Naive Bayes: Examining the Partisanship of Online Content

Ankit Gupta, Hirsh Jain, Nishant Kakar, and Emily Wang

Computer Science 51

Harvard University

Demo Video

https://www.youtube.com/watch?v=xTW6iZJwUww&feature=youtu.be

Overview

We implemented the Naive Bayes algorithm, a cross validation system, and a web crawler. The Naive Bayes algorithm takes in a training set of designated liberal and conservative online articles and builds up a matrix of all of the words that appear in either liberal and conservative articles, as well as their frequencies. Then, given any random article, our program compares the frequencies of occurrences of words in the liberal and conservative dictionaries and calculates, from these frequencies, how likely the article is to be liberal or conservative. We create the dictionary of the article's words and compare it with our pre-calculated liberal and conservative dictionaries.

Our algorithm relies on the "naive" or "bag of words" assumption that the occurrence of each individual word in the sample texts is independent from the other words that appear. This is clearly untrue, as words are logically linked through sentences. However, making this assumption enables us to use Bayes theorem, which enables a simple (if naive) computation of likelihoods that an article is liberal or conservative given the word frequencies.

Cross validation runs on the training sets and tests the algorithm on subsets of the training set. The training sets are divided into subsets (currently, our cross validation system divides each training set into ten parts). The smaller set is then "validated," or identified as conservative or liberal using the remainder of the training sets, and the accuracy of these identifications is tested against the original designation of the articles as liberal or conservative.

We implemented the web crawler using a breadth-first search with a queue. The web crawler takes in a sample article on a website, adds it to the queue, scrapes the page for other links on the same site, and continues until it has built an array of 100 links from the site. We then iterate through these links, classify each as liberal, conservative, or indeterminant, and then compute what ratio of articles were liberal, conservative, or indeterminant.

Planning

We accomplished what we set out to do in our initial project proposal. Here are links to our initial proposal, final proposal and our functionality checkpoint. We had planned to implement the Naive Bayes algorithm with at least the basic implementation, and then as extensions have some form of cross validation and a web crawler, all of which were accomplished. By the final specification, we had figured out how to parse text from webpages and build up arrays with word count frequencies. Our proposals forced us to adhere to a strict schedule, and we were able to set reasonable goals for ourselves; in general, we felt like we planned well. By the functionality checkpoint, the core functionality of the program (the Naive Bayes algorithm) was completed, and we were able to identify the partisan bias of a document based on a small training set. This week, we focused on building up the training

set and implementing the extensions - namely, cross validation and the web crawler.

The various parts of our final project were completed on track relative to the deadlines set by the CS51 final project timeline, giving us ample time to develop some extensions to the core functionality of the project.

Design and Implementation

Our code consists of several disparate parts.

naive_bayes.py is the main file and is called in order to run the other parts of the code. It takes in user input and runs the correct parts of the project, based on which parts were requested by the user (ie running cross validation, crawling a web page, or testing the partisanship of a single article)

global_vars.py is the file in which we define the specific locations of the files that we refer to. For example, all of our code refers to a specific `liberal_file`, which should contain a list of liberal links for training. In `global_vars.py`, we indicate that our specific liberal file is stored at `files/liberal.txt`. This allows us to change the files to which we are referring more easily - we simply have to change the location in `global_vars.py`, rather than going through each individual spot that liberal_file is called.

check_doc.py is the file that contains a number of functions allowing us to conclude whether an article is liberal or conservative. It creates the vector for a certain article and then dots it with a general vector. Then, it uses the result to determine whether the article is conservative, liberal, or inconclusive.

cross_validate.py is the file that implements cross validation. As described in the overview section, cross validation runs on the training sets and tests the algorithm on subsets of the training set. We open writable files which will contain both liberal and conservative subsets. Then we access certain liberal and conservative subsets from the original liberal and conservative files. These subsets are subsequently run on each other and the articles are identified as either liberal and conservative. The accuracy of this identification process was then tested based on our original classification of these articles. We specifically have 10 steps of validation (since it is 10-fold cross-validation) which each take a different subset of the training set to test against the remainder of the training set.

matrix_builder.py is the file that has all of the content related to building the matrix of word counts for both liberal and conservative files. We then use this matrix, which contains the word counts of each word for each article. At the top of this files are several functions that begin with "update." These functions update many global variables that need to be reset every time we run a different validation set, or when we update the global variables at the beginning of Version 1. Lastly we have a function called `clean_matrix` which goes through and sets to zero the frequencies of all of the neutral words, found in `files/neutral_words.txt`, since we do not want these to influence the judgment of being liberal or conservative. Since a matrix is implemented as a list of python lists, we have a

function `list_builder` that given an article returns a list of its word counts, and then the `matrix_builder` iterates through the list of articles and constructs a matrix by repeatedly appending these lists.

parser.py is the file that contains the basic parser for any HTML page, called `parse_page`, as well as a few other functions to make our lives easier. `parse_page` is called on a url and a dictionary, and generates a new dictionary with key-value pairs representing words and their frequencies. The returned dictionary combines the dictionary passed in as an argument with the dictionary generated by the words in the article (it does so using a function we wrote above called `combine_dict`). It is also in this file that we implement stemming, which regards "tax", "taxing", "taxation", and other words with the same stem as the same word. We also have functions called `build_dict`, `divide_dict`, `create_ordering`, and `line_count`. The first takes in a list of articles and builds a joint dictionary for all of them, the second divides the frequencies (all of the values) by some specific value, the third assigns a specific ordering to our generated dictionary (which otherwise may not have one), and the last takes a file and counts the number of lines, all of which were needed elsewhere.

vector_builder.py iterates through the matrix and totals the word frequencies for each individual word across all articles in a specified range (the first half of the matrix is always liberal, and the second half is always conservative, and so we needed to specify what range to sum over). To build the liberal and conservative vectors, the ranges of the matrix (i.e. 0 to the number of liberal files for the liberal vector) are passed into `lib_vector` and `con_vector` respectively, and these functions then return the vector of word frequencies.

web_crawl.py will access a designated url in `global_vars.py` and crawl that webpage for links to other web pages. Web crawling is implemented using breadth first search on a queue. Note that we only add links to the queue if they are from the same main website (meaning if we are on an article on CNN, we will only link to other parts of CNN). After building up an array of 100 links, the array is then passed to `checkdoc.py` to count up the number of liberal articles, conservative articles, and indeterminate articles. Proportions of liberal, conservative, and indeterminate articles are then printed.

neutral_words.txt is just a text file that consists of a hundred words we have designated as nonpartisan. For example, words like "and," "but," "or," "the," and so on would not be helpful for our analysis as they would probably appear with high frequency in both liberal and conservative documents and reduce the impact of more important words.

conservative.txt and **liberal.txt** are both libraries of conservative and liberal articles, respectively. These are the training sets that we use for the Naive Bayes algorithm. Also, subsets of these training sets are using the cross-validation to determine the legitimacy of our algorithm.

testcases.txt is the set of links to documents whose partisanship the user wishes to test in version 1. Version 1 returns the ratios of liberal, conservative, and indeterminate articles from this set.

validate_cons.txt, **validate_lib.txt**, and **validate_totest.txt** are documents written into during the cross validation process - they are not to be changed by the user.

Most of the complexity in our project comes from the complex algorithm we are implementing, as we are conducting machine learning to classify the documents. However, we also use a variety of data structures in order to accomplish our goals. In particular, we use a queue (in particular, one from the collections library that allows for fast pop and push) for the web crawler. We also make extensive use of dictionaries for the word frequencies. We also use what we call an `ordering_dict` to impose an ordering on the words (it gives each word in the dictionary a unique index) so that the dictionary of word frequencies can be converted into a matrix (2×2 array), that we then used for very fast operations.

Reflection

First and foremost, we were pleasantly surprised at the accuracy of our algorithm despite the naive assumption. We were concerned that because both liberal and conservative news coverage responds to current events, the vocabularies of news articles would likely be very similar. However, our algorithm is in fact very accurate - cross validation has verified the accuracy of the algorithm given our current training sets.

We were a little less pleased with the robustness of our current web crawler. When testing the web parser, we were not able to successfully run it on all the sites we wanted to run it on, since links are not formatted uniformly across all websites. In a particularly tragic turn of events, we were unable to test the partisanship of The Onion. If we were given more time, we'd write the web crawler so that it'd be able to crawl websites with unusual HTML formatting.

We were pleased with the multiplicity of Python libraries at our disposal and the ease of Python's syntax. Since two of the group members began the project not knowing Python, learning the language and finding libraries to help with some simple tasks (taking the dot product of two vectors) and some not-so-simple tasks (implementing stemming on words) was more straightforward than expected.

We were also very happy about the level of modularization in our code. Implementing extensions was significantly easier because we were able to rely on pre-existing frameworks. In particular, since we separated aspects of our core functionality into separate files, we were able to just import whichever files we needed for the extensions, which kept matters organized. Choosing to modularize in this way was a very good idea, as it allowed us to quickly debug issues, organize code into broad categories, and add extensions as needed.

One of our original ideas involved having several files where variables were evaluated immediately (not in a function). We thought that this would not pose any problems, but ultimately found that it was a bad idea. In particular, we found that the immediate evaluation caused issues when we needed to run the algorithm several times (such as during cross-validation), as the classifier would just use the computed values rather than recomputing them. So, later we changed the code so that essentially the only place where evaluation occurs outside a

function is in the `naive_bayes.py` file, which is one that the user calls. This showed us the importance of thinking ahead in developing our code.

If we were to redo our entire project using the same Naive Bayes algorithm, we would probably use more or less the same framework. However, the most interesting extension/continuation of our project would involve making our algorithm a little less naive.

In terms of splitting up the work, we all had major contributions to each of the files. Often we ended up working at the same time on just one person's computer so that we could bounce ideas off of eachother. Clearly, there was a lot of code written, and it was all the result of the combined efforts of the members in the group. Additionally, each member of the group was responsible for finding roughly 50 articles for the training sets, which allowed us to build the overall set of 200 articles (100 liberal, 100 conservative). Lastly, we all tested various parts of the code on our own computers, and independently found and reported bugs to each other, which we then resolved either individually or as a group.

The most important lesson we learned is that when developing a long and complex project, it is critical to design code not for what is the next task, but for tasks that may arise several days or weeks down the line. In particular, it is critical to design code that allows for user cases to arise that were not previously considered. We did this very effectively for the parser file, which is why after implementing it once, we did not have to modify the file much. However, in some of the other situations, a little bit more foresight could have saved us some time. Now, however, we are confident that our code could be built on even more, and could be used a variety of other interesting explorations.

Advice for Future Students

Start early and do your best to follow your self-imposed timeline. Starting early and planning ahead helped our group end up with enough time to implement all of the ideas we generated earlier in the planning stage. It was really exciting to see the results of our ideas. Getting to test the partisanship of random news sites (Unsurprisingly, the Cato Institute is very conservative!) using the web crawler was one of the more fun outcomes of the project, but we wouldn't have had the time to implement the web crawler had we not budgeted a week to spend on implementing extensions.

Also, learning the concept behind the algorithm before diving in was helpful and definitely saved time. Although we were familiar with Bayes' Theorem from our statistics courses, learning how exactly the theorem applied in this case (including the intricacies, like how to build up the matrix of word frequencies, when to take the log of the frequencies, and when to take the dot product of the frequency vectors) helped us modularize the project in advance. And the modularization resulted in better-organized, more readable code.