Analyzing the Sentiment of Ice Cream Reviews

Eric Sclafani

Data set: https://www.kaggle.com/datasets/tysonpo/ice-cream-dataset

1 Introduction

In the world of Natural Language Processing (NLP), sentiment analysis is an important task because it lets entities classify between different classes based on the type of words used. Examples of this task used in modern day include classifying product reviews and classifying how people feel about one thing versus another. People tend to use particular language when expressing sentiment and language is creative in nature (i.e. there are many different ways to say the same thing). Thus, accurately extracting sentiment from text can be challenging and there are many different ways to achieve it.

The sentiment analysis task I chose to undertake involves performing binary classification on an ice cream review data set. I classify user reviews as positive or negative. My first step was to process the data, which is a deceptively important and complex task. Next, the two separate techniques I use to perform the classification are the multinomial naive bayes classifier and support vector machine. These models differ from each other greatly. After calculating the results for each model, I compare them and attain predictable results.

2 Data Preprocessing

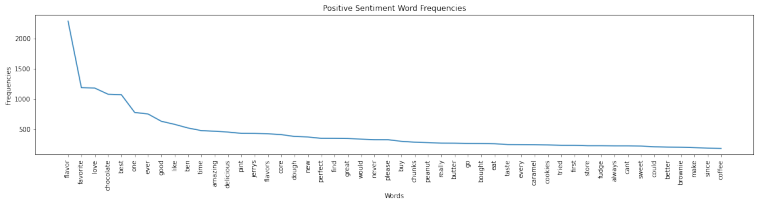
The data set I use for classification is a csv consisting of approximately 8000 ice cream reviews for Ben and Jerry's Ice Cream. This data is not cleaned for any machine learning tasks, so I wrote a **text_preprocessor** file to handle this. The default columns are as follows: "key", "author", "date", "stars",

"title", "helpful_yes", "helpful_no", and "text". The only relevant ones for this task, however, are: "stars", "title" and "text" because they hold the only useful information used for this classification.

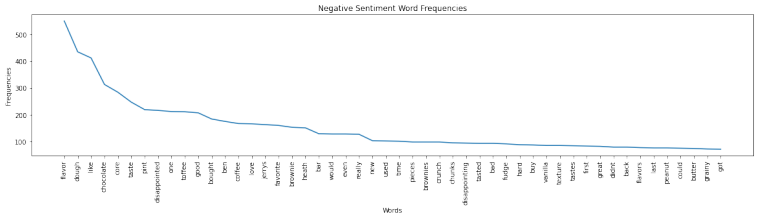
After removing the useless columns, I operate over the "stars" column. Any review with less than 3 stars gets converted to 0 (negative class) and 3 or greater to 1 (positive). The next step is combining the "title" and "text" columns since both exhibit valuable text data. After this, the final two processing techniques I do are cleaning the data of punctuation and stopwords (words that hold no value), and dropping na values. After dropping na values, I am left with 5332 reviews, for which I do an 80/20 train/test split.

3 Multinomial Naive Bayes Classifier

Naive bayes (NB) classifiers are simplistic in nature, but surprisingly effective for certain, more basic NLP tasks such as product reviews and spam filtering. The first important decision when constructing one is choosing how to prepare your data after preprocessing it. There are several options to go with and the one I chose was a bag of words (BoW). This data structure focuses on keeping track of word frequencies.



(a) Positive Word Counts



(b) Negative Word Counts

Figure 1: Word counts by class

Figure 1 shows the 50 most common words in each class generated by my code. Although the words themselves are difficult to read along the x axis, the point of the plots is to show that word frequencies almost always follow a long tail distribution. In other words, certain words occur very frequently, while other not so (even after dropping stop words).

After attaining the word counts, the NB classifier involves the following steps (all equations in this section are from [1]:

1. **Prior probability** of each class: out of all documents, how many belong to each class?

   $\hat{P}(c) = \frac{N_c}{N_{doc}}$

2. **Likelihood probability**: for every word in a review, what is the probability of that word occurring in each class of reviews?

   $\hat{P}(w_i|c) = \frac{count(w_i,c)}{\sum_{w \in V} count(w,c)}$

3. After taking all the word log likelihood probabilities given a class, add them up and add that sum to the log prior probability. Do that for both classes and take the argmax to see which class a review belongs.

One additional point to bring up about this method: since it involves dividing by the count of a word, what if that word doesn't exist in a class? This would result in dividing by zero, which would mess everything up. One way around this is to use laplace smoothing. This simple method involves adding one to each word count. This prevents zero from appearing in the denominator.

When it comes to the implementation in Python, I implemented it as a class with a suite of methods to handle the calculations. I modeled my classifier's structure a little differently than how they're made in sklearn. Instead of first instantiating a model object and then using class methods to modify the object, I made it so you feed the data into the model upon instantiation.

4 Linear SVM

The next classifier I used for this binary classification is the support vector machine (SVM). This algorithm is more complex than my NB one, so I decided to use sklearn instead of making it from scratch. Before implementing the SVM, I had to choose a way to vectorize my data. The option I chose is called Term Frequency Inverse Document Frequency (TF-IDF). I thought this was a good choice because it is the exact opposite of the BoW model used in NB. The goal of TF-IDF is to allocate more value to words that appear infrequently [2]. I am not going to go too in-depth with how this data structure works, but here's a short explanation: the model prioritizes words that appear frequently in a document, but punishes words that appear frequently across all documents. This means that stopwords, as well as frequent non-stopwords, are not given much meaning.

Now that the data is vectorized, it can be fed into a linear support vector machine (SVM). SVMs are a good choice for binary classification because it seeks to divide data linearly based off of a hyperplane. It also uses the outermost data points from each class as support vectors which help it find the most optimal margin. When it comes to the Python implementation, sklearn's model is the go-to implementation. I fit the SVM model to my data and produced a working classifier.

5 Results

As mentioned in the introduction, I received predictable results from these models. For training and testing each model, I used an 80/20 split, so 3732 reviews for training and 1600 for testing. Here are the results for NB:

- **Naive Bayes**:
  - Accuracy: 70%
  - Precision: 81%
  - Recall: 80%
  - F1-score: 80%

NB did better than I was expecting given its relative simplicity. Recall that the BoW NB primarily focused on word counts and extracting different

types of probabilities from those frequencies. With such a simplistic engine, one would think the model would achieve sub-par results. The results it produced are not state of the art by any means, but shows how certain tasks can be done with less complicated models.This NB can definitely be improved upon from this point and the results would be even better.

For the support vector machine, the results were more interesting. Here they are:

- **Support vector machine**:
  - Negative class:
    - Precision: 81%
    - Recall: 57 %
    - F1-score: 80%
  - Positive class:
    - Precision: 90%
    - Recall: 97%
    - F1-score: 93%

A major difference between my NB model and sklearn's SVM is that SVM calculates the precision, recall and f1-score for each class. This was the first time I heard of doing it for both class, so that's why I didn't do it for NB. Also, note the difference in scores between the two classes. I'm not sure about this, but this may be due to the large disparity in positive and negative reviews. In my training data, there are 2950 positive reviews, and 782 negative ones. Unfortunately, I am still not well-versed in support vector machines and sklearn in general, so I wasn't able to investigate this further.

6 Conclusions

Not all NLP tasks require state of the art powerful machines. Some tasks, such as the binary classification of product reviews, can be done with simple models such as Naive Bayes and support vector machines. When it comes to my implementations, many improvements can be made to both to improve the metrics. For example, the NB model does not account for negation or sarcasm, two more complex linguistic processes. For the SVM, I could have

spent more time with hyperparameter tuning and seeing if any of the optional arguments could improve the classification.

This is a project I've been wanting to do for a long time, specifically writing my own naive bayes classifier. My general philosophy with programming is this: If I can do it in base code without libraries, then I will try. I think this is the best way to learn things like machine learning models. I would have loved to program the SVM with base Python. Of course, in practice, libraries like sklearn will almost always out perform self made models. Because I manually coded the NB, I further solidified my grasp on certain machine learning concepts such as precision / recall / f1-score, training and testing, and the data preparation process.

References

[1] Dan Jurafsky and James Martin. *Speech Language Processing*. 2021.

[2] Bruno Stecanella. "Understanding TF-IDF: A Simple Introduction". In: (2019).