David Clark, u1225394
Prof. De Santo
LING 5300
May 3, 2023

Email Authorship Analysis with N-Gram Models

Background:

Authorship Analysis involves predicting a text's author based on its similarity to previously studied texts by the same author. In a general sense, this is an essential skill for media-literacy and general learning. In schools, children learn to identify authors by their rhetoric, by their persuasive styles and goals. In offices, too, grown adults try (and often fail) to distinguish between well-intentioned communications and fraud. In politics and news, of course, but also in our email inboxes, we are constantly subjected to communications from diverse parties, many with competing objectives. Learning to recognize exactly who or what is communicating with you is the first step to understanding what is true and what is actionable, so building systems to do authorship analysis well (or, more reasonably, to assist and enable humans to do this well) can be highly productive.

Building something that radically alters our relationship to truth is beyond the scope of this project, so the goal of this paper will instead be to build a simple (but functional) model for a simple (but practical) use case: to build an n-gram model that predicts the authorship of emails. Theoretically, such a system could be developed to detect spam or prioritize messages from high-priority authors (though other metadata may be the preferred implementation strategy. In practice, the system will be a teaching tool for myself in particular: a short crash-course on working with corpora and simple language models.

This is a supervised learning problem, so any program that tackles it can be evaluated with a fairly common scheme. I plan to calculate the typical measures of precision and recall, both for the program as a whole and for individual authors. Ideally, the program rarely or never attributes an email to an unrelated author (precision), and it always attributes an email to its correct author (recall).

At the scale of the entire program (and with certain assumptions), these metrics end up being the same; if all emails in the test data have an identifiable author, and if the program must assign a single known author to every email in the test data, then every misattribution (hurting precision) must entail a missed-attribution (hurting recall).

At the scale of individual authors, however, precision and recall are not so closely linked, so we should be able to calculate and compare separate values for each. The primary goal is to achieve high precision and recall across all authors, but having the statistics for individual authors may give us additional insights.

Methodology:

Before discussing how the code works, I'm going to describe how to access and read the code. The code for this project will be available as a downloadable .ipnyb with this paper, as well as online on colab. The code depends on files extracted from the Enron Email Dataset, which will be available in the same places. The code is commented and layed out in logical order, though it perhaps makes too liberal a use of global variables. There are print statements after nearly every block that give a window into what progress the code is making on each step, though some are more descriptive than others. Variable names tend to err on the side of being verbose.

As far as external resources go, the code uses an email corpus that will be described later and a few standard libraries and subsections of NLTK (in particular, the word tokenizer function and stopword corpus).

The general outline of the program involves defining a set of authors, loading a list of emails written by each author, removing metadata (via obtuse but mostly functional regex), tokenizing the content, separating testing and training data for each author, and using the training data to calculate the relative frequency of each n-gram for each author. With these frequencies for each author, we can receive an unknown test-data email and calculate each author's probability of choosing that particular sequence of bigrams; the author that generates the test email with highest probability is our 'predicted author.'

Each of these steps involves some complexity, which I will address now. (We may see that my code increases this complexity, or introduces complexity without need, but that's neither here nor there.)

To start, the process of defining the authors was more of a set-up problem than a coding one. I chose a subset of authors from the corpus that each had a reasonably large "_sent_mail" folder. I specifically uploaded these authors' folders to Google Drive in order to have them more easily interact with Colab, and to avoid loading large amounts of unused data. In code, the program runs terminal commands to navigate the file structure, and it recognizes as many authors as have files in its project directory.

Loading the emails for each author is potentially the most tedious part, as the fileIO is likely to stall significantly. If you run this code, it may be useful to know I've had this part take five to ten minutes on Colab. The process is straightforward, however, as it involves running a terminal "ls" command in each author's directory and iteratively reading each file into an array. We assume emails in an author's "_sent_mail" directory are written by the owner of the directory, though this isn't strictly true in the case of forwarded emails. Is it the best way? Should I be using pandas frames? Who knows! It seems to work, which is my main priority.

After loading the emails into memory, I attempt some basic data-cleanup to avoid messy or undesirable behaviors. For example, each email begins with a set of formulaic header fields that both pollute the output and potentially allow the program to "cheat", e.g. by learning that emails authored by "allen-p" are extremely likely to start with "From: allen-p". I use regular expressions to remove headers, names, and dates, where possible, but the actual code is rather ad-hoc (and uncommented regex strings are a shortcut to the deepest pits in hell, so I apologize for increasing everyone's misery here). If I had more time, I would write more complete filters, because I know some information (e.g. oddly formatted dividers, email addresses) can escape and enter into the final training / testing data.

The cleaned-up emails are tokenized with NLTK's word-tokenizer. At this stage, I also remove stopwords from the list of tokens. If I've done a good job of cleaning up the data, the most common tokens should relate to the body of the email; if not, I know I'd like to fix my regex.

After this much preprocessing, I finally split the emails (now lists of tokens) between testing and training sets. For each author, 10% of their works are used in testing and their other 90% are used to learn bigram frequencies. The training emails are assimilated into a single list of tokens (annotated with EMAIL_START and EMAIL_END tags) to make the next steps easier.

The process of learning bigrams is also a simple enough; as a class, we did something similar in assignment three. NLTK has a bigram() function, and for-loops

and Counters will quickly yield absolute frequencies for each bigram. The absolute frequency divided by the number of bigrams yields the relative frequency, and our job for this phase is finished. Lists of bigrams' relative frequencies are calculated for each author separately.

Finally, having these bigram probabilities, we can produce estimates for the likelihood that each author wrote a given email. For a given email of unknown origin, for each author, we take each bigram in the text and incorporate its probability into an overall probability of the author generating the email. (The math specifically involves adding together the log() of each probability, and this is done to avoid dealing with floating-point underflow errors.) We attribute authorship to whichever potential author is most likely to generate the given email.

With this method of determining authorship, we can run our testing data by our program to see how well it performs. Details for this are in the code and under "Results", below.

Data:

I wanted to do authorship analysis on emails in particular, so I needed to find a corpus of emails. I needed a large supply of emails that could be associated with a consistent set of authors.

Rather than digging through my own spam folder, I used data from the Enron Email Dataset, which contains approximately 500,000 messages or 1.7 gigabytes of data. The full dataset can be downloaded at https://www.cs.cmu.edu/~enron/ or explored online at http://www.enron-mail.com/email/. The emails were generated "in the wild" by approximately 150 Enron employees, mostly senior management. There are folders for each author, and subdirectories for various topics ("bankruptcy") and email-related functions ("all_documents"). The content is helpfully structured and diverse within reason, with some emails being shorter or showing different styling choices.

My program and analysis use only a subset of that data. In part, this is because I had issues with effectively cleaning and processing so much of it. To simplify the task, I gathered all emails in this analysis from directories labeled "_sent_mail"; each email could then be associated with the owner of the directory. Some authors were more prolific than others, and my data includes authors responsible from anywhere between (roughly) 50 and 1500 emails. I used emails from 12 different authors, representing over 7000 documents in total.

The same program and analysis could be applied to larger collections of emails, even from the same corpus, but more care would be needed to ensure the larger collection was equally clean. Ideally, names and non-lexical stylistic markers (e.g. typing "---" as a divider) are scrubbed before the data is used. Additionally, pulling emails from other folders would also require more thoroughly examining each message's metadata, inspecting 'from' fields to assign authorship.

From the data I selected, I used 10% for testing and 90% for training. More concretely, I had ≈ 740 testing emails and ≈ 6600 training emails. This gave me as many as 141 test emails that could be uniquely attributed to one author; in the sparsest case, however, I had as few as 6.

In the future, I would like to spend more time cleaning and expanding my dataset. I believe the data could be made more convincing with more time, both in terms of human and compute time.

Results:

Overall, my program predicts the correct author of a testing email with ≈ 76% accuracy. “Accuracy,” here, refers to both precision and recall, due to a quirk of the way positive results are counted (discussed under “Background”). This seems to perform significantly better than the most naive of strategies (discussed next).

Rather than generating our baseline in code, we can use simple probability to calculate outcomes for two naive strategies that don’t interact with the emails’ texts at all. First, we will compare against a “random guessing” strategy, then against a “modal guessing” strategy; neither of these require any knowledge of the contents of the emails, or even any knowledge about natural languages.

In the first case, we can imagine an author-identification program that simply receives a list of authors and, for each test-email, outputs a known author uniformly at random. For the 12 authors my program uses, each author would be chosen approximately 8.3% of the time. Each test-email has only one true author, so the program will guess correctly only about 8.3% of the time. My program performs well above that baseline.

Alternatively, our hypothetical baseline-program could guess only the most prolific author. When it reads its list of authors and corresponding lists of emails, it can note which author has the most training data. If the amount of training data is uneven (which it is), and if the amount of training data correlates with the amount of test data (which it does), then the program could decide to consistently guess the most prolific author. This program would correctly identify this one author’s emails 100% of the time (excellent recall, for one author) but would only do well with those specific emails. Their overall combined precision and recall would equal the proportion of test emails penned by the most prolific author. In my data, I have ≈ 740 test emails, 141 of which are from the modal author: at best, this strategy could expect accuracy near 19%. Again, my program performs better.

Compared to two naive approaches, then, my program performs well. If I had more time, I might want to make additional comparisons between particular implementations of my program. For example, I might examine how changing the n-gram length changes the overall precision/recall; right now, I use n=2 (bigrams), but it may be that n=3 or n=4 perform better. I might also compare versions of my program with and without stopword-filtering; I removed nltk’s english list of stopwords, and this tends to improve performance on semantic or sentiment tasks, but it may actually hurt when analyzing a particular author’s style (e.g. some authors may prefer certain grammatical constructions).

Having made these program-level comparisons, I’d like to briefly compare precision and recall across various authors. The full data for this can be found in the .ipnyb, under the section labeled “TESTING PRECISION & RECALL.” The range of the authors’ scores is interesting, of course, but most interesting to me is that high recall values seem to correlate with larger pools of training/testing data. I would guess that having more training data for an author allows the program to more “confidently” recognize that author’s emails, and that the smaller training data pools are more likely to have outlier-bigrams with distorted probabilities. If I had more time, I would investigate this further, and I would actually run the statistics to see if this trend is statistically significant.

Discussion:

I've been using the phrase "if I had more time" throughout this paper, and every specific thing I've said that way would still help, but it seems that the most important thing I needed was time. More time has been more rewarding, consistently throughout the project.

To reiterate, I'd like to run my program with more data, more preprocessing, and more configurations of parameters (e.g. different n-gram lengths). I'd like to analyze my data in more statistically rigorous ways. If I were to do it again, I would have also included more figures to ease my poor grader's eyes. Hindsight is twenty-twenty. Oh, and who would have thought shuffling gigabytes worth of email between free-tier cloud services would be difficult?

Reqs:

Write up requirements (5 to 10 pages, single-spaced):

1. Background: What is the problem, how are you approaching it? How will you evaluate success? Which metrics will you use and why?
2. Methodology: details of the implementation, which resources did you use, etc
3. Data: What data are you using, where did you get it, what is your gold standard, how is your data divided between development and test?
4. Results: What is your baseline? What did your metrics reveal?
5. Discussion: What can be done to improve the results? Which parts were particularly challenging?
6. If you are implementing something that does not have accuracy results (e.g. a text adventure), incorporate a longer discussion of the details of your implementation, maybe a run through simulation (e.g. with screenshots) of the core points
7. If you are submitting files as an archive to Canvas, please:
    - Submit the write up separately, as .pdf.
    - Use zip or gzip, not rar or other more exotic archiving systems.
    - you can also share your project with me as a github repo, if you are familiar with those and prefer it