Comparing SVM and Naïve Bayes for Sentiment Analysis of Stack Overflow Comments

by Tyler Sattler

Abstract: Stack Overflow is a platform for programmers to post and answer questions regarding their code issues. Using the dataset provided by Calefato [1], we analyzed the polarity of each comment based on a three-class metric: Negative, neutral, and positive response. The dataset provided 4424 labeled comments and was split to 70% train and 30% test. Using the nltk tool box[2], each comment , referred to as a document, was pre-processed by tokenizing and lemmatizing the text to be in a more general form without stop words. The text was then vectorized using Term Frequency-Inverse Document method [6], TF-IDF, to quantify each word in a document for analysis. We used scikit-learn library [3] to implement SVM and Naïve Bayes classifier as a comparison to our implementations that we created from scratch. Using scikit-learn we achieved an accuracy of 70.1% and 77.67% for Naïve Bayes and SVM respectively. Our method achieved an accuracy of 74.96% and 60.1% respectively. Our algorithm showed a 4.86% better accuracy for detecting sentiment when comparing Naïve Bayes but showed a decrease in accuracy of 17.57% for SVM. Further research would be needed to find the source of the accuracy difference for the SVM implementation.

1. Introduction

Understanding human behavior in an online setting can provide useful information for social platforms. It is typical human response to ignore or not gain new information from a source that is deemed negative [4]. People who receive less negative feedback tend to have higher self-efficacy and better performance in the work life. The goal of this paper is to predict if a comment from Stack Overflow is either negative, neutral, or positive to provide more insight for the platform.

Stack overflow is a question and answer website for professional and amateur programmers used to crowd source computer programming questions. Using an experimental dataset for sentiment analysis and emotion mining [1], we received 4424 labeled comments in Stack overflow. Using the dataset, we have incorporated a Support Vector Machine and Naïve Bayes algorithm, from scratch, to learn the dataset and be able to predict the polarity of a comment in Stack overflow. These algorithms are less complex then Recurrent Neural Networks but do not perform as well.

Popular pre-processing methods include Term Frequency-Inverse Document, or TF-IDF [6]. This method assigns weights to a word evaluating how important it is with respect to the document in a collection. The metric increases as the number of times the word appears in the document and decreases by the frequency of the word in the collection. It is computed by the following:

$$TFIDF\ score\ for\ term\ i\ in\ document\ j = TF(i,j) * IDF(i)$$

$$Where\ TF(i,j) = \frac{Term\ i\ frequency\ in\ document\ j}{Total\ words\ in\ document}$$

$$and\ IDF(i) = \log\left(\frac{Total\ Documents}{documents\ with\ term\ i}\right)$$

SVM and Naïve Bayes tend to be the go-to non-complex algorithms for text classification. Naïve Bayes is considered a generative model which uses prior knowledge of the statistics to find the probability that features belong in a class. On the other hand, SVM is considered a discriminative model that depends on the observed data while learning how to classify from the given statistics. SVM focuses more on decision boundaries where Naïve Bayes focuses on where the feature lies within the statistical distribution.

Following the implementation of Kibriya [5], we incorporated the statistics from TF-IDF score to create a multinomial Naïve Bayesian classifier. Naïve Bayes follows the classical Baye's theorem of:

$$P(class|features) = \frac{P(features|class)P(class)}{P(features)}$$

Since the probability of the features remain a constant, we can ignore the denominator. For training purposes, we focus on:

$$P(class|features) = P(features|class)P(class)$$

$$where\ P(class) = \frac{|Di|}{|D|}, D_i\ is\ a\ subset\ of\ documents\ in\ D\ in\ category\ c_i \in C$$

$$P(features_i|class_i) = \frac{(n_{ij}+1)}{(n_i)}, n_{ij}\ is\ the\ number\ of\ occurerences\ of\ word\ w_j\ in$$

$$T_i\ the\ collection\ of\ documents\ in\ D_i, and\ n_i\ is\ the\ total\ number\ of\ word\ occurences\ in\ T_i$$

Normally word occurrences are weighted as 1, but since we have more prior information with TF-IDF weights, we can weight the words by there respective TF-IDF score. For testing we do the following:

$$Given\ a\ test\ document\ X, n\ number\ of\ word\ occurrences\ in\ X$$

$$argmax_{c_{c_i \in C}}\ P(c_i)\prod_{i=1}^{n} P(a_i|c_i)\ \ where\ a_i is\ the\ word\ occuring\ at\ the\ ith\ position\ in\ X$$

We can calculate the priors by multiply out each feature's priors by the notion that each feature vector is independent from one another. An assumption made in Naïve Bayes.

The SVM algorithm followed the work of Cortes [7] with minor tweaks. The following algorithm was tweaked for training:

$$maximize_{a,b,t}\, t$$

$$subject\ to\ a^T x_i - b \geq t, \forall i = 1 \ldots., M$$

$$a^T y_i - b \leq -t, \forall i = 1, \ldots., N$$

$$\left|\left|a\right|\right|_2 \leq 1$$

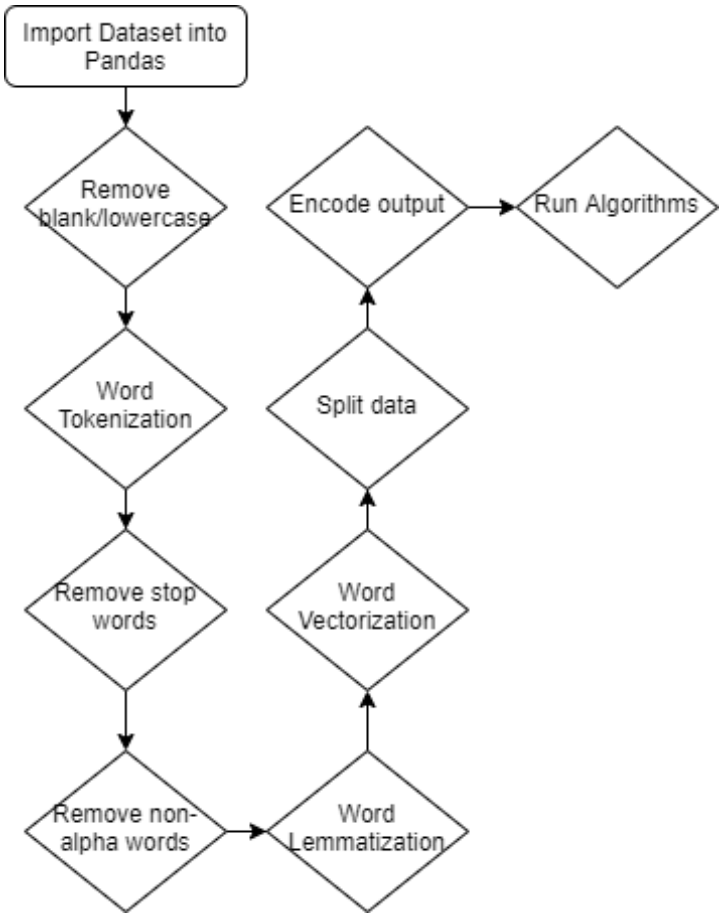
$$where\ a\ are\ the\ training\ weights, b\ is\ the\ biased\ term, t\ is\ the\ distance\ from\ nearest\ point\ to\ seperating\ line, M\ is\ the\ number\ of\ training\ vectors\ for\ class1, and\ N\ is\ the\ number\ of\ training\ vectors\ for\ class2.$$

SVM creates a hyper plane dividing both classes, maximizing the distance between the two closest points from different classes thus increasing the robustness of the classifier. Depending on which side of the line the datapoint lies on for the testing set determines the class.

In section two we will discuss the experimental set-up where we implemented our algorithm's and scikit-learn's [3] SVM and NB algorithms for a baseline metric. Section three explores the results and a comparison of both methods. Section 4 will conclude and provide a summarization of the report providing insights to what was learned and achieved.

2. Experimental set-up

The implemented algorithms and preprocessing were created using python with the following libraries: pandas, numpy, nltk [2], and cvxpy. Below demonstrates the pipeline of the process:



We first start by importing the dataset into python using the panda's library. The dataset contains 4424 labeled comments from Calefato [1]. Each comment is labeled by their polarity: negative, neutral, or positive. The initial preprocess step removes all empty entries within the pandas data frame. We then lower case each word for the tokenization process. Tokenization then breaks up the text into a list of words using the nltk library. With the same library we remove common words like the, if, etc. referred to as non-alpha language. Within this step we also remove non-English terms to remove username

references in the comments. The next step converts all language into a common base, referred to as lemmatization. For example, studies would be turned into study. We want similar impact words across documents to be quantized the same for further generalization.

At this point we take a document like this. The example is a negative labeled comment:

"Vineet, what you are trying to do is a terrible idea! If you ever find yourself having to engineer something as fundamental as that, take a step back and work out what else does it! I don't really believe in the whole spoon fed SO thing, but what mtwebster says is on the right track!"

To this:

"['try', 'terrible', 'idea', 'ever', 'find', 'engineer', 'something', 'fundamental', 'take', 'step', 'back', 'work', 'else', 'really', 'believe', 'whole', 'spoon', 'feed', 'thing', 'right', 'track']"

Now we have a vector of words per document that we can quantify using the TF-IDF method explained earlier. Using the entire dataset, we perform the word vectorization and create a vocabulary of 3897 unique words. Each word gets a unique numeric key. The words in the document are now converted to their respective TF-IDF score representing the statistics of the words impact.

The data was then split into 70% training and 30% testing. This ratio was determined by the providers of the dataset. The labels were encoded to 0, 1, and 2 representing negative, neutral, and positive in that order. The three classes are fairly balanced with 1186 neutral, 1069 positive, and 842 negative. No measures were taken for class imbalance. The testing set consisted of 508 neutral, 458 positive, and 360 negative. Now that the data is pre-processed, the learning algorithms are ran. For SVM testing, since classes are greater than two, a one versus one method was deployed. Three classifiers were created, positive vs neutral, positive vs negative, and negative vs neutral. A majority vote system was in place to determine the class of the testing documents. This method incorporated the cvxpy library to perform convex optimization.

For a baseline we used scikit-learn's library to implement SVM and Naïve Baye's classifiers. This gives us a tool to compare our method to in hopes of improvement.

3. Results and discussion

3.1 Scikit-learn library

Naïve Bayesian Classifier:

Table1. Confusion Matrix for Naïve Bayesian classifier using Scikit-learn library.

	Neutral	Negative	Positive
Neutral	163	33	3
Negative	176	395	83
Positive	21	80	372

Table2. Statistic Report for Naïve Bayes Classifier

	precision	recall	f1-score	support
0	0.82	0.45	0.58	360
1	0.60	0.78	0.68	508
2	0.79	0.81	0.80	458
accuracy			0.70	1326
macro avg	0.74	0.68	0.69	1326
weighted avg	0.73	0.70	0.69	1326

This method provided an accuracy of 70.13%. The majority of misclassification came from misclassifying neutral comments as negative. This is shown in neutrals low recall score.

SVM Classifier:

Table3. Confusion Matrix for SVM classifier using Scikit-learn library.

	Neutral	Negative	Positive
Neutral	226	40	11
Negative	128	436	79
Positive	6	80	368

Table4. Statistic Report for SVM Classifier

	precision	recall	f1-score	support
0	0.82	0.63	0.71	360
1	0.68	0.86	0.76	508
2	0.91	0.80	0.85	458
accuracy			0.78	1326
macro avg	0.80	0.76	0.77	1326
weighted avg	0.79	0.78	0.78	1326

This method provided an accuracy of 77.67%. The majority of misclassification came from misclassifying neutral comments as negative, like naïve bayes method. This algorithm outperformed the naïve bayes by approximately 7%. Through research, it is common for SVM to outperform Naïve Bayes.

3.2 Algorithms from scratch

Naïve Bayesian Classifier:

Table5. Confusion Matrix for Naïve Bayesian classifier using Scikit-learn library.

	Neutral	Negative	Positive
Neutral	267	91	15
Negative	76	329	45
Positive	17	88	398

Table6. Statistic Report for Naïve Bayes Classifier

	precision	recall	f1-score	support
0	0.72	0.74	0.73	360
1	0.73	0.65	0.69	508
2	0.79	0.87	0.83	458
accuracy			0.75	1326
macro avg	0.75	0.75	0.75	1326
weighted avg	0.75	0.75	0.75	1326

This method provided an accuracy of 74.96%. The majority of misclassification came from misclassifying negative comments as neutral. Even though it is a majority, compared to our baseline, the misclassifications are more spread out between the different combinations. Incorporating the statistics from TF-IDF metric increased the testing accuracy by approximately 4% compared to the baseline.

SVM Classifier:

Table7. Confusion Matrix for Naïve Bayesian classifier using Scikit-learn library.

	Neutral	Negative	Positive
Neutral	197	107	31
Negative	133	313	140
Positive	30	88	287

Table8. Statistic Report for SVM Classifier

	precision	recall	f1-score	support
0	0.59	0.55	0.57	360
1	0.53	0.62	0.57	508
2	0.71	0.63	0.67	458
accuracy			0.60	1326
macro avg	0.61	0.60	0.60	1326
weighted avg	0.61	0.60	0.60	1326

This method provided an accuracy of 60.11%. The majority of misclassification came from misclassifying neutral comments as negative. This implementation performed far worse than the baseline. This is likely due to improper implementation.

4. Conclusion

We took labeled comments from Stack Overflow and predicted the polarity to an accuracy of 74.96% for Naïve Bayes, and 60.11% for SVM. Compared to the baseline, the NB method outperformed by 4% while the SVM implementation underperformed by 17%. It is known knowledge that SVM outperforms NB for text classification demonstrating that the SVM was not implemented correctly. Majority of the classifiers misclassed neutral comments for negative, complemented by the recall score. This could indicate that key words in neutral comments can be considered negative to the classifiers. To build upon this project, a recurrent neural network could be deployed as a more complex model. RNN are shown to perform well on text analysis but are computationally expensive. Other vectorization methods could be used instead of TF-IDF, including word2vec. Boosters can also be deployed to improve the simpler models.

5. References

[1] Fabio Calefato, Filippo Lanubile, & Nicole Novielli. (2019). Experimental datasets for sentiment analysis and emotion mining - Emotion Mining Toolkit (EMTk) (Version v1.0) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.2575509

[2] Bird, Steven, Edward Loper and Ewan Klein (2009), Natural Language Processing with Python. O'Reilly Media Inc.

[3] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[4] Nease, A. A., Mudgett, B. O., & Quiñones, M. A. (1999). Relationships among feedback sign, self-efficacy, and acceptance of performance feedback. Journal of Applied Psychology, 84(5), 806–814. https://doi.org/10.1037/0021-9010.84.5.806

[5] Kibriya A.M., Frank E., Pfahringer B., Holmes G. (2004) Multinomial Naive Bayes for Text Categorization Revisited. In: Webb G.I., Yu X. (eds) AI 2004: Advances in Artificial Intelligence. AI 2004. Lecture Notes in Computer Science, vol 3339. Springer, Berlin, Heidelberg

[6] Jones, Karen Sparck. "A statistical interpretation of term specificity and its application in retrieval." Journal of documentation (1972).

[7] Cortes, C., & Vapnik, V. (1995). Support-vector networks. Machine Learning, 20(3), 273–297.