Patrick Tan ptan@college.harvard.edu

Timothy Kang tkang01@college.harvard.edu

Kevin Zhang kevinzhang@college.harvard.edu

CS51 Final Project: A Dynamic Naïve Bayes Algorithm

Demo Video

[insert link here]

Overview

For our project, we implemented the Naïve Bayes algorithm (the basic specification can be found at http://guidetodatamining.com/guide/ch7/DataMining-ch7.pdf) and made modifications to make the program learn dynamically, as it analyzed more and more documents.

Instructions and Semantics

File Structure

The files included in the `code/` directory includes a file called `reviews.txt`. This file contains the reviews from Amazon that we used to test our program. These reviews can be found in the downloadable Watches.txt.gz file at http://snap.stanford.edu/data/web-Amazon-links.html.[1]

The main executable programs are contained in analyzer_d.py and analyzer.py. These files use alphatree.py (an alphabetical tree structure that contains utility methods as well as relevant methods to the Naïve Bayes algorithm), document.py (an abstraction of a text document that contains a utility tokenizer method), and util.py (a collection of utility functions we used throughout the project).

In order to run our program, grouper.py and collector.py are used to generate seed files and test files from reviews.txt, respectively. The code in grouper.py and collector.py are specific to the semantic structure of reviews.txt (so it won't work on just any file you give it!). Our code requires that you have Python installed on your machine. On Windows, instructions can be found at https://docs.python.org/2/using/windows.html for setting up Python. On OS X and Unix-based systems, instructions can be found at https://docs.python.org/2/using/mac.html.

[1] Paper by J. McAuley and J. Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. RecSys, 2013.

Running our Programs/How It Works

Generating seed files and test files

To generate the seed files and test files necessary for our program to run, open up a bash terminal on your system (Windows uses Command Prompt, while OS X uses Terminal). Navigate to the directory containing our code using `cd` and `..` as necessary. Once in the correct directory, start Python (simply type `python` and hit the return key). Run the following script:

```
import grouper
import collector
grouper.format(“reviews.txt”, 10)
collector.format(“reviews.txt”, 100)
```

The above script generates 10 negative reviews and 10 positive reviews (both named `0.txt` through `9.txt`), along with 100 test documents that we can use to test our program (named `0.txt` through `99.txt`). The negative reviews are put into a folder called `negative`, the positive reviews are put into a folder called `positive`, and the test documents are put into a folder named `test_docs` (original, we know). You can open these files up and examine them yourselves. To generate different numbers of files, simply change the numbers in the script above.

Grouper.py and collector.py work by iterating through the large file `reviews.txt`, reading data and outputting files until the requisite number of files have been obtained. Reviews with a rating of 1.0 out of 5 are considered negative, and reviews with a rating of 5.0 are considered positive. Only these reviews are examined and outputted into files. In each review, words that contain characters that not alphabetical, - or ‘ are removed.

The resulting files only contain words with the above 28 characters. This is done because of the structure of an alphabetical tree structure that we use to store our data; an exact number of characters must be the known alphabet. If we had used a letter dictionary structure instead, an infinite number of characters for our alphabet would be allowed. However, an alphabetical tree structure requires that the alphabet consist of a finite number of characters, and that each character should correspond to an array index in the internal array structure. Grouper.py outputs the pure text of the review into the text files; collector.py outputs the heuristic of the file (this is done to evaluate the accuracy of our program) along with the pure text on a line after it.

Seeding the program

A seed file that tells the analyzer where the seed files are (for the Naïve Bayes algorithm) is found in `seed.txt`. The structure of seed.txt is fairly simple; each heuristic is printed to a line, followed by a subsequent line containing a number describing the number of files to integrate into the seed. This file is automatically generated by `grouper.py`; ensure that the number contained in seed.txt is equal to or less than the actual number of reviews contained in the respective folder, or the analyzer program will crash upon initialization! Edit these numbers if you'd like the analyzer to integrate fewer numbers of seed files.

Analyzing the files

The main programs are contained in analyzer_d.py and analyzer.py. Analyzer.py is a simple, basic version of the Naïve Bayes algorithm that analyzes one document at a time. Analyzer_d.py does better, learning dynamically and analyzing the contents of an entire directory. To run the simple analyzer, open up a bash terminal as before and run the following script:

```
from analyzer import Analyzer
a = Analyzer("seed.txt")
```

You should see several lines of output detailing the process that the analyzer program goes through. Once the program completes its initialization (which should take less than a second or so), run the following:

```
a.analyze("test_docs/0.txt")
```

The above script should result in output that looks like:

```
{'positive': 7.549351224232794, 'negative': 8.350594168474613}
```

These numbers represent the calculations done as part of the Naïve Bayes algorithm; they are the sum of the logs of the probabilities of each word being present in each heuristic. The higher the number, the higher the probability that the review is that particular heuristic. In this instance, it looks like `0.txt` is more likely a negative review (which unfortunately it isn't…). The greater the margin between the calculated values, the more confident we can be that our program is correct. In this case, the inaccurate guess is not that surprising, since 7.56 is fairly close to 8.35. The simple analyzer is a just a basic implementation of the Naïve Bayes algorithm.

Analyzer_d.py is where things get interesting. To run the code in analyzer_d.py, run the following script:

```
from analyzer_d import Analyzer
a = Analyzer(“seed.txt”)
```

The result should be output similar to the output from analyzer.py. To execute the dynamic program, run the following code:

```
a.learn(“test_docs”, False)
```

There are two things of note here:

- “`test_docs`”: this is the name of the directory containing the formatted review files to analyze
- “`False`”: this flag defines the type of learning the analyzer undergoes

We wrote our dynamic analyzer in a way that allows for two types of learning:

- Guided: if the flag in the code above is `True` instead of `False`, the analyzer program is guided through its learning. While it still makes guesses based on the files it has seen so far, the program is told what the correct heuristic is, and it learns based on the correct values.
- Unguided: if the flag in the code above is `False` as opposed to `True`, the analyzer program is not guided through its learning. The program will assume that whatever it guesses is correct.

The output of the above script contains one line for each file it analyzes of the format:

```
File n: N.  0/1
```

The values correspond to the following:

- `n`: this is the number of the file analyzed.
- `N`: this is whether the program guessed correctly or not. A flag of `N` means that the program guessed incorrectly, while a Y means that the program guessed correctly.
- `0/1`: this is the number of correctly guessed files so far versus the total number of files analyzed.

Note that running the learning script several times results in higher and higher scores, although the scores fluctuate widely; in general, if the analyzer was told to analyze the same files over and over, we found that the accuracy approached 85-86% after a few iterations. Alternatively, if the program was told the correct heuristics, accuracies varied widely but steadily increased over time (which is to be expected). The exact numbers are results of the reviews that we used (had we used a different set of reviews from the Stanford site noted at the beginning of this paper, we would have obtained different results).

In Review

erratic at best (we tended to code in spurts rather than consistent, daily activity). Our milestones were mostly achieved, although two out of three of our extensions failed. We weren't able to implement n-grams or Twitter integration for various reasons. After finishing the dynamic extension of the Naïve Bayes algorithm, we targeted Twitter integration; however, we were unable to get the Twitter API to accept our client keys successfully or pull old tweets from the Twitter database. As a result, Twitter integration failed, and we never worked on implementing n-grams into our code.

We decided to use Python for this project because two members had had exposure to the language before, it has a reputation for being an easy language to code in, and we wanted to learn the language. Python proved difficult in some regards; in particular, types were a concern when writing our code. Python has loose type definitions (both Python and Javascript allow variables to change type), and it allows objects of different types to be contained in the same array, dictionary, etc. This was surprisingly difficult to work around; we hadn't realized how much we relied on type definitions when we were coding. In the end, as long as we were careful about commenting and made sure the methods plugged into each other correctly, dynamic typing was an issue we were able to work around by keeping careful track of our classes and variable types. We made no use of interfaces since there was very little inheritance in our project; testing was fairly straightforward after we wrote several functions to output data in readable formats (writing a to_string function for our alphabetical tree class was particularly useful… and annoying to do).

There weren't many surprises that we encountered; coding was relatively straightforward. If anything, the biggest surprise was how easy and intuitive it was to code in Python. Some of the more interesting moments included realizing that the `&&` operator didn't exist in Python; using `and` was surprisingly unintuitive after coding in OCaml, C, Java, and other languages for so long.

Deciding how to store words was a difficult task; we wanted to write our own structure that would be suited to the task, but figuring out the most efficient way to do so was difficult. In the end, we decided on a lazy alphabetical tree in which nodes stored both subsequent words and word counts that ended in that particular node. For example, the words "bar", "bat", and "ban" would be stored in three nodes; the first node would only contain a reference to another node that represented the letter "b" and contain no values, the "b" node would a reference to another node that represented the letter "a" and contain no values, and the "a" node would contain no references to other nodes but rather three values that corresponded to 1 each of "r", "t", and "n". Lazy implementation was useful; nodes were only initialized when needed, saving a lot of space that would otherwise have been useless. This choice ended up being fairly good for us; the structure allowed easy traversal and adding, and allowed us to add custom values to the array for each word (which was important for the algorithm's success).

If we had had more time, Twitter integration would have been important. Put simply, we didn't leave enough time for ourselves to complete this extension, and hadn't realized how difficult it was. In addition, there were issues with the way the Twitter API was implemented; the usual method of retrieving tweets about a particular subject involves streaming tweets from the site live. We needed to retrieve old tweets and analyze those, not analyze tweets as they were posted. The live method may be useful for analyzing tweets about something like the release of a movie, but it was not relevant to our project. In addition, our code could have used some cleanup and abstraction; in particular, grouper and collector are fairly messy, and we could have implemented our analyzer in a way that could use multiple seeds as long as all the relevant files for each seed are separated into distinct folders.

Patrick was responsible for finding our test files and clarifying the idea, driving the project and helping to draw out exactly what we needed to do. Tim was responsible for writing programs that both recreated the test files in a format that we could use and finding methods to convert those test files into data that we could store. Both Tim and Patrick were responsible for documenting our process and creating the video for this project. Kevin was responsible for coding the analyzer program and creating the alphabetical tree structure, and creating the write-ups.

The most important thing we learned from tackling this project was probably just knowledge of Python; each group member has a fairly good grasp on the syntax of Python, libraries available for use, and how to compile and run Python programs. The process was enjoyable and interesting, and we had a lot of fun learning to code and spending late nights with coffee, bad music and our laptops.