AH-SAR: Ad Hoc Sentiment Analysis on Ratemyprofessor
An easy-to-build and fast-to-train system with high accuracy

Qian Chen, Tony Li, Xinyu Xie
New York University, qc543@nyu.edu, yl4649@nyu.edu, xx708@nyu.edu

ABSTRACT

In this paper, we present multiple solutions of sentiment analysis using different classification models on the students' comments of college professors from the website RateMyProfessors.com (RMP). Our aim is to investigate the relationship between the numerical scores students rate on professors and the underlying sentiment behind the textual commentary students write, create an easy-to-implement, fast-to-train solution for sentiment analysis, and provide a new dimension for evaluating professors from an NLP standpoint that incorporates qualitative analysis with quantitative analysis.

Keywords: Sentiment Analysis, RateMyProfessors, Natural Language Processing, TF-IDF, N-gram, Vector Similarity, Machine Learning, Neural Network, Linear Regression, Stochastic Gradient Descent, Multinomial Naive Bayes

INTRODUCTION

As professors play a crucial part in college students' academic experience, online platforms of rating and reviewing professors have become very popular in an era of shared information. As the biggest among these platforms, RateMyProfessors.com(RMP) has been the go-to option and decisive factor that influences college students' decision making before selecting a class. From sharing basic class information such as attendance policy and textbook requirements, to rating the quality and difficulty of a class and giving personal comments on teaching style and workload, students share their experience and influence the decisions of others, while also having their own decisions influenced. Given the large amount of data that covers the opinions of college students from all over the country, it is interesting to delve into the data and dig deeper into the sentiments behind students' comments.

A review on RateMyProfessors.com consists of several quantitative rating scores (quality, level of difficulty, etc), and also the student's subjective comment in natural language. Such a design in nature facilitates the development and evaluation of a sentiment analysis system, as it makes it possible to have the system make its judgement on the sentiment of a review based on the natural language comment, and evaluate the result with regard to the quantitative scores already available on the website. Although our sentiment analysis system is built targeting analyzing student reviews for college faculty, it could also, by adding corpus from other domains (called domain adaptation), be generalized to analyze a wider range of online verbal comments, and help the entities being reviewed identify their potential strengths and weaknesses.

For our sentiment analysis system specifically designed for RMP, our aim is to explore and investigate the relationship between the numerical scores students rate on professors and the underlying sentiment behind the textual commentary students write along with the numerical score, and create an easy-to-implement, fast-to-train solution for sentiment analysis with high accuracy and other evaluation metrics. Once the system is done, we wrap it into an easy-to-use python script (which we call it our *application*) so that users can use our system directly without looking too deep into the actual code implementation, while the structure integrity of our system will be kept untouched from the users. By doing so, we provide the users a new dimension for evaluating their professors from an NLP standpoint, which could be helpful in their own decision making process on choosing professors.

WORKFLOW

We here introduce the four-stages workflow first to create a preliminary demonstration of our system first, so we can explain with the algorithmics better when we dive into details afterwards. Notice that although in the actual development process, some stages were overlapping, embedded and nested, or reordered, the conceptual and logical workflow of our system is best described as shown in the flowchart below.

Pre-Process stage

We retrieve *Raw Data* from the Internet and categorize them as either *Lexicon* or *Corpus*.

Then we performed *Data Refinement* on the raw data, which includes removing duplicates and null values, normalizing texts into lower case without special symbols like mentioning or hashtag, and stemming each token of the text using NLTK PorterStemmer. Note that a commonly used approach - removing stop words (i.e. words that do not have significant meaning in a sentence, like "he",

"she", "there", "that", etc.) - as well as using NLTK lemmatizer are not adapted in our own pre-process stage, because we actually tested our system on various dataset with different level of pre-processing, and we found that removal of stop words and usage of lemmatizer will lower the accuracy and other evaluation metrics.

For some unlabeled data, we also labeled them by ourselves, which we call it *Data Labeling* in the flowchart.

At last, as a conventional approach in Machine Learning, we shuffle the data with Pandas Dataframe functionality and divide them into development set, training set, and testing set, which is called *Data Resampling*.

Model Generation stage

Now that we have a refined and labeled dataset, we design models and methods to solve our problem in the *Model Design* process. The solutions we will talk about in this paper include two types of classification systems implemented on our own and some online-available systems implemented by predecessors.

Next is *Model Training*, where we train our model using the dataset we have.

At last, we perform a *Model Exporting* on the trained models, so that we can speed up by tens in the future when we re-run the program, as long as the model design and dataset are not changed.

Model Evaluation stage

Now that we have some running models, we perform *Model Testing* on not only the metrics of Precision, Recall, and F-measure, but the executing speed as well.

Next is *Models Comparison*, where we compare the metrics of different models and make optimization decisions based on the performance: e.g. which models to optimize further, or which models to integrate.

Model Generation and **Model Evaluation** collectively form a cycle. We repeat the cycle for better performance until we launch the application.

Application Launch stage

Now that we have an optimized model, we wrap it with certain user interfaces as a basic and rudimentary *Working Application* and launch on GitHub. By doing so, we can allow others to utilize our system without spending extra time on looking at the detailed code, and therefore maintain the system integrity by preventing users without NLP knowledge from accidentally changing the system file and crashing the system.

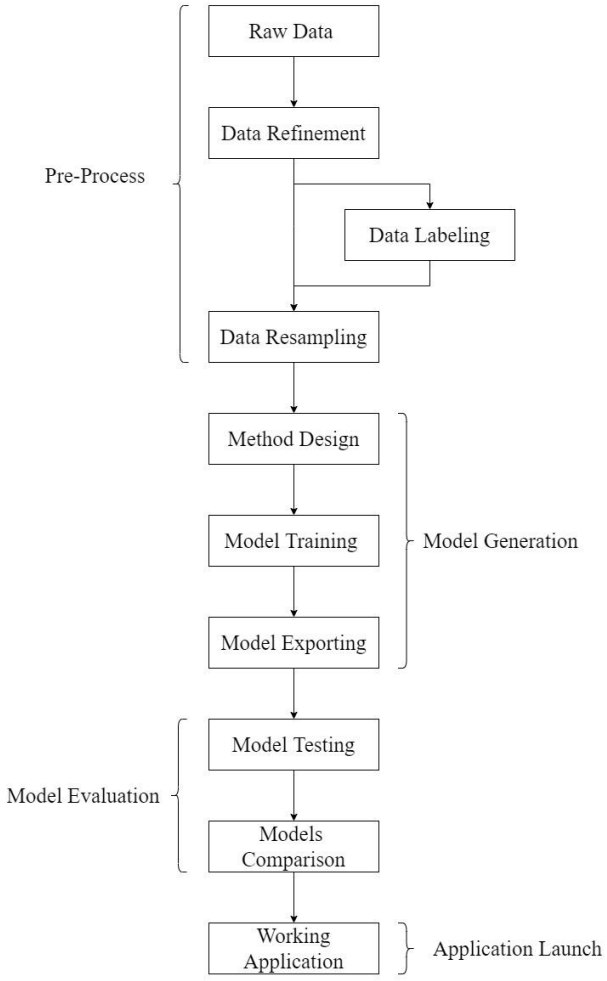

Figure 1: The entire workflow of our project. Notice that the four stages of the workflow are presented in a logical and loosely-chronological order.

DATASET

We have two types of datasets: a dictionary mapping individual words to related values, in this case, the sentiment score, which we call *Lexicon*; a collection of texts and related sentiment score, which we call *Corpus*. We perform our tasks based on these datasets, but with our own improvement or modification.

I. Lexicon

HTL/EMNLP

A lexicon consists of 8222 entries with duplicates for words with multiple Part of Speech tags (Theresa Wilson, Janyce Wiebe, and Paul Hoffmann, 2005). The sentiment value for each entry is categorical: positive, neutral, or negative. For the purpose of our own system, we only

extract the positive and negative entries, and drop all duplicates, for simplicity.

Vader

A lexicon consists of 7520 entries with special punctuational expression, i.e. :-) or ('}{') , which are generally considered illegal syntax in formal English language (Hutto, C.J. & Gilbert, E.E., 2014). The sentiment value for each entry is numerical in scale of [-4, 4] and is derived from the ratings given by 10 human judges, which is also included in the lexicon as an array. For the purpose of our own system, we only extract the formal English words and related numerical values, and we categoricalize the values into positive and negative, for simplicity.

SentiWords

A lexicon consists of 155,287 entries with numerical values in scale of [-1, 1] (Gatti, Lorenzo, Marco Guerini, and Marco Turchi, 2016). In this lexicon, sequences of words are connected and delimited by underscore sign (e.g. "willard_van_orman_quine"), and the majority of such sequences of words merely has its meaning in our Sentiment Analysis system since they are just word salad. Some obvious data inconsistency is found in this lexicon as well, e.g. commonly used dirty word "fuck_all" has a negative sentiment value of *-0.07997* and yet "fuck_off" has a positive value of *0.07118*; words "willful" and "willfully" have positive values of *0.17800* and *0.15793* respectively, but their noun morphology "willfulness" has negative value of *-0.34088*. After attempts of incorporating this lexicon into our system, we decide to actually not use it, as it lowers the performance of our system in terms of both speed and accuracy.

II. Corpus

IMDB Film Reviews

A corpus consists of film reviews from IMDB with 25,000 positive and 25,000 negative. Corpus are pre-processed and divided into valid, train, and test subsets by Kaggle user Ziqi Yuan; the original source is released by Stanford (Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts, 2011). Our system was at first developed on this corpus, but we transitioned into using the RMP corpus that we labeled by ourselves by the end.

RMP

A corpus consists of textual commentary on college professors on RMP and related sentiment scores in binary form (-1, or 1) labeled by ourselves. We wrote our specialized scraper to retrieve the commentary data from RMP website, and then label them semi-manually: we manually label the comments with quality score between 2.5 and 3.5, while the comments over 4.0 are automatically labeled as positive and comments under 2.0 as negative. We ended up using a self-labeled RMP corpus of 80,000 comments, of which 40,000 are positive and 40,000 are negative. 80% of the RMP corpus was used as the training set, while the other 20% was used as the testing set.

Twitter Sentiment Classification

A corpus consists of 800,000 positive and 800,000 negative tweets retrieved from Twitter and labeled using distant supervision (Go A, Bhayani R, Huang L, 2009). This corpus looks very large, comprehensive, and thus should be valuable to our system; but in fact, it contains an "unusual amount of slang and non-normal spelling", as the authors noted in their paper, and it also contains illegal grammar, word salad, and abusive languages that generally do not appear in commentary text. After attempts of incorporating this lexicon into our system, we decide to actually not use it, as it lowers the performance of our system in terms of both speed and accuracy significantly.

MODEL

The computer does not understand the definition of the words directly, but instead it could grasp a sense from the surrounding context of the word. We construct three types of models based on the lexicon and corpus we have, and the later two models accommodate this special need for the computer to understand words and make judgement and prediction on its own.

Mapping between Words and Sentiment

Since we develop in a Python environment, this is just a Python Dictionary of word and sentiment pairs. We build the dictionary based on the two lexicons: HLT/EMNLP and Vader.

Sample key-value pairs in the Python Dictionary would be: {'amazing': 1, 'abandon': 0, ...}

Mapping between Token and Occurrences

A Python Dictionary of pairs of tokens and tokens' occurrence counts in positive and negative sentences. We build the dictionary based on the corpus, and the tokens mentioned above can be extended to sequences of tokens, i.e. N-grams. In our system, we have unigram, bigram, and trigram models. Quad-gram is once on the schedule, but we consider it not worthy to run by the end, as it will most likely not improve the accuracy by the idea of diminishing marginal return. A sample key-value pair would be: {'very disorganized': (5, 20)}, where the 5 means 5 occurrences in positive sentences, and the 20 means 20 occurrences in negative sentences. For the purpose of our system, we further normalize the occurrences into weights, so (5, 20) is now (0.20, 0.80), meaning that statistically this bigram sequence has a 20% chance to be positive, and 80% chance to be negative, based on the corpus we have.

Sample key-value pairs in the Python Dictionary would be: {'good professor: (0.9, 0.1), 'monotone voice': (0.2, 0.8), ...}

Vector Representation of Sentences

Vector representation of sentences is a vectorized abstraction of the sentences in textual form. In particular, we have Count Vectorization which considers only the count of each token in a sentence, and Terms Frequency - Inverse Document Frequency (TF-IDF) Vectorization, which considers not only the count of each token within a sentence, but the count of each token in the entire corpus as well. The higher the TF-IDF score of a token is in a sentence, the more specific and important this token is to this sentence.

METHOD

For the three types of models mentioned above, we implement five methods progressively on our own, and end up with incorporating some of them. We also adapt and utilize four methods available online implemented by predecessors in the field of NLP, which will be discussed after the self-implemented methods.

I. Self-implemented Methods

Lexicon-based Unigram Lookup

For each token in the sentence, we look up its related sentiment value stored in the lexicon-based unigram model, i.e. the mapping between words and sentiment. Each token, if existing in the model, has a binary value of either -1 (negative) or 1 (positive). The binary values of each token, collectively, average into a total sentiment weight, and we use this total weight as the final decision of the sentiment of the sentence. The averaging process of a sentence *s* contains *n* tokens can be described as such:

$$Sentiment(s) = \frac{1}{n}\sum_{i}^{n} unigramModel(s[i])$$

The averaged sentiment score is a single value. A sentence is positive if its sentiment score is above 0, and negative if below.

Corpus-based Unigram Lookup

Similarly, for each token in the sentence, we look up its related sentiment weights stored in the corpus-based unigram model, i.e. the mapping between token and occurrences. and if the token does not exist in the model, we treat it as (0.5, 0.5), inferring that this token is neutral. The sentiment weights of each token, collectively, average into a total sentiment weight and we use this total weight as the final decision of the sentiment of the sentence.

$$positiveSentiment(s) = \frac{1}{n}\sum_{i}^{n} unigramModel(s[i][0])$$

$$negativeSentiment(s) = \frac{1}{n}\sum_{i}^{n} unigramModel(s[i][1])$$

The averaged sentiment weight is a value pair in the form of (*positive weight*, *negative weight*) in the end of computation. A sentence of sentiment weight = (0.6, 0.4) suggests that this sentence should be positive overall.

Corpus-based Bigram Lookup

Similarly, for each bigram sequence in the sentence, we look up its related sentiment weights in the bigram model. If a bigram sequence does not exist in the model, we divide it into two unigram and back-off to the unigram model. Notice that we have two different unigram models above, and we choose the lexicon-based one to back-off with, because it provides a much better accuracy and other evaluation metrics. The sentiment weights of each bigram sequence in a sentence, collectively, average into a total sentiment weight and we use this total weight as the final decision of the sentiment of the sentence.

Corpus-based Trigram Lookup

Similarly, for each trigram sequence in the sentence, we look up its related sentiments weights in the trigram model. If a trigram sequence does not exist in the model, we divide it into two bigrams and back-off to the bigram model; if any of the two bigrams does not exist in the bigram model, we further back-off to the lexicon-based unigram model.

TF-IDF Vector Similarity

We compute the cosine similarity between each TF-IDF vectorized sentence from the testing corpus and each TF-IDF vectorized sentence in the training corpus. Cosine similarity between two vectors *x* and *y* of *n* dimensions is defined as such:

$$Similarity(x, y) = \frac{\sum_i^n x_i \times y_i}{\sqrt{\sum_i^n x_i^2} \times \sqrt{\sum_i^n y_i^2}}$$

Then, for each instance from the testing corpus, we get the top N instances from the training corpus with highest cosine similarity score (meaning they are most-related), and compute their average sentiment score as the final decision of the sentiment of the test sentence. Notice that in our system, N is decided to be 20.

II. Adaption of Methods by Predecessors

Logistic Regression (LR)

The underlying Logistic Regression is based on linear regression. Through the input sample data, the linear prediction equation is obtained based on the multiple linear regression model:

$$y = w_0 + w_1 x_1 + w_2 x_2 + ...$$

The model returns continuous values, which cannot be directly used. Therefore, there's a way to convert continuous predicted values into discrete predicted values, which is using the Sigmoid function:

$$y = \frac{1}{1 + e^{-x}}$$

In our attempt to perform LR on both count vectors and TF-IDF vectors, the *predict()* function returns the LR predicted tag value after training.

Stochastic Gradient Descent (SGD)

SGD is a simple and efficient optimization technique in sklearn. The mathematical procedure is taking a set of training samples (x1, y1), …, (xn, yn) and learning a linear scoring function:

$$f(x) = w^T x + b$$

The training error is minimized by the following equation:

$$E(w, b) = \frac{1}{n} \sum_{i}^{n} L(y_i, f(x_i)) + \alpha R(w)$$

Multinomial Naive Bayes (MNB)

Assume there's a dataset D = d1, d2, d3, … dn, for any di, there exists a feature set Xi. The size of di is m, which means it has m features (the feature could be a word, a N-gram, etc. Usually before using MNB, we would obtain P(C), P(X), and P(X|C) from the training data. After that, we can obtain:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

This is the value that we're trying to obtain with MNB, which is given a piece of text that is characterized by X, the probability that the text belongs to category C. We use mnb.predict() to make such predictions on TF-IDF after constructing a model with MNB.

Word2Vector with Keras (W2V)

We implemented keras.metrics to calculate several important metrics including precision, recall, true positives, true negatives, false positives, and false negatives of the sequential model, which is a stack of layers where each layer features exactly one input and one output tensor. We used sequential() to initialize a class with a fully-connected layer (Dense) and 32 neurons. At the same time, relu is connected and the input is 500 dimension. Then, the model is compiled and used to fit our W2V test set.

RESULTS

We test each of our systems with a random sample of 20% of the RMP corpus, and record the Precisions, Recalls, and F-measures on positive comments alone, negative comments alone, and overall. The results indicate that in our setting, N-gram systems are overall more favorable than vector-based systems. The same holds even for vector-based systems with the boost of machine learning (classification models) or neural network (Tensorflow.Keras). The best performance comes from our self-implemented bigram system, which achieved an average F-measure of 89.3%. Detailed test results and interpretations are as follows.

I. Self-implemented Models

		1	2	3	4	5
Time Cost (sec)	Generating Model	0.01	24.97	27.63	30.35	1256.50
	System testing	6.72	6.31	6.80	7.52	13.63
Precision (%)	On positive	65.8	94.5	91.7	90.9	85.9
	On negative	87.3	74.1	85.8	84.1	81.5
	Overall	71.2	81.3	88.5	87.2	83.5
Recall (%)	On positive	87.0	66.6	84.8	82.8	73.5
	On negative	39.0	96.1	92.2	91.7	81.7
	Overall	62.9	81.3	88.5	87.2	77.6
F-measure (%)	On positive	74.9	78.2	88.1	86.7	79.3
	On negative	53.9	83.7	88.9	87.7	81.6
	Overall	71.2	81.3	88.5	87.2	83.5

Table 1: statistics of performance of self-implemented models in a typical test.
Numbers and corresponding models:

1 Lexicon-based Unigram
2 Corpus-based Unigram
3 Corpus-based Bigram
4 Corpus-based Trigram
5 TF-IDF Vector Similarity

Note that for all corpus-based models, precision on negative is notably lower than precision on positive. A reason we found for this result is that we didn't handle negation (not, never, however, etc) in our models, so the detection on negative is not accurate enough.

Lexicon-based Unigram / Corpus-based Unigram

These unigram models both have an average F-measure in the mid-60 percent, but the variance of F-measures in different rounds of tests is very large. Besides, these models have poor performance on detecting negative sentiments.

Corpus-based Bigram

This model has the best performance so far, with an average F-measure of almost 90 percent and very small variance. The model also has high performance in terms of speed.

Corpus-based Trigram

This model has the second-best performance so far, with an average F-measure in the high-80 percent, but about 1% lower than the bigram model. With the bigram model embedded in this model, its performance in terms of speed is also high.

TF-IDF Vector Similarity

Our IF-IDF model with cosine similarity achieved an average F-measure around 80%, but its performance in terms of speed is very poor.

II. Adaptation of Models by Predecessors

Count Vector (CV) and TF-IDF Vector (TV)

As stated before, we integrated CV and TV models with 3 classification models respectively, including Logistic Regression (LR), Stochastic Gradient Descent (SGD), and Multinomial Naive Bayes (MNB), resulting in a total of 6 adapted models. F-measures of these models turned out to be astonishingly low, mostly ranging from 50% to 70%. We consider these results to be fair, as the original authors that implemented these models also got similar results on their corpus.

		LR		SGD		MNB	
		CV	TV	CV	TV	CV	TV
Precision (%)	On positive	65.0	65.0	50.0	50.0	66.0	66.0
	On negative	70.0	70.0	100.0	0.0	69.0	69.0
	Overall	67.5	67.5	75.0	25.0	67.5	67.5
Recall (%)	On positive	74.0	74.0	100.0	100.0	72.0	72.0
	On negative	61.0	61.0	0.0	0.0	63.0	63.0
	Overall	67.5	67.5	50	50	67.5	67.5
F-measure (%)	On positive	69.0	69.0	67.0	67.0	69.0	69.0
	On negative	65.0	65.0	0.0	0.0	66.0	66.0
	Overall	67.0	67.0	33.5	33.5	67.5	67.5

Table 2: statistics of performance of adapted models in a typical test.

Word2Vector with Keras (W2V)

This model achieved an average F-measure in the mid-80 percent, which is about 5% lower than our N-gram model.

AUC (%)	93.78
Precision (%)	86.76
Recall (%)	85.40
F-measure (%)	86.07
True positives	6767
True negatives	7043
False positives	1033
False negatives	1157

Table 3: statistics of W2V model performance in a typical test. “AUC” is the Area Under the ROC curve, which is a measure of how well a model can distinguish between two classification groups (in our case, positive/negative). It can also be regarded as a loose measurement of accuracy.

Conclusion and Further Work

Performance Analysis and Reflection

So far, our self-implemented N-gram model, with a higher F-measure and lower time cost, has proved its performance against the mainstream models by our predecessors in similar application settings. After implementing/adapting, optimizing, and testing over 10 different models, we have also observed that the approach of sentence vectorization may not be as powerful as N-gram language models in the sentiment analysis of informal, non-academic texts.

As we developed our N-gram models, we achieved huge improvement when upgrading from unigram to bigram, but much less, and even none, improvement when upgrading from bigram to trigram. At first, we attributed this abnormal phenomenon to our corpus not being large

enough; but we still got similar results after we doubled the size of our training corpus. This observation implies that a trigram may not necessarily carry more information than a bigram in the context of informal, non-academic texts.

Application Launch

We wrap our model with certain user interfaces as a basic and rudimentary *Working Application* and launch it on GitHub. Our application asks the user to enter either a professor name or the url of a RMP rating page, and if there are multiple professors with the same name, prompts the user to select the professor he wants based on academic institution and department. The application outputs the quantitative rating scores available on the corresponding webpage, together with a series of sentiment scores generated by our sentiment analysis system based on all the textual comments for this professor. The results will give the user a more comprehensive picture of what people think about this professor, and facilitate the user's decision making process. The codebase of our system, along with lexicons and corpus, resides in the GitHub repository: https://github.com/Xinyu-bot/NLP_SentimentAnalysis_RMP.

Further Work

Our sentiment analysis model can be further improved from several aspects, if time and resources allow.

First, a larger and more accurately labeled corpus of online commentaries would potentially raise the accuracy of our models. Although we have filtered out some obviously problematic datasets obtained from the Internet (such as SentiWords mentioned before) in the early phase of this project, the quality of some datasets we eventually used is still questionable. Besides, the approach of binary classification, i.e. classifying texts as either positive or negative, can also be improved by adding more categorical values (neutral, mildly positive/negative, etc), or turning categories into continuous quantitative scores.

Secondly, our models haven't accounted for *negation* or *hyperbaton* such as not, never, but, however, etc. As mentioned before, the lack of special handling for negation has given rise to a relatively low performance on negative detection. The next step would be to detect negation in the texts and to determine the scope of each negation, i.e. the sequence of words that are affected by a negation term.

Furthermore, we have read papers on *detection of sarcasm*, which is one of the frontier research fields in sentiment analysis. Most sarcasm detection systems now focus on social media posts. Based on our examination of some typical social media datasets in this project, the use of language in those datasets can be quite different from the use of language in the context of evaluating professors. So some adaptation and localization would be required if we want to extend existing sarcasm detection systems to our context.

The integration of more corpus, specially handling on negation, and sarcasm detection gives us the prospects of a more comprehensive and accurate sentiment analysis system. Hopefully, the development of such a system will contribute to our understanding of sentiments behind natural language texts, and offer us a deeper insight into how machines can be efficiently trained to interpret natural language.

Acknowledgment

We would like to thank professor Adam Meyers and our mentor Kawshik Kannan for providing us guidance and feedback throughout the project. We would also like to thank the providers of our corpus and lexicon for giving us a good headstart in designing our models, and the original authors of our adapted models.

References

Theresa Wilson, Janyce Wiebe and Paul Hoffmann (2005). *Recognizing Contextual Polarity in Phrase-Level Sentiment Analysis*. Proceedings of HLT/EMNLP 2005, Vancouver, Canada.

Hutto, C.J. & Gilbert, E.E. (2014). *VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text*. Eighth International Conference on Weblogs and Social Media (ICWSM-14). Ann Arbor, MI, June 2014.

Gatti, Lorenzo, Marco Guerini, and Marco Turchi. *SentiWords: Deriving a high precision and high coverage lexicon for sentiment analysis*. IEEE Transactions on Affective Computing 7.4 (2016): 409-421.

Yuan, Ziqi. *IMDB Dataset*. (2020). https://www.kaggle.com/columbine/imdb-dataset-sentiment-analysis-in-csv-format.

Besbes, Ahmed. *Sentiment Analysis on Twitter using Word2vec and Keras*. (2017). https://www.ahmedbesbes.com/blog/sentiment-analysis-with-keras-and-word-2-vec

Sentiment Analysis of IMDB Movie Reviews. (2020). https://www.kaggle.com/lakshmi25npathi/sentiment-analysis-of-imdb-movie-reviews/notebook