arOund

Active Classification of Tweets through Supervised Machine Learning

- Marc Abousleiman, Ivan Cisneros, Andrew Malek -

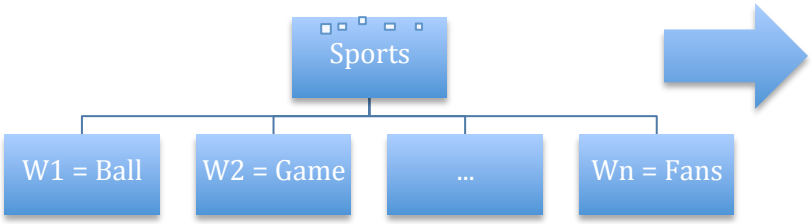
1. Overview

We implemented the Naïve Bayes Classification algorithm to classify tweets and other sentences of similar length and content type into sets of predefined categories. In this particular case we chose the categories “Sports”, “Weather”, and “Finance” because of the availability of data that could be used to train the program.

How the Algorithm Works:

The Naïve Bayes Classification algorithm is a naïve classifier that uses Bayes’ rule from probability theory to accurately match a target sentence or words with predefined categories. The algorithm makes use of the Maximum A Posterior estimate in that it must have a set of empirical references of known classification before it can begin classification of unknown data. Thus, it is composed of two parts, the learning phase and the testing phase.

- Learning Phase: In order for the algorithm to match an input with a category it must have a reference dictionary of words for that category. This dictionary of words is built using a large amount of training data of known categories (e.g. to build a Sports dictionary, one must first have a large amount of sentences and phrases relating to sports from which the algorithm can build a dictionary of words that will match with the Sports category). The algorithm assumes that each word in the dictionary as well as the input in question is independent. Thus, the first step of the algorithm is to find all unique words and their respective frequencies for each category, then find the conditional probability that the words belong to that particular dictionary of words using the frequencies as a basis.
- Testing Phase: A test phrase is inputted; this can either be a sentence, set of sentences, or a word. If the input is a single word, all the category dictionaries are searched through until matches are found; if a single match is found, the input word will be classified as being of that category; if multiple matches are found across categories, the one with the highest probability is the category that is given; if no match is found, the input is labeled as None. Similarly for sentences and sets of sentences, the conditional probability of each word is found; the sentence/s are classified based on the normalization of the probabilities of all the words contained in the sentence/s.



Word	Probability It is in Sports based on frequency
Here's	0.10
what	0.10
the	0.15
fans	0.75
think	0.10
about	0.10
the	0.15
WorldCup	0.95

What We Did:

In our implementation, we realized that a large amount of training and testing data could be acquired from Twitter, the social media site famous for its high volume of user input and succinct but dense tweets. Thus we made a twitter crawler that fetches our necessary data (tweets from nonrandom sources), and the Bayes Classifier, which is broken up into parts for readability; our classifier is modeled after the process described here (http://www.stanford.edu/class/cs124/lec/naivebayes.pdf). The whole program was designed around the gathering, analyzing, and classifying tweets into categories of our choosing (but this same program can be extended to whatever form of classification is desired – all that is needed is the large set of training data from which the algorithm can learn to classify for that particular category).

2. Video

Our Demo Video: https://www.youtube.com/watch?v=nRD3I6nvQE4

3. Planning

Originally, we planned to crawl twitter, filter tweets based on events, and match an event to geographical location based on the context of the tweet. This plan called for a graphical map display. This is described in our Draft Spec (found in report/ folder) For this, we planned on using a graph and PageRank algorithm, as well as Event objects which would carry with them information about the particular event (e.g. the source of tweet, the number of tweets that went into creating the object, an "importance" rating, a time counter from when it was created) which would serve as metrics for how the events would be ranked (relevance, importance, and proximity being factors that influenced this ranking), as described in our Final Spec (found in report/ folder).

We then elected to implement the Bayesian Classifier algorithm rather than the PageRank algorithm because we found that implementing an algorithm that we have already done before was not as engaging as we had hoped. The Bayesian Classifier employs machine learning and large data sets, so in many ways it is similar to PageRank, but different enough that we can be challenged. Because it is also widely used in linguistics research and in such things as email spam filters, we thought that it was a natural path to explore for our purposes. In light of our decision to change our core algorithm

from PageRank to the Naïve Bayesian Classifier, we came across the need to change our end goal as well. As such, a geographical display of events – and the Events data structure as a whole – was rather unnecessary. Because the classifier classifies data rather than ranks it, instead of displaying the geographic location of a tweet, we began working with the accurate classification of tweets and what could be done with that.

Thus, many of the graph and rank specific milestones were replaced with new goals. Rather than have tweets create events, we planned on having tweets create classification categories. Rather than have tweets match to a geographical location based on context of their content, we planned on having tweets correctly match with a classification based on the subject matter of their content. We would still need the crawler for this implementation, so that was kept and improved. We added to that the plans mentioned above as well as plans on getting the actual classification algorithm done by April 25th. The process went well, and we managed to meet our new goals in time.

4. Design and Implementation

The team as a whole did not have much experience with Python, but its simplicity and the fact that it is an OOP language allowed the team to adapt to it quickly. The frameworks we chose to work with were:

- GitHub: Used for repository and version archiving. Mainly for collaboration, but also to secure our work safely.
- Tweepy: A Python library for accessing the Twitter API. Provided classes and functions which made it easier to use Python for Twitter crawling. Can be at: http://www.tweepy.org/
    - The version we use is 2.3 and installation instructions can be found at: http://hexenwarg.wordpress.com/2013/12/04/tweepy-install/
- Python: We used version 2.7.5. Hopefully the code works on later versions as well, but we have not tested it on any version other than 2.7.5.

As specified earlier, our classification algorithm relied on large amounts of training data in order to build accurate and extensive dictionaries for each of our categories. As such, we assembled a list of the most popular Twitter users for each category and crawled through their history of tweets. Choosing to use legitimate sources from which we derived this data would reduce randomness and make for more accurate reference dictionaries (since it is assumed that legitimate specialized Twitter sources will strictly tweet about only one subject). We would pull 200 tweets from each source (this number was found to be experimentally about the average amount of tweets a source of that caliber would have) and with 40 sources per category this would total approximately 8,000 tweets per category for use as training data. The crawler itself continuously runs (While true … sleep(x) OR Cron Job), and is able to get new tweets that are created while the script is running. It detects if the tweet has already been written to the file so as to not write the same tweet more than once.

The tweets were then parsed, stripped of punctuation and other abnormalities, and all common words (such as articles and prepositions) were deleted. The results were separate csv files for each category, which contained the words found in the tweets as well as their frequency. This was inputted into the primary algorithm, which used frequency count, dictionary size and Bayes' rule to create new csv files of the words and their Laplace-smoothed probabilities.

The two functions that allow one to view the results are “classify_a_tweet.py” and “analytics.py”. The first file contains a script that will prompt for command line input of a string, which it will then try to classify. The second file runs an analysis of the classification algorithm’s accuracy; another set of Twitter users are crawled for tweets and the tweets are inputted individually into a classify() function for classification. The results are written to a file called “analytics.txt”.

Initial implementations of each module of the project worked reasonably well. As we progressed through the project, bugs were fixed and code was optimized to reduce repetitiveness and unnecessary steps. By far the most difficult part was getting a coded version of the classifier algorithm to be streamlined and accurate.

5. Reflection

The importance of categorization of things like tweets cannot be understated; newsfeeds can be a slew of random information, and, especially in Twitter, the ability to organize this information for purposes of preference or aesthetics would be a great addition. Through the use of a Bayesian Classifier, it would be possible to organize ones newsfeed in a way that is unique to each person’s preferences and tastes. If we had more time and resources, we would include more categories to test, as well as other metrics for classification; it would be possible to even include seemingly subjective categories, such as “importance”.

We were all surprised with Python’s softly-typed structure. Coming from OCaml, the softly-typed structure of Python made it slightly more difficult to read the code quickly, and to plan modules preemptively. The inputs and outputs of functions had to be carefully considered in order to make sure that we weren’t working with incompatible data types. Another surprise was the limit Twitter imposed on GET requests. Twitter limits GET requests to a few thousand, and once that limit is exceeded, one has to wait 15 minutes before making another request. This became burdensome when testing code early in the process, but the final implementation of the twitter crawler takes this limit into account and doesn’t request more than the limit.

Seeing how preliminary testing of the algorithm resulted in accuracy score >80%, we would say that the implementation was very successful. Of course, the process can be fine-tuned. The most direct way that the results could be influenced would be to gather even more training data, so that the reference dictionaries and probabilities are as encompassing as possible. Another way to affect the accuracy would be to use human categorized data (our assumption was that Twitter sources like ESPN and ChampionsLeague would only tweet about things relating to their specific category, in this case Sports) but there is no way to know whether there are any deviations from this pattern without reviewing each tweet that is passed in to the learning stage of the algorithm. With use of crowd-sourcing platforms such as Mechanical Turk, we could ensure that our training data is virtually guaranteed to be of the category that we assume it is.

The use of Tweepy and Python was a great decision. Because Python is easy to learn and widely used, overcoming bugs and problems was less time-consuming because of the plethora of python-specific online resources. Tweepy was also simple enough that we did not need to delve too deep into the Twitter API to be able to implement the functionality that we needed. We would definitely use the same frameworks for similar projects.

The most important thing we learned was the importance of maintaining a good version control system. Since we were a team of three, collaboration through a repository system was a necessity, but sometimes there were irreconcilable merge conflicts that would be a pain to sort out. Thus, we learned how to use GitHub correctly through lots of use. Additionally, maintaining readable, organized code was integral to working together; it eased collaboration and made it easier to spot sources of bugs.

6. Advice for Future Students

Our advice for future students is to plan things very meticulously, since that will help break down a large problem into more manageable parts. Not only will this aid in terms of motivation, but it will also ensure that the team as a whole knows what to work on and that everyone has an idea of what the finished product will do.