampling

Probabilistic Context Free Grammar

The first probabilistic model we employ for sampling is probabilistic context-free grammar. Context-free grammars have been the subject of much research in natural language processing, because they can define a wide range of languages using a compact set of rules. We were interested in this as a probabilistic model for a few reasons. Primarily, unlike in graphical models, PCFGs provide an elegant framework for defining sentences of variable and potentially unbounded length. Secondly, CFGs and PCFGs model syntax structure, which we hoped would lead to grammatical sentences.

A PCFG consists of a weighted set of productions. Take the following simple example:

S → DP VP (1.0)	D → 'a' (.5)
DP → D NP (.75)	D → 'the' (.5)
DP → PNP (.25)	N → 'cat' (1.0)
NP → N (1.0)	PN → 'Joe' (1.0)
PNP → PN (1.0)	V → 'runs' (.75)

VP → V (1.0) V → 'sits' (.25)

The start token, S, represents an entire sentence. It has only one realization, a determiner phrase (DP) followed by a verb phrase (DP), which occurs with probability 1. A determiner phrase, however, can be realized as either a determiner (D) and a noun phrase (NP), or as proper noun phrase (PNP). One example sentence we might generate with the model is (S (DP (D the) (NP (N cat))) (VP (V sits))), "the cat sits".

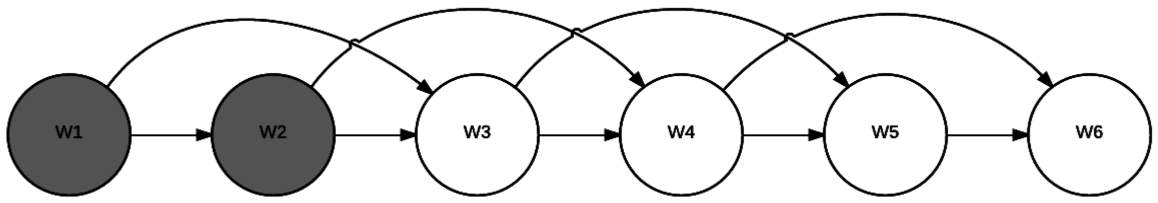
Using the Penn Treebank dataset as a training set, we can train a PCFG to sample random sentences. The maximum likelihood estimates of the weights for each possible production is simply the empirical frequency of its occurrence in the dataset. Unfortunately, inconsistencies in this dataset make training incredibly difficult. We were unable to produce a model that generated reasonable sentences from this dataset, so we primarily used other approaches.

Markov Chain

For the random movie review generation we are creating a Markov chain of bigrams and the probabilities of possible following words. To do this we are processing the movie reviews and storing the each pair of consecutive words as a bigram and then finding each occurrence of that bigram and recording the possible words that follow it to create the probability. Thus, the probability to go from bigram (w1, w2) to w3 is the the number of occurrences of w3 after (w1, w2) divided by the number of occurrences of (w1, w2).

$$P(\mathbf{w}) = P(w_1)P(w_2|w_1)\prod_{i=3}^{n} P(w_i|w_{i-1}, w_{i-2})$$

Once the model is built, all that is needed is to supply a seed bigram (chosen randomly or user supplied) and then the program can iterate through the chain, with each new word supplying the second half of the next bigram (e.g. (w1, w2) to w3 becomes (w2, w3) to choose the next word).



$$P(w_{3:n}|w_1, w_2) = \prod_{i=3}^{n} P(w_i|w_{i-1}, w_{i-2})$$

During the quarter we made multiple attempts at integrating part of speech into this same type of markov chain and other related forms. These attempts, however, remained fruitless. What happened for most of the models we managed to create was that they nearly word for word mimicked long strings of words from individual movie reviews. This was due to there being such a large number of pairs of parts of speech that there was often only a couple of possible sequences of subsequent words given any pair of parts of speech and pair of words (i.e. the set of words possible given previous word 1 & 2 and previous parts of speech 1 & 2 was very small.). Because of this we ended up getting rid of the parts of speech implementation and focusing instead on making the simpler markov model more feature rich.

One improvement on our model was the inclusion of select punctuation. We are using the NLTK word tokenizer to create individual tokens as opposed to our own tokenizer implementation used previously. The NLTK tokenizer includes tokenized punctuation such as end of sentence marks, commas, apostrophes, etc. We filter out some of these marks to simplify the model to more basic sentences (i.e. removing any HTML markup, colons/semicolons, quotation marks).

One example of this was adding the ability for the user to supply the seed words. Using this feature we can make a "review" of a particular movie by supplying appropriate seeds (e.g. ("The", "Godfather"), ("Dark", "Knight")).

Another feature we implemented was finding the most likely sentence based on a given seed. This corresponds to solving the following MAP inference problem:

$$w_{MAP} = argmax_w \ P(w_1)P(w_2|w_1)\prod_{i=3}^{n} P(w_i|w_{i-1}, w_{i-2})$$

Our first approach uses a greedy algorithm that chooses the next most likely word in the chain as opposed to finding the complete sentence with the highest probability as a whole. At the time of writing this report, we are working on implementing a more complete algorithm to find the true MAP assignment. We also hope to perform MAP with evidence, such as a small number of seed words at the beginning of the sentence. The two approaches we are looking into for this are the Viterbi algorithm of finding the most likely sequence of a Markov chain and variable elimination.

Sampling Results

The resulting text of the markov generator are mixed. At times they can seem somewhat coherent if rambly, and others are complete gibberish. Following are a few example sentences from the generator:

```
"i know how this young pilot who had shut
himself off also he gives the comedy of stan
the man with ."

"two other films such as ca you made us wonder
why chinese actors in it that beautiful smile ,
at best ."
```

Results with a user supplied seed (first two words) seem slightly better but not much:

```
“dark knight is left hanging all over the head
vampire , in fact everybody was about 15
minutes without a word about .”
```

Conclusions and Future Directions

Our results show that simple models using a straightforward bag-of-words feature representation are incredibly effective at identifying sentiment in movie reviews. From this we can conclude that most information about sentiment is stored at the level of individual words, rather than within higher-level structure in the text. We gave some examples where our models fail, but overall our models are quite successful, especially considering their low computational costs. Our models match the classification performance of state-of-the-art probabilistic models. We also create, to the best of our knowledge, the first baseline results for regression on this dataset.

We have also shown some surprisingly impressive results in sentence sampling. The failure of probabilistic context-free grammars for sampling reinforces our conclusion that sentiment is stored at the word level. PCFGs allow us to model sentence structure, but not semantics. Unsurprisingly, very little information about sentiment is stored within the syntax of a sentence. Our Markov chain models, however, model the relationships between adjacent words directly, rather than through their grammatical function in the sentence, which turns out to be a much more effective strategy in this domain.

In future work, it would be interesting to further study the trade-off between computational cost and model performance. It may be possible with more complex models to achieve higher levels of accuracy, but at potentially much higher computational cost. The limitations of our own hardware used for this project prohibited the use of any deep neural network approaches, but their recent successes on many different types of machine learning problems (Krizhevsky, Sutskever, and Hinton, 2012) suggest they may be effective in this domain well.

Not discussed in this work is the impact and applications of sentiment analysis. In future work, it would be interesting to see the effects of algorithms like the ones we introduce here in practice. Some potentially impactful research questions arise from this idea: Could we identify signs of mental health issues from text samples? Would it be possible to detect criminal activity from email communication while still maintaining user privacy? The potential impact of sentiment analysis is immense, and to our knowledge, has not been studied in any depth.

References

Blei, David M., Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation." *the Journal of machine Learning research* 3 (2003): 993-1022.

Bollen, Johan, Huina Mao, and Xiaojun Zeng. "Twitter mood predicts the stock market." *Journal of Computational Science* 2.1 (2011): 1-8.

Brants, Thorsten. "TnT: a statistical part-of-speech tagger." *Proceedings of the sixth conference on Applied natural language processing*. Association for Computational Linguistics, 2000.

Fan, Rong-En, et al. "LIBLINEAR: A library for large linear classification." *The Journal of Machine Learning Research* 9 (2008): 1871-1874.

Garcia, Pablo Gamallo ; Marcos. "Citius: A Naive-Bayes Strategy for Sentiment Analysis on English Tweets." (n.d.): n. pag. *Http://alt.qcri.org*. Web.

Ghiassi, M., J. Skinner, and D. Zimbra. "Twitter brand sentiment analysis: A hybrid system using n-gram analysis and dynamic artificial neural network."*Expert Systems with applications* 40.16 (2013): 6266-6282.

Joachims, Thorsten. "A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization." (1996).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.

Maas, Andrew L., Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. *The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011).*

Mullen, Tony, and Nigel Collier. "Sentiment Analysis using Support Vector Machines with Diverse Information Sources." *EMNLP*. Vol. 4. 2004.

Pang, Bo, and Lillian Lee. "Opinion mining and sentiment analysis."*Foundations and trends in information retrieval* 2.1-2 (2008): 1-135.

Appendix

Instructions

The majority of our code requires a working installation of Python with the Natural Language Toolkit (NLTK) and its dependencies. Installation instructions for NLTK can be found here: http://www.nltk.org/install.html. In order to train models directly through the interface, you will need to download the ACLIMDB dataset and place its contents in the source directory. The dataset can be found here: http://ai.stanford.edu/~amaas/data/sentiment/ (80+ MB). In order to run our demo code, run "inteface.py" with Python after installing the necessary dependencies.

The included MATLAB code requires LIBLINEAR 1.96, which can be downloaded here: http://www.csie.ntu.edu.tw/~cjlin/liblinear/. Extract the downloaded folder into the source directory. The MATLAB code also requires the ACLIMDB database.

To use the text generator, you simply need to press the "Generate" button in the center column. It can take a moment to load and sometimes can cause the UI to freeze for a moment while it generates the sentences, but it will load shortly. Alternatively you can enter your own two word seed in the bottom center text box (it must be only 2 words, no punctuation of any kind, case insensitive) and then press the "Gen w/ Seed" button to generate sentences based on your given seed.

Software Used

Several of our core algorithms, including feature preprocessing, Naive Bayes, and the Markov Chain model, are implemented in Python with the help of the Natural Language Toolkit (NLTK). This toolkit has built-in utilities for part-of-speech tagging, building context-free grammars, and accessing popular text datasets. NLTK also has implementations of many of the models we use in this work, which we can use for comparison. These implementations include functions for visualizing aspects of our model, such as giving the likelihood ratios of the most informative features used in Naive Bayes. We also use NLTK's word tokenizer for building the list of individual features.

The sparse Support Vector Machine implementation comes from LIBLINEAR (Fan, Rong-En, et al.) which interfaces with MATLAB. We use MATLAB to run experiments and generate figures. The logistic regression model is also implemented in MATLAB, using a hand-written implementation of stochastic gradient descent. Once these models are trained, they are saved in files accessible by Python.

We use Tkinter to build the graphical interface from which we can train models, adjust their parameters, and display their results.

Division of Work

Jonathan ran experiments to determine optimal settings for the Sparse SVM and feature preprocessing steps. He also worked on data exploration and unsupervised learning. He contributed the working implementation of stochastic gradient descent that was used to train the Logistic Regression model. He also trained a probabilistic context-free grammar for sentence sampling and produced the examples given in this paper.

Taylor implemented the random review generation. She worked on creating the markov models from the dataset as well as the subsequent generation features such as user selected seed, and most likely sentence. She also contributed to the attempt to getting the PCFG working for review generation.

Jacobus was responsible for the interface and the comparisons of classifier models and the implementation of the python code in general. He worked on implementing the Naive Bayes classifier.

The group as a whole met frequently outside of class to collaborate and share results. We communicated through email and used a Git repository to share code. Presentations and reports were written collaboratively using Google Docs.