Tweet Classification

Michael Zuo (mz4@williams.edu) and John Freeman (jcf1@williams.edu)
Williams College

Code can be found at: https://github.com/muhmuhten/lessnaivebayes

1 Introduction

Twitter is one of the most popular websites that exists today. It has 317 million active monthly users that collectively post content at an average of 60 tweets per second. This means that Twitter has tons of data, and unlike other social media websites like Facebook, Twitter lends itself well to data analysis due to its format and use of hashtags. All tweets on Twitter must be composed of 140 characters or less, which means the structure of any one tweet cant differ all that much from any other tweet. This means that tweets should be easier to compare than something like a Facebook post which does not have this limit in place. Twitter also allows people to both contribute to and look for tweets about a shared subject with the use of hashtags. Hashtags allow the creator of a tweet to tag or label their tweet as being about a certain subject and allows others that might be looking for peoples option on some subject to look them up by simply searching for the hashtag of that subject. This means that tweets are also labeled, or tagged, when they are made by their creators. With the accessibility of tweets through the Twitter API, the abundance of data, the rigid structure of that data, and the fact that much of that data is already tagged, we wanted to figure out if there was any common patters within tweets of a given hashtag which a computer could recognize and use to classify a new, un-tagged tweet based on its content by assigning it a hashtag it is likely to have based on some metric. We wished to accomplish this task by using techniques such as Naive Bayes classification and bigram probabilities.

2 Justification

When thinking about a project to do, we realized that Twitter hashtags were simply categories, and that ”guessing” the hashtag of a tweet given just its content was actually just a classification problem. We thought that this would be a fun problem to explore and see if we could do this classification using machine learning and techniques we used throughout our course. We also liked the fact that because it was using Twitter data, we would have access to lots of easily available data which was already pre-tagged for us. We also liked that it was a problem that should have quantifiable results that we could measure to help with both debugging our model and measure how well it accomplished its task. We thought a useful application for a program like this would be to help reclassify tweets that should include a certain hashtag but don’t for

whatever reason whether it be because the tweet was composed before the hashtag was created or, perhaps, because the creator of the tweet had no knowledge of the hashtags existence.

3 Hypothesis

We had a few high-level hypotheses about this problem that influenced both how we thought about and the design decisions we made in trying to solve the problem. First, we believed this problem was solvable and that Naive Bayes would be sufficient in solving it. This premise is based on our belief that tweets of the same hashtag should be similar, so training on examples and then using those observations to classify others would be a good strategy. Second, we also believed that certain phrases could be significant in classifying some hashtags from others, so we thought a model that used bigram probabilities would perform better than one that used unigram probabilities. Third, we believed that certain punctuation, in particular #'s and emoji's, would be significant in classifying between categories. Finally, we believed that some hashtags would be much harder to classify between than others since hashtags that were on similar topics, like two political hashtags, would contain similar words and be of a similar structure, so we would guess our model to perform worse in such cases.

4 Background

Here, we will go through some of the techniques and background information required to understand how our model and setup for it works. All of the data we used for this project was obtained from Twitter using the Twitter API.

4.1 Naive Bayes Classification

We decided to use Naive Bayes as our classification model for this project. This means that we are measuring and classifying new tweets based on the structure and vocabulary of prior examples of tweets in that subject which were in our training set. We believed this to be a good strategy for classifying tweets based on hashtags since we thought tweets using the same hashtag would use similar language, including grammar and emojis, independent of what the actual hashtag was.

4.2 Bigram Probabilities

We decided to use bigram probabilities in our model. This means we are taking into account one previous word of context whenever we are measuring the probability of a single word as opposed to just considering the frequency of that word itself devoid of any context. We thought that a given hashtag might have some sequence of words that might be important to its classification but would get lost in a unigram model, so we tried using bigram probabilities to avoid this from happening. We also believed that due to how short tweets are required to be, it wouldn't make sense to go all the way to a trigram model.

4.3 Smoothing

We are using Add-One smoothing for both our unigram and bigram probabilities. In doing so, we assume that non-observed words and pairs have actually probability such that a unigram not observed in the dataset

but not in a particular category is less probable for the category than a unigram not observed at all, and likewise for bigrams, so that our method ensures that a word or bigram is never considered less probable for a category than one which has never been seen at all.

5 Model Implementation

Our model is an adapted version of NaiveBayesTextClassifier.java from Michael's Assignment 9. We are also using a modified version of Tokenizer.java and DataPoint.java. We also wrote a bunch of testscripts and datasets to use our model and print out statistics of how well it did.

5.1 Twitter Data

We used the Twitter API and the Ruby program T to gather data for our model to train and test on. Twitter has a very easy to use API that allowed us to grab tweets that contained a certain hashtag, and the program T helped with the parsing of that data. We then structured the data further until it was of the form that Tokenizer was expecting, and used that as a dataset to feed into our program. The one downside we found using to the Twitter API is the fact that it rate-limits the number of tweets you can gather for a given hashtag based on how recent the tweets are, which restricted the size of our datasets.

5.2 NaiveBayesTextClassifier.java

The big changes we made to NaiveBayesTextClassifier.java and DataPoint.java were to modify them so that we could calculate and use bigram probabilities along with unigram probabilities. We did this by adding a data structure to hold the bigram probabilities when building the model, calculating them, and added a switch option to specify whether the user wanted to use the unigram or bigram approach to classify the datapoints.

5.3 Tokenizer.java

We changed Tokenizer.java such that it regards #'s and emojis as significant, and doesn't filter them out. This change was made since we thought emojis would be important is classifying between different hashtags as it has continued to become a more and more popular way in which people express themselves on social media. In order to solve the obvious problem of #'s in tweets which are usually used to signify that tweets classification by the user, we simply removed the text after the hashtag, or category of the tweet, in order to signify that there is a Twitter hashtag in that location, but not actually give our model the content of that hashtag. This is necessary because often hashtags also carry literal meaning in a tweet, so that many tweets become unintelligible or empty when hashtags are eliminated.

5.4 Test Scripts

We wrote a bunch of helpful testscripts that allow us to run our program in such a way that we can use it's output to compute different evaluation metrics and create a confusion matrix for the different hashtags we are classifying between. Information on how these testscripts work and how they should be run is included in the README.md along with examples.

6 Results

We measured our results using three different metrics (accuracy, recall, and precision) along with providing a confusion matrix. A confusion matrix allows us to compare the actual categorization of different categories vs. the results of our program to see, for example, if there was a bias towards one of the categories that could help explain the stats for the results we are getting. The metrics accuracy, recall, and precision are defined as follows:

- Accuracy = $\frac{(true\,positives)+(true\,negatives)}{(true\,positives)+(false\,positives)+(true\,negatives)+(false\,negatives)}$
- Precision = $\frac{(true\,positives)}{(true\,positives)+(false\,positives)}$
- Recall = $\frac{(true\,positives)}{(true\,positives)+(false\,negatives)}$

The results we found were very surprising. Unigram did, overall, much better than bigram did, which seems to show some errors in either how we thought about the problem or the implementation of our model. Here are a few interesting examples of how our model did on different hastag classifications.

6.1 Love and Politics

We tried running our model on the two hashtags "#love" and "#politics" and compared the results we between the unigram and bigram probabilities. The size of the training set was 574 using #politics and 414 using #love and the testing set was 206 love and 274 politics.

Confusion matrix for the unigram model:

	Observed Politics	Observed Love
Expected Politics	184	90
Expected Love	7	199

Confusion matrix for the bigram model:

	Observed Politics	Observed Love
Expected Politics	111	163
Expected Love	2	204

The accuracy for the unigram probabilities was much higher than for bigram at 79.79% to 65.62%. They both also had high recalls for love and high precision for politics with unigram at 96.60% love recall and 96.33% politics precision and bigram at 99.02% love recall and 98.23% politics recall. Neither was very good in the sections of love precision or politics recall, but unigram did much better in both of these categories at 68.85% to 55.58% for love precision and 67.15% to 40.51% for politics recall.

6.2 Angry and Trump

We also tried running our model on the two hashtags "#angry" and "#trump" and again compared the results we between the unigram and bigram probabilities. The size of the training set was 1490 #angry and 806 using #trump and the testing set was 498 #angry and 275 #trump.

Confusion matrix for the unigram model:

	Observed Angry	Observed Trump
Expected Angry	479	19
Expected Trump	45	230

Confusion matrix for the bigram model:

	Observed Angry	Observed Trump
Expected Angry	303	195
Expected Trump	10	265

The gap in accuracy here was even higher than in love and politics with unigram accuracy at 91.72% and bigram at 73.47%. The unigram model did well in all of it's stats, scoring in the 90's for all precision and recall except for trump recall where it got 83.63%. The bigram model, on the other hand, had over 96% in both angry precision and trump recall, but stuggled when it came to angry recall and trump precision at 60.84% and 57.60% respectively. Here, the unigram model was an extremely effective classifier across the board whereas the bigram model seemed to just label a lot of angry as trump which explains the category numbers we see and contributes to the low overall accuracy.

6.3 Happy, Sad, and Angry

Finally, we tried running our model on three hashtags "#happy," "#sad," and "#angry" and compared the results we between the unigram and bigram probabilities. The size of the training set was 5047 #happy, 3658 #sad, and 1488 #angry. The testing set was 2912 #happy, 961 #sad, and 498 #angry.

Confusion matrix for the unigram model:

	Observed Happy	Observed Sad	Observed Angry
Expected Happy	2338	183	391
Expected Sad	148	594	219
Expected Angry	84	8	406

Confusion matrix for the bigram model:

	Observed Happy	Observed Sad	Observed Angry
Expected Happy	1788	94	1030
Expected Sad	90	323	548
Expected Angry	31	2	465

Here, we again get a huge discrepancy in accuracy between models with unigram at 76.36% and bigram at 58.93%. Here, bigram is very aggressive in labeling tweet as angry, especially for expected happy tweets with 1030 of the expected happy tweets being labeled as angry. Unigram does a little of this as well, but to a much lesser extent. Unigram had a much more spread out pattern where it tended to mislabel tweets whereas bigram just seemed to label too many tweets as angry. We believe this may show a flaw with using bigram on a training set whose data is temporally close since when data is limited, bigram tends to assigns undue weight to phrases which are common as a result of when they were published rather than what category they belong to.

7 Conclusion

We have a lot of thoughts about why we got the results that we got.

7.1 Training Sets

We believe that the biggest reason the unigram model did so well compared to our bigram model is due to small sizes of our testing sets, which restricted us from harnessing all the power of using bigram probabilities to take into account common phrases that tend to appear more often in certain hashtags than others. We believe that given larger training sets, the bigram model would start to pull ahead and beat the unigram model. The reason we were unable to actually test this hypothesis is because we found the Twitter API to be much more restricting for our uses than we expected it to be. While the API does give you access to tons of data and allows you to search through that data for certain hashtags, it limits the number of actual tweets you can get of that hashtag based on how recent the tweets are. This means that we not only can only get a certain number of tweets of a given hashtag, which we hope is popular now so we can get more data, but all of that data is also recent, which is another factor we did not consider.

7.2 Hashtag Selection

We also consider the fact that we might have chosen bad hashtags to try and categorize without more sophisticated techniques as a lot of the hashtags are very broad, especially considering our training data sets' sizes.

7.3 Closing Thoughts

We found this problem to be much harder than we anticipated. We believe that the real challenge with actually classifying the tweets is the fact that each tweet is written by a different user and how short each tweet is as a single datapoint. That isn't even getting into the fact that a given tweet can, and often is, tagged with multiple hashtags which may or may not be related. Given what we leaned, we believe this problem cannot be adequately solved using the techniques we implemented, but we do believe that more advanced techniques that sufficiently contextualize each tweet could be promising in trying to solve this problem.

8 Citations

- Twitter API Client: T, sferik, https://github.com/sferik/t
- Bigram Presentation: NLP Programming Tutorial 2 - Bigram Language Models, Graham Neubig, http://www.phontron.com/slides/nlp-programming-en-02-bigramlm.pdf
- Smoothing Presentation: Smoothing, Interpolation, and Backoff, Clare Cardie, https://www.cs.cornell.edu/courses/cs4740/2014sp/lectures/smoothing+backoff.pdf
- Thanks to Johan Boye for information from class along with code for Tokenizer.java, Datapoint.java, and NaiveBayesTextClassifier.java.