2024 Industry Documents Undergraduate Summer Fellowship - JUUL Labs Collection Final Blog Post

By Gordon Lichtstein

Table of Contents

Introduction

The 2024 Industry Documents Undergraduate Summer Fellowship is an 8-week virtual summer fellowship aimed at doing research into the JUUL Labs Collection, a set of documents related to the e-cigarette brand from lawsuits against JUUL, the FDA's Center from Tobacco products, and Schlesinger Law. The UCSF Industry Documents Library (IDL) is a repository for documents like those from JUUL to make them available to the public after processing (although not all documents in the IDL are public)

Throughout this fellowship, I designed and executed multiple projects utilizing natural language processing (NLP) techniques to investigate and learn from the JUUL Labs Collection and the IDL as a whole. I participated in weekly meetings with the IDL and Amazon Web Services (AWS) workshops.

For some brief background about me, I'm an incoming freshman at MIT planning to double major in computer science and linguistics. I'm passionate about their intersection in the field of NLP, and the application of NLP for the betterment of humanity, such as in environmental sustainability or the digital humanities.

Thank you so much to UCSF, my co-workers, and mentors at the IDL whose support made this fellowship not only possible but incredible. I learned so much, and I can't thank them enough.

Project 1: OCR Accuracy Without Ground Truth Data

OCR Context

Optical Character Recognition, or OCR, is a heavily utilized technology with broad applications across many industries. In essence, OCR is just converting pictures of printed text or handwriting into digital text - something along the lines of a .txt file or Word document. While at face value this seems simple (probably because it's very easy for us humans to do), in fact it is not. Think of all the situations you see text in on a daily basis - vastly different fonts, lighting conditions, rotations of the text, formats of the text, layout of the page (e.g. newspapers), and handwriting styles.

There exist a vast array of open-source and commercial OCR technologies with a wide range of capabilities. However, there exists a general dichotomy in the solutions available: highly accurate yet expensive or slow OCR vs comparatively inaccurate yet cheap or very fast OCR. Proprietary solutions such as AWS Textract fall in the expensive/slow/accurate category, while open-source solutions like Tesseract fall more in the cheap/fast/inaccurate category.

UCSF IDL Context

In the UCSF Industry Documents Library, the problem is even more complex. Text might need to be redacted, we have to deal with many file formats (e.g. pdf, tiff, Excel sheets, images, etc…), and with varying qualities of text. Some digital scans of physical documents might be very poor, grainy, have various ink stamps on them, or have any other number of issues.

Specifically relevant to the JUUL Labs and RJ Reynolds collections I focused on in my fellowship, there was a handwriting campaign that led to a huge chunk of inconsistently written and formatted letters.

The IDL has a massive database of documents to support researchers studying digital humanities around the world. In order to easily find the specific documents they are looking for, they can search the IDL website for keywords (e.g. "JUUL", "disclosure", or "youth"), which will go through our previously-generated OCR. This OCR was generated using older OCR technology. Much of this OCR text is fine, with most keywords from the original documents being contained in the OCR text. However, some of the original OCR text doesn't accurately transcribe the content of the original document, leading it to lack keywords, particularly on handwritten text. This is essential data to have, as scanned letters from a letter writing campaign compose an important portion of the dataset. If a researcher were to search for a keyword, these documents likely wouldn't appear. It would be almost like they are invisible, inaccessible by many searches.

This could also introduce biases into research, as documents with poor quality OCR would be less likely to be included in search results. These might be handwritten documents like letters, which could contain very valuable information for researchers, a topic my fellow summer fellow Theo Zhang investigated.

Modern software like Textract can get acceptable OCR for even the most illegible handwriting, but it would be unfeasibly expensive and time-consuming to re-run all documents through an algorithm like that. Most of the original OCR is fine too, so running this through Textract wouldn't yeild as much benefit. We need some way of using the already-generated OCR, and specifically applying the more expensive algorithm in the places it is most needed, like in handwritten letters.

My Solution

In thinking about this problem, I believe that the best solution is to develop or identify a metric that can distinguish between poor quality OCR (such as that from handwriting) and acceptable OCR, perhaps with a few typos or mid-level errors (such as from scanned-in emails). This is not a trivial problem, as the poor quality OCR can have some of the trappings of reasonable English like the length of "words", and the acceptable OCR can have some of the indicators of meaningless text (e.g. erroneous numbers, many typos, etc…).

There already exist standard metrics for testing the accuracy of OCR text such as word error rate and character error rate, but these rely on having "ground truth" text - transcriptions of the documents generated by humans that are known to be correct. We need an algorithm that can do the same, except without any ground truth data.

In this project, I identified and tested numerous algorithms on their ability to distinguish between these two types of OCR text. Many of these algorithms come from completely different areas of computer science and even mathematics, and represent many hours of research identifying and testing them. I also explore where the best of these algorithms fail, unique methods to combine them, and optimizations.

Autocorrect-based Algorithms

The first general category of algorithms for detecting poor quality OCR was autocorrect algorithms. The main idea is to take OCR text, run it through an autocorrect algorithm, then look to see how much autocorrect changed the text. If autocorrect changed the text a whole lot, then the original text probably was of poor quality. On the other hand, if autocorrect doesn't do much (doesn't change many words), the text is probably high quality to begin with.

In practice, there are a few more steps. When I take in a file with OCR text, I first split it into lines. Then, I apply a normalizer algorithm (whisper's EnglishTextNormalizer). This reduces text to lower case and performs other normalization steps to make the autocorrect effects more consistent. Many files contain numerous empty lines, and some autocorrect algorithms also do not work on them. So to mitigate this, I map ever totally empty line to an arbitrary string that never appears in text and will not be changed by autocorrect (in my case I used "1111111111111111"). Thus, new lines will always equal new lines in the normalized and the normalized + autocorrected text, without causing errors if the autocorrect algorithm doesn't accept empty new lines. Then the process diverges. These normalized lines are saved, and another version is fed through the autocorrect algorithm. Finally, I compare each normalized line to its corresponding normalized + autocorrected version using a word error rate (WER) algorithm (I use jiwer.wer) and place the WER for each line in a new array. WER measures the proportion of words containing errors to the total numer of words. A high WER means the text is low quality, and a low WER means the text is high quality. I then average the WERs and return this value as the final error for this file and this particular autocorrect algorithm.

In this category, I tested two Python libraries: autocorrect and jamspell.

Autocorrect is, a multilingual spelling corrector based on the older autocorrect library and Peter Norvig's spelling corrector. This algorithm uses the Levenshtein distance (also called the edit distance), which calculates the number of edits it takes to change one word to another word, to determine the word from a corpus closest to misspelled words, and correcting the misspelled word to that word. When a misspelled word could potentially be converted to multiple known words with the same edit distance, it chooses the most common target word in the corpus. The library autocorrect also has a few other features which I tested. It has a fast mode, that only detects single typos (e.g. edit distance of 1), and also an OCR-specific mode that only uses edits based on replacements, as these are particularly common for OCR text (e.g. amount is OCR-ed as omount). I tested every combination of these features (standard autocorrect, OCR-mode, fast mode, and OCR + fast mode). These could be helpful, as speed is of the essence for our project. We want our algorithm to save time and money, and as such it has to be able to process thousands of documents very quickly. Anything that reduces computation time a tiny bit will correspond to a huge improvement when run on thousands of documents.

Jamspell is another autocorrection library focusing on accuracy, speed, and cross-platform availability. It is a more modern and advanced library, utilizing an N-gram Markov model to achieve context-specific accuracy, and implements a number of optimizations (such as a Bloom filter) over its predecessor SymSpell. I expected this algorithm to perform better than the more basic autocorrect library.

Gibberish Detector Algorithms

The second major category of algorithms I used was gibberish detectors. Gibberish can actually be a significant problem for those who host and manage websites and web services, as malicious users can automatically submit random queries to occupy the server and slow down other users. In some cases, each user query costs the website a tiny amount of money, and random queries could become extremely expensive. In a similar scenario, malicious users of forum websites like Reddit could automatically submit many junk posts. Thus, numerous tools have already been developed to quickly detect gibberish text quickly and very efficiently, as a computationally expensive gibberish detector could impose more strain on servers than a junk query. For these, I tested three tools: gibberish-detector, autonlp_gibberish_detector, nostril, and random_string_detector. For each, I run the algorithm through a similar process as described above, where I split the text into lines, pass each line to the gibberish detector, and average the scores into a final gibberish score. For detectors that return one True/False value, I return the proportion of gibberish lines. While this is not a perfect way of generating metrics, it is more than sufficient and is very efficient.

Gibberish-Detector is a Python program forked from rrenaud's algorithm of the same name, modified to support Python 3 and using a 3-character Markov chain model instead of rrenaud's 2-character model (Random-String-Detector also uses a 2-gram model. This model returns a True/False value. Markov models are the precursors to modern neural networks, in this scenario they work by training on a corpus of correct English text (such as Wikipedia), and record how often each sequence of n characters (n = 3 in the case of amitness' Gibberish-Detector). Then, when the model sees new text, it calculates the probability of seeing each sequence of n characters in the text and multiplies them all together. If this probability per character is below a cutoff value (which can be optimized, as I will discuss later), then the text is marked as gibberish. A lot probability per character score would align with strange text that was not commonly seen in the original corpus of example text. As an implementation note, Gibberish-Detector is not a Python library or module, and is two Python programs and a compressed file of frequencies. This made it slightly harder to run from Google Colab. To make this easier I downloaded the relevant files into my Google Drive which I then connected to Colab, to allow for file persistence. Then, I imported the Python file's methods from inside my main Google Colab notebook.

Autonlp-Gibberish-Detector uses a neural classifier model to identify text as either gibberish or reasonable English. Like the Markov models some of the previously described algorithms use, neural networks have to be trained on a corpus. In classifiers like Autonlp-Gibberish-Detector, the corpus must be labeled, meaning that the corpus consists of some gibberish text that we know is gibberish text, and some text that is known to be correct. During training the classifier makes an educated guess as to which category text in the corpus belongs to, and then we correct it's guess and modify the model to be more likely to generate the correct answer next time. Neural models like Autonlp-Gibberish-Detector return a confidence score as to whether the text is gibberish or not, and I use the average of this score directly.

Nostril, or the Nonsense String Evaluator, is a Python module that provides a convenient function that returns a True/False value as to whether the the input text (a short string) is gibberish text or not. This model was actually specifically trained on source code to identify meaningful identifiers like variable and function names, but it actually works quite well on standard English text. Nostril works by using heuristics to detect simple True/False cases, and an TF-IDF scoring system based on 4-grams for more complex cases, described in the paper "Nostril: A nonsense string evaluator written in Python". This is similar to the Markov models described above, but with an added caveat that TD-IDF adjusts the frequency lists based on both the frequency of n-grams within one document, but also across all documents in the training set. In the implementation of this algorithm, I came across the problem that Nostril will refuse to evaluate very short strings (less than 6 characters). This is a problem for lines that contain fewer than 6 characters, as Nostril will throw an error. To account for this, I simply caught any errors that Nostril produced and marked that line as nonsense. While again this is a very rudimentary metric, it is quick

Random-string detector is a Python library that uses a 2-gram (bigram) model, like Gibberish-Detector but with less context information. I expected that this library would perform worse than Gibberish-Detector as it used a 2-gram model compared to a 3-gram model, but that it might also be faster. This library returns a True/False value.

Mathematical Algorithms

This general category encompasses mathematical methods for detecting gibberish text. Here I analyzed two metrics: the index of coincidence and the character entropy. These methods are generally quick, and I wanted to test their accuracy compared to the specifically designed gibberish detection algorithms.

The index of coincidence (IC) is a metric originally from cryptography to detect correct decryption of ciphertext. The IC of a string is also not affected by substitution ciphers (where each unique character of plaintext is mapped to unique ciphertext character). It measures the probability of pulling two random characters from a text and getting the same one. The probability of pulling the first letter is n / L, where n is the count of how many times the target character appears in our text, and L is the total number of characters in the text. The probability of pulling the second letter is (n-1) / (L-1), as by pulling the first letter out, we have reduced both the number of our target characters, and the total number of characters in the text. To calculate the IC of a specific character, we multiply these two values together ((n/L) * ((n-1)/(L-1)). To calculate the IC of a whole text, we calculate the average IC of all characters contained in the text. The idea of the IC is that for random or gibberish text, each character is about evenly distributed as as such the IC for that text will be close to 1. For text in English or any other language, characters are not evenly distributed so the IC will be somewhat greater than 1, somewhere around 1.7. Notably, my algorithm normalizes text to be lowercase and counts all characters present in the text except spaces and new lines (even numbers, punctuation, and symbols). This differs a bit from its original usage where ciphertext and plaintext would usually only contain letters. As such the IC values for my text can sometimes be over 2, but gibberish text still has an IC value of about 1. This is a relatively simple algorithm to implement with a dictionary in Python, which is an efficient data structure that allows this algorithm to run very quickly.

Character entropy as I am using the term here refers to the entropy of the distribution of characters in a document. I am using Shannon's definition of entropy of the distribution of a quantity (such as the frequency of letters) to be the average of (-1 * p * log(p)), where f is the proportion of text taken up by each character. A low entropy means the text is very predictable, while a high entropy means the text is difficult to predict. I've seen this metric used in an attempt to determine whether or not the Voynich Manuscript represents meaningful language or is a hoax (apparently the language has a significantly lower character entropy than languages we know to be real). As a note, in my implementation of this, I do not count spaces.

LLM-based Algorithms

The final general category I investigated (though not extensively) was LLM-based gibberish identification. LLMs, at a high level, work by predicting a probability distribution for the next token (part of a word) based on previous tokens. LLMs like ChatGPT take your prompt and continuously select the next token according to this probability distribution. I take advantage of this for identifying nonsense text by tokenizing (splitting the text into tokens) each line and measuring the perplexity of GPT-2 on that line (any language model would do, but I wanted something relatively lightweight). Perplexity is e to the power of the loss per token of input text of the model. Loss is how “incorrect” or how “surprised” the model is by the next token. I averaged the perplexity of each line of the file. High perplexity means the text is very “confusing” - that it doesn’t look like anything it has seen before - that it is gibberish. In the implementation of this algorithm, I initially got exemplary results: extremely quick and very accurate. This made me a bit suspicious as LLMs are generally somewhat slow, plus I was running GPT-2 on Google Colab without huge computational resources so it would likely be even slower. Upon investigation, I realized that I had been passing in the file name instead of the file text to GPT-2. Somehow, the file name is highly correlated with whether the document is good OCR or bad OCR. After fixing this error, the perplexity-based metric was extremely slow and gave frequent errors. LLMs have maximum context sizes - the maximum number of tokens they can process in one run - and I was exceeding it for some longer lines. This gave frequent errors that I caught and recorded. The metric was the slowest by far (taking over 4 hours to run on 160 documents) and it wasn’t even very accurate. As a result of this poor initial performance, extreme length of tests which limited me from running other code on Google Colab, and the outstanding performance of other metrics, I decided not to pursue LLM-based metrics any further.

Experimental Metrics - Testing Metrics

Each of the algorithms above generated a somewhat continuous metric. The ones that inherently returned True/False were modified as described above to produce a proportion of gibberish between 0 and 1. Some algorithms similarly generated scores between 0 and 1, while others generated numbers that were smaller or larger.

The data I had available were the JUUL hot 60 documents (mostly PDFs of emails, word documents, Excel sheets, and other things that have accurate OCR), and the RJ Reynolds 100 handwritten documents, which mostly contained handwritten letters which have very poor OCR transcriptions. Every document has an OCR file associated with it, run previously with older technology. I took advantage of this pre-labeled data by assuming that the JUUL documents were good OCR, and the RJR documents were bad OCR, but it’s important to note that this assumption isn’t perfect. Although the JUUL documents may contain small amounts of handwritten or mixed text and the RJR documents may contain some printed text, the two sets were sufficiently different to be valuable in measuring transcription accuracy.

For each metric, I ran every bad OCR file and every good OCR file through it, and recorded this result in dictionaries based on whether the origin file (the key is the name of the file, the value is the metric score). Errors in processing (such as GPT-2 running out of tokens) were recorded as value = -1. Then, I normalized those scores to be between 0 and 1 by dividing each score by the maximum score I calculated for that metric. It is possible that running this algorithm on future text would generate scores greater than 1, but this doesn’t affect anything due to the cutoffs which I will describe shortly. Errors were converted to either 0 or 1, and for every metric I tested which value was better for errors.

Now, for every metric I have two dictionaries: one containing the scores when ran on the bad OCR documents, and one when run on the good OCR documents. I could compare the average metric scores between the two categories, but this isn't actually very useful in the real world. We really want a decision function that when a metric is passed as input, it outputs True if the text is good OCR, and False if the text is bad OCR. This function and its optimization is described below. For now, I will assume that we have such a function.

Then, I iterate through every file in the dictionary and run this decision function on the metric score. I then take the proportion of files that it correctly sorts and return this as the final accuracy of that metric.

Decision Functions

The individual metrics generate one-dimensional outputs (just one number between 0 and 1). Thus, the best decision function will be a hard cutoff. Any value below the cutoff will be labeled good OCR, and any value above the cutoff will be labeled bad OCR. Whether good OCR is above or below the cutoff is actually arbitrary and will not affect the final accuracy of the metric, but it will invert whether high scores = good OCR or high scores = bad OCR.

OK so now we know the general format of the decision function: a hard cutoff. But how can we optimize this cutoff? One solution is to iterate through every possible cutoff between 0 and 1, and test how many the model got correct. However, this would require testing an infinite number of points (infeasible). Another idea would be to test cutoffs all spaced 0.001 (or any other arbitrarily small number) apart. Yet again, this would be inefficient and slow. The solution I employed is to test every value each metric generated as a potential cutoff point. This would still generate the optimal cutoff for our specific dataset, but the best cutoff might be slightly different in the real world.

This cutoff point could then be tweaked up or down manually to select for more false positives or false negatives, depending on the needs of the user.

Testing Environment

All tests were run on the free tier of Google Colab. This did pose occasional problems for the slower metrics, as Colab will automatically disconnect you after a certain period of inactivity (even while code is running). However, this was relatively minor. I was on a Python 3 CPU runtime with 12.7 GB of RAM and 107.7 GB of disk space. It's likely that these computation times could be sped up by multithreading, but I did not experiment with this.

Results

Some decision functions generate higher scores for good OCR documents, while others generate higher scores for bad OCR documents. For example, my GPT-2 based metric would score the worst possible OCR text as a 1 and the most perfect OCR as a 0, while Nostril would score perfect OCR as a 1 and absolutely horrible OCR as a 0. I have separated the two below. If a number is in the “Accuracy, printed < handwritten” column, then the metric is higher for handwritten (poor quality) text, whereas if it is in the “Accuracy, printed > handwriting” column, then the metric is greater for printed (good quality) text.

The time in seconds taken for each algorithm to run through the 160 documents, the average time in seconds to run on one document, and the time in days to run through about 19.971 million documents is shown to the far right of the table. The time taken to run through about 20 million documents was included, as this is the total number of documents in the UCSF IDL (it may have changed by the time you are reading this, so I would recommend checking the UCSF IDL stats here).

		Accuracy			
	0 or 1 better for errors	printed, handwriting	handwriting, printed	time (sec) per doc	days / 19,971,203
Autocorrect	1	0.79375		22.41744735	5181.752219
Autocorrect (ocr)	1	0.73125		6.439200085	1488.409399
Autocorrect (fast)	1	0.7125		1.055654182	244.0125458
Autocorrect (ocr fast)	1	0.775		0.975997594	225.6000703
Jamspell	1	0.875		1.983058323	458.3803279
Gibberish-Detector	1	0.89375		0.03305484504	7.640567365
Autonlp Gibberish Detector	0		0.84375	48.40951883	11189.77231
Nostril	1	0.975		0.0527151227	12.18500482
Random-String-Detector	1	0.85625		0.01944239736	4.494074819
Character Entropy	0		0.79375	0.01866874546	4.315246589
Index of Coincedence	1	0.88125		0.0198105216	4.57916607
GPT-2	0		0.84667	91.62318412	21178.53252

Conclusion & Discussion

Overall, the most accurate metric was Nostril by far, achieving an accuracy of 97.5%. This means that out of the 160 documents, it only for 4 incorrect. Other metrics such as Jamspell, index of coincidence, and Gibberish-Detector also did well, with accuracies significantly over 80%. Nostril was reasonably fast and very accurate, but the index of coincidence struck a very good balance between speed and accuracy. It is extremely important to note that this doesn't exactly mean that we can distinguish between handwriting and printed text 97.5% of the time, only that we can distinguish between these two collections with 97.5% accuracy, which may or may not translate to larger or different samples. For example, we compared mostly printed text to pue handwriting, and this sort of pure handwriting is a small percentage of the overall data.

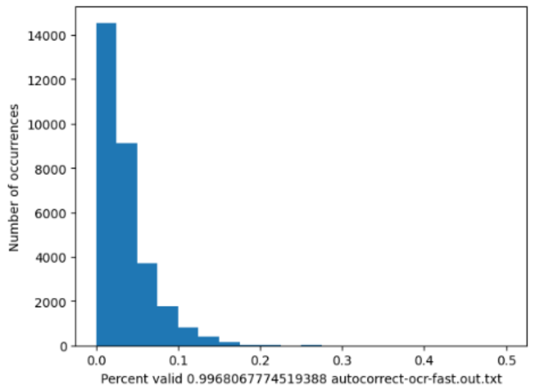
Many documents are some mix of handwriting and printed text, and we would likely detect these as “good OCR”. In fact though, the handwritten information may be the most useful part of the document, such as in printed forms where all the interesting information is handwritten in. While lowering the cutoff may be able to detect some of these documents, future investigations on mixed handwriting & printed documents are necessary to confirm. The results of one such investigation are described below.

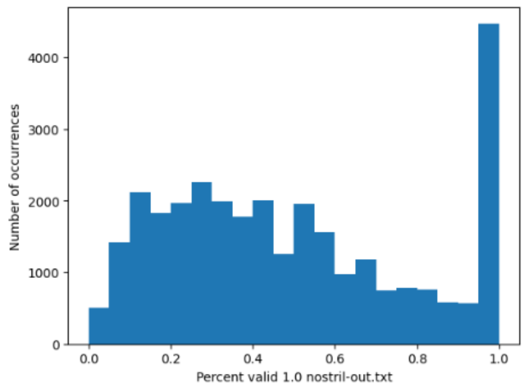
It is interesting to observe that 1 is a better value for errors when printed < handwriting, while 0 is better for errors when the opposite is true. This makes intuitive sense, as an error likely indicates poor quality text, and 1 represents the worst possible quality text when the metric is higher for handwritten (poor quality) text compared to printed (high quality) text.

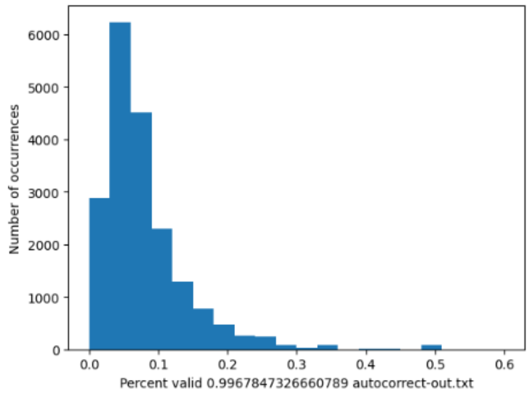
Evaluation on Random IDL Data

To test the generalizability of these metrics, I ran 3 promising metrics on a randomly selected set of about 30,000 documents from the Industry Documents Library I scraped from the public library. The reason I didn’t run any more due to time constraints. I wanted to see how well these metrics would be able to distinguish between high and low-quality OCR. However, there aren’t labeled datasets like there were for the JUUL data. So, in order to evaluate these metrics, we can look at the distribution of scores. If a metric were very good at separating high and low-quality OCR, we would hope to see a bimodal distribution, making separating the two sets easy. Additionally, we’d like a metric that can generate a number as often as possible, and not fail often. Thus, I also include the percent of the time the accuracy generated a value.

Looking at the graphs below, it is clear that Nostril has the most bimodal distribution. The metric score is on the x-axis, and the count of how many documents fell into that bucket is on the y-axis. This is even further evidence that this is the most robust and balanced metric.







Category Comparison and the Time/Accuracy Trade-off

Some categories performed remarkably better than others. Autocorrect-based algorithms were the worst of the categories, often failing to reach 80% accuracy. I believe this is because these algorithms essentially “gave up” correcting extremely bad OCR, which is what most of the RJR handwritten documents are. The autocorrect can’t correct single letters or totally meaningless symbols, so oftentimes the autocorrect algorithms didn’t change much of the bad OCR text. These algorithms were also extremely slow, and the more accurate algorithm Jamspell was slower then the second and third best performers in this category.

Gibberish detection algorithms performed the best, with the two best performers being in this category. These algorithms were much faster than the autocorrect-based metrics, but within this category we yet again see a tradeoff between time taken and accuracy. While Nostril was the most accurate algorithm, it was not the quickest in the category (this title goes to Random-String-Detector, which was almost 3 times quicker than Nostril).

The mathematical category performed surprisingly well, better than the autocorrect-based algorithms but marginally worse than the gibberish detection algorithms. The third best algorithm (index of coincedence) is in this category. What they lose in accuracy, they make up for in speed. These algorithms take only one pass through the text to count the occurrences each character, then perform one calculation per character.

The LLM category performed acceptably well, but took an extremely long time, eliminating this group as a candidate for large-scale gibberish detection.

While there is no one best algorithm for every use case, it is my opinion that the index of coincidence is the best compromise between speed and accuracy, especially when applied to large datasets. For smaller datasets, one could afford to spend a bit more time running the more accurate Nostril algorithm.

Error Analysis

I am primarily going to investigate the errors that Nostril makes. I’ve listed the documents Nostril gets wrong alongside some commentary.

The two other most promising algorithms get many of the same documents as Nostril incorrect, although their errors don’t necessarily overlap with each other as much. One of the four documents that Nostril got “incorrect” was arguably a mistake in the dataset, as it mostly contains printed text. Additionally, some of the incorrectly labeled files are Excel files, hinting at a deeper error with how the original OCR algorithm handled these files.

It is interesting to note that these algorithms are sometimes getting the scan of an email or two incorrect. these scans seem generally clean, and this is worth some further investigation.

Printed documents Nostril incorrectly identifies as bad OCR

mjxw0291 - This is a Excel file (not a PDF or other easily OCR-able format)
sgxd0283 - I think this is happening because there are a lot of lines with only email addresses, strange numbers, and names. There is also a bit of clutter
Handwritten documents Nostril incorrectly identifies as good OCR
Lghl0187
fqdx0231 - This is a printed PDF (incorrectly labeled as handwritten)

Multiple Metrics

One potential idea is to create a combination of metrics that could potentially score higher than any one metric. We could run multiple metrics on a text, and take a weighted average over these scores. This would only be beneficial if the documents that our best metric identifies incorrectly are documents that other metrics identify correctly. However, in analyzing the incorrect results of Nostril, Random-String-Detector, and other top performers, the documents that Nostril were identifying incorrectly were also identified incorrectly by the other algorithms.

Another limitation is in the total time required to run these metrics. Our goal is to save time and money, and running 10 computationally expensive metrics wouldn't do either of those things. The final blow to combinations of metrics is done when considering the potential benefit: 2.5%. The absolute worst an optimized weighted combination of metrics could do is 97.5% correct, and the absolute best it could do is 100% correct. Keep in mind that 2 of the 4 documents that contribute to that remaining 2.5% were actually correctly categorized when we looked back at them. So at best, it's possible we might be able to improve our performance by 2.5% maybe, at the cost of spending 2 times as much computational time. I actually tested this with 2 metric combinations, and I could not generate an increase in accuracy at all.

Learning and Takeaways

I unfortunately could not include one or two algorithms, as I couldn't get them running in Google Colab. Many of my initial hypotheses and ideas (e.g. using autocorrect algorithms) were ultimately not correct. There were also barriers around downloading and accessing documents programmatically, which inadvertently resulted in the discovery of a minor bug. I would also like to test these methods on documents that contain mixed handwriting and printed text, as these are the majority of documents with handwriting in the UCSF IDL database.

I think this project was valuable, as it taught me a lot about the research process, utilizing Google Colab, and about designing and executing independent projects generally. Being almost completely independent, I was able to decide the point to stop investigating. Actually similar to the cutoff function I used, I needed some way of determining at what point my current research was "good enough". If I didn't have a cutoff, a project like this could go on for weeks, with only marginal improvements being made. I quite enjoyed the freedom, and I'm proud of the work I did.

Project 2: Embedding Search and Visualization

Introduction

Currently, the UCSF Industry Documents Library uses Apache Solr's MoreLikeThis (MLT) feature to allow researchers or anybody browsing the web to discover more documents similar to the one they are currently viewing. This is important for those exploring the dataset or looking for specific information to navigate more easily. Anybody can click the "More Like This" button in the top right corner when viewing a document, and get a list of about 10 similar documents. However, sometimes this doesn't work how we want. Apache Solr is currently configured to produce this list based on a weighted sum of fields like the industry, author, and title similarity. This doesn't take into consideration document contents or themes of documents, and while often the titles of documents are correlated with their contents, a lot of the time they are not. There are cases where the current MLT search returns no results as there aren't any documents with similar authors and names, and poor quality MLT results are much more common.

While there might be many possible solutions to this problem, I believe the most elegant is to generate document embeddings, a way of taking a document and converting it to a vector that (hopefully) represents all the key information of that document. Other algorithms like TF-IDF compare word frequencies and counts, but an embedding-based search could go much deeper.

Document embeddings capture semantic content by encoding the context and meaning of words in a dense vector space. This method uses natural language processing (NLP) to (hopefully) provide more nuanced and relevant search results. Unlike TF-IDF or similar algorithms like MLT, which rely on the frequency of terms and often miss context, embeddings allow for more sophisticated comparisons that consider the semantic and thematic elements of documents, improving the accuracy and relevance of search results.

We can also generate interesting visualizations, necessitating dimensionality reductions. As the embeddings are just vectors (lists of numbers), we could imagine each one as representing a point in a high-dimensional space. Documents close to eachother in this space would also hopefully be close to eachother in meaning or theme. However, visualizing high-dimensional data requires reducing it to 2 or 3 dimensions, which makes it comprehensible to our 2 and 3-D adjusted minds. Dimensionality reduction techniques such as PCA, t-SNE, and UMAP can transform embeddings into lower-dimensional representations, enabling the creation of interactive and intuitive visualizations which help users explore the dataset more effectively, revealing clusters and patterns that might not be apparent through text alone. Additionally, the visualizations give the user an ability to grasp entire collections or databases at once, which can often consist of thousands or millions of documents in the case of UCSF.

Embedding Search

To create an embedding search algorithm, we need two main parts: an embedding algorithm and a similarity metric. There are also nearest neighbor algorithms that are necessary to the search, but Apache Solr provides its own kNN (k-nearest neighbor) algorithm that gets approximate closest documents based on the embeddings and distance function quickly.

The first algorithm I tested was Doc2Vec, an extension of the Word2Vec model designed to generate vector representations of entire documents, rather than single words. It creates these embeddings by training on a corpus of provided documents and learning to predict words based on both their surrounding words and a unique document identifier. This lets Doc2Vec to produce vectors that encode the document content and structure, making it useful for various natural language processing tasks such as document classification, clustering, visualization, and embedding search. Doc2Vec generates embeddings that are tailored to the specific corpus it is trained on, ensuring that the vectors represent the unique characteristics of the documents. We can also choose the exact size of the vector (a variable I will illustrate in the visualization section), resulting in quick and efficient similarity searches and training for documents within the training dataset. It also means that we don't have to send data to a third party or run extremely resource-intensive machine learning models, so we can save a lot of money. However, its performance might degrade when applied to out-of-sample data, as the model might not generalize well to documents with different styles or content not seen during training. For UCSF specifically, this means that an embedding model trained on only JUUL documents wouldn't embed documents from the opioids industry very well. Additionally, in my testing it seemed that running Doc2Vec on a random sample of the entire IDL database, a lot of contextual information was lost. This is shown in the visualizations, where Doc2Vec doesn't group similar documents well. The solution to these problems is likely training and running different Doc2Vec models for each industry or even for each collection. Then when we execute a similarity search, we first filter for only documents in the same industry, then sort by closeness to the original document according to our distance function. This would mean giving up seamless visualizations of the entire database at once, but would be significantly faster and more accurate than training and running one Doc2Vec model on the entire database.

The second major category of embedding algorithm I tested were TogetherAI embedding models, pretrained models used by large language models (LLMs) to generate vector representations of text, so that text can be understood and utilized by LLMs. They perform well across diverse and variable text inputs due to their pretraining on extensive and varied datasets (mostly text data from the internet). They can likely capture broader contextual relationships and nuance, making them suitable for a wider range of documents. However, they're often resource-intensive and slow, requiring significant computational power for processing and dimensionality reduction. Usually they would be used via a third party, sending data to them and receiving embeddings, but this incurs a fee and may restrict our ability to work with sensitive or unredacted data. Additionally, there are a few other key practical barriers to using TogetherAI models. For one, they have maximum context sizes in the range of 512 tokens to 32768 tokens. However, tokens don't correlate very well with words or number of pages, and the number of tokens a document uses up can be quite variable depending on the text it is used on. For example, low quality OCR looks like gibberish and would likely take up more tokens per page than perfectly coherent OCR. As it is hard to predict the number of tokens a document will take up, I essentially used a trial and error approach, where I first tried to feed in the entire document, and if that errored because of too many input tokens, I cut the document down by 2/3 and retried it recursively. Thus, eventually the document would fit in the context size and it would return an embedding. This is not elegant at all and there are far better ways of embedding large documents (such as taking many embeddings and averaging them), but these are all complicated when submitting bulk API requests to TogetherAI. Another practical difficulty with the TogetherAI embedding models are their dimensionality. Embedding models often produce vectors with a dimensionality of 768 or more, which would take up too much storage space and would be too slow to search through. Thus, dimensionality reduction algorithms must be applied to reduce the number of dimensions, but this warps the data. I tested all 8 of TogetherAI's embedding models, using the $5 of free credits anybody can get by signing up for an account. It is difficult to compare their performance as we don't actually know what documents are "closer" to others, because this is a subjective judgment. The best way to get a sense of performance is to look at the visualizations, but even these are poor metrics for how the embeddings actually look and act in the high-dimensional space. However, I am interesting in developing a small dataset to test this based on human understanding of document similarity.

Similarity Metrics:

Apache Solr includes a dense vector search, so these embeddings can be plugged right into our current database. It's important that this dense vector kNN search is approximate, as this balances speed of retrieval with accuracy of results. There are multiple similarity metrics:

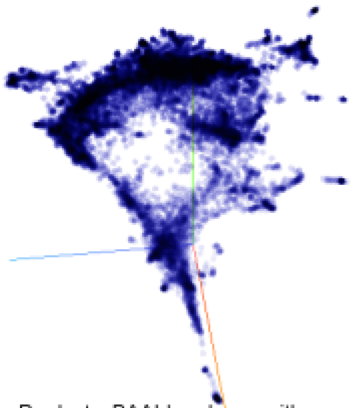
Euclidean Distance measures the straight-line distance between two points in a multi-dimensional space, similar to the Pythagorean theorem (it is actually the Pythagorean theorem, but with more numbers squared under the square root). It is useful when the magnitude of the vectors is important, but it is the slowest of the algorithms.

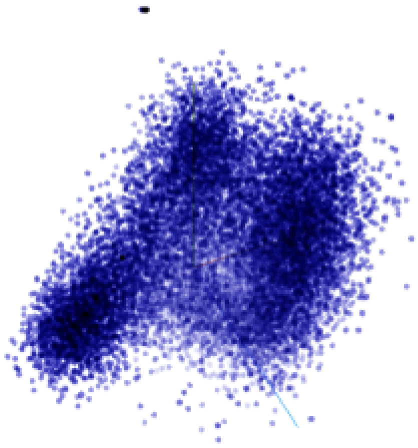
Cosine Similarity calculates the cosine of the angle between two vectors, focusing on their direction rather than their magnitude. This metric is useful when the orientation of the vectors is more relevant than their length, such as in text similarity tasks where document size varies. Something interesting that I observed when testing is that in extremely high dimensional data (e.g. the 768 dimensions generated by TogetherAI embedding models), the 10 closest documents were the exact same when measuring distance using cosine and Euclidean distance. I believe this is related to the curse of dimensionality, as in high dimensions distances become more uniform. Overall, I believe this is the strongest of all the algorithms, and it is generally standard for these types of applications.

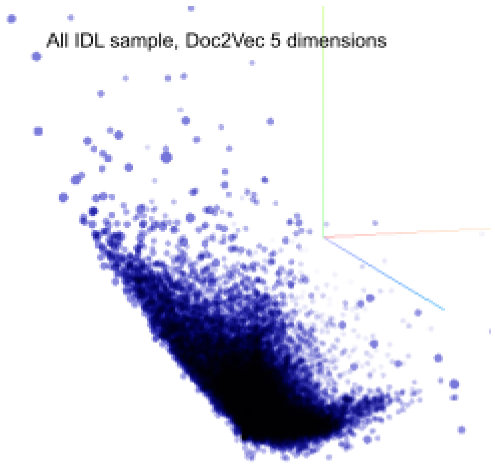
Dot Product (Inner Product) measures the length of the projection of one vector onto another, combining aspects of both magnitude and direction. It's computationally efficient and often used in scenarios where both the size and alignment of vectors matter, or when speed is necessary.

Dimensionality Reductions

As previously mentioned, to efficiently utilize the embeddings from the TogetherAI models, we must reduce their number of dimensions from ~768 to somewhere between 10 and 100. I often defaulted to using 20 dimensions, but this was somewhat arbitrarily chosen based on what seemed to be the standard in previous work. Another application of dimensionality reduction algorithms is in visualization of data. Instead of converting to somewhere around 20 dimensions, we can convert the data down to 2 or 3 dimensions, which we can visualize with standard tools. This can also be applied to the Doc2Vec embeddings to visualize them as well. There are a few key embedding algorithms I tested, which I'll describe below.


Tobacco Products, BAAI-bge-base, with dimensionality reduced from 768 to 20 ahead of time


Tobacco products, BAAI-bge-base raw data

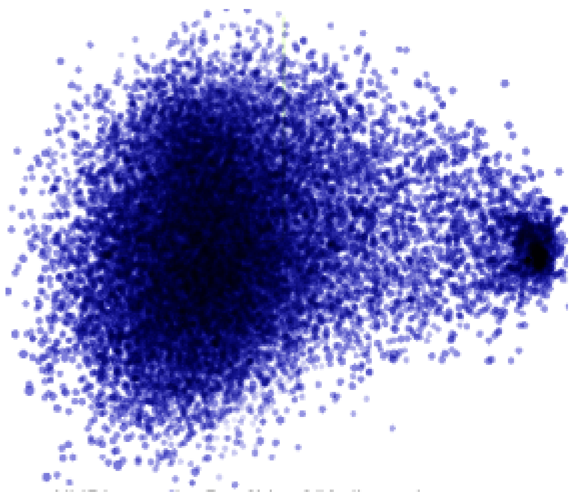

All IDL sample, Doc2Vec 5 dimensions

Principal Component Analysis (PCA) transforms the data into a set of orthogonal components that capture the maximum variance, reducing dimensionality while preserving as much variance as possible using eigenvalue decomposition. It's computationally efficient, very fast, and easy to apply to new data, making it a practical choice for large datasets or initial discovery work. However, PCA's linear nature means it can struggle with capturing complex, non-linear relationships like those present in document embeddings. Additionally, PCA is sensitive to outliers which can skew results significantly.

t-distributed Stochastic Neighbor Embedding (t-SNE) works by converting high-dimensional Euclidean distances into conditional probabilities that represent similarities, and then optimizing the low-dimensional representation to reflect these probabilities. It's very good at preserving local structures in the data and can understand non-linearity, making it useful for visualizing clusters, but it has a high computational cost, scaling quadratically in both time and space with the number of data points, making it unsuitable for very large datasets.

The stochastic nature of t-SNE also means that different runs can produce different results, and it commonly has a number of hyperparameters to tune.

Uniform Manifold Approximation and Projection (UMAP) fundamentally works similarly to t-SNE, computing hihg-dimensional similarities and tuning a low-dimensional graph to to match it as much as possible, but it also uses sophisticated techniques from algebraic topology and cross-entropy similarity functions to speed up generation. Like t-SNE, it works well for non-linear data and is stochastic, but it is considered better at preserving the overall shape of the data and is much faster than t-SNE. However, it's effectiveness depends significantly on the tuning of its hyperparameters, and it's much more sensitive to small changes in them than t-SNE. Thus, in exchange for running each reduction faster, we must run a greater quantity of reductions testing different hyperparameters.


All IDL sample, Doc2Vec 250 dimensions

The final option I considered were using autoencoders, which are neural network models designed to learn efficient codings of input data. They work by compressing the data into a lower-dimensional representation and then reconstructing it back to the original input (training to match the identity function). This process can capture complex, non-linear relationships within the data and contextual understandings. While autoencoders may provide the better dimensionality reduction, they are computationally expensive to train and run, especially on large amounts of data.

Due to the difficulty of pursuing this option, I decided to leave it to future experiments. Dimensionality reduction isn't the core of this project, and wouldn't be relevant if we used Doc2Vec (the overall best algorithm). Because of its speed and the size of my datasets, I largely stuck to using PCA for initial visual investigation, and UMAP for more in-depth searching.

Conclusion

The initial implementation has demonstrated promising results, showing that embedding-based search can provide relevant document recommendations. The performance of the embedding models, particularly Doc2Vec, was quite good, often yielding recommendations similar to the original MLT algorithm, which suggests that the previous method had some merit. However, it was also able to return appropriate results when MLT did not and actually appeared to have contextual understanding. TogetherAI models, while providing better contextual understanding, often struggled with long documents, necessitating truncation, which might lead to loss of important information. Doc2Vec's efficiency and cost-effectiveness make it a strong candidate for continued use, especially when combined with pre-filtering for industry.

For interactive visualizations, see the version of this post on my website.

Limitations

One significant challenge is scaling the system to handle larger datasets effectively. I tested on three datasets: one of 2.8k documents (San Francisco Walgreen litigation documents), 17k (Tobacco Products Liability Project collection), and one of 30k (my random sample of all the IDL documents). None of these compare to the roughly 20 million documents in the IDL currently, so there would likely be unpredictable complications when scaling.

Next Steps and Future Investigations

Autoencoders - It would be interesting to study the potential of autoencoder models for generating higher-quality, context-aware embeddings specifically for our datasets and compare them to more general-purpose dimensionality reduction algorithms like UMAP and PCA. Performance would be an extremely important factor due to the size of our datasets, so I could specifically research lightweight autoencoder models.

Matryoshka Embedding Models - These are machine-learning based models trained to generate valid embeddings at a range of dimension sized, capturing the most important information in the first few dimensions. These perform better than standard embedding models overall, and could eliminate the need for extra dimensionality reduction algorithms.

Dataset for Similarity Quality - Compile or curate a benchmark for accuracy of document similarity algorithms. This would likely be subjective, but by averaging many opinions I could potentially generate a useful testing dataset that would allow me to systematically test and measure the performance of various embedding models, better quantify the information lost by dimensionality reductions, and formalize some hyperparameter optimization.

Testing Scalability - I think this would be the most useful research area, as it is incredibly relevant for UCSF. I could perform tests on embedding time, search time, model training time and space, database space, additional optimization algorithms such as prefiltering, and other relevant factors when scaling up to large datasets.

Project 3: Sensitive Information Scanning

Background and Need

The IDL gets large amounts of sensitive data, including PHI, PII, and other information that is better left off the public internet. This data has often been processed through tools like Everlaw or others, which provide some automatic redaction services for easily recognizable data. However, these automatic tools don’t catch everything, so there must be an element of human review. But humans can’t look at every single document, so there are likely at least a few bits of private data that make it through the process.

This is a challenging process to totally automate, however, as we really can only access the OCR text easily, but this OCR is often of a low quality. Therefore, automatic algorithms that are scanning for sensitive data in the OCR text may miss data that is perfectly legible to a human looking at the original document, but that is scanned in badly, handwritten, or written sideways.

My project idea was to scan for some of the low-hanging fruit and partially corrupted data that automatic redaction tools might have missed, all from the public IDL database.

Methodology and Tools

The main tool I used for this project was txt-ferret, a tool for scanning databases for sensitive information using regular expressions. It comes pre-configured with many regular expressions to identify credit card numbers of various types, but I also added many of my own to help identify social security numbers.

However, there were a few difficulties in this process. For one, I wanted to scan the entire UCSF database, which meant that scanning individual files would be impossible. Therefore, the only option was to scan the full compressed files, without decompressing them entirely. Txt-ferret includes a feature to do this by decompressing parts of the file at a time.

Additionally, not all the detected credit cards or social security numbers will be legitimate. Many might be accident detection of other numbers with the same format as that type of data. Thus, txt-ferret allows for the use of sanity-checking algorithms, such as the Luhn algorithm for credit card numbers. These can filter out many of the accidental detections, but even so there are a lot of false positives.

Results

This process flagged about 28,000 credit card numbers, inlcuding false positives. Many were receipts from the 90s so wouldn’t be a whole lot of use to people trying to steal money, but could be used to identify people in the documents. There were no social security numbers found with strict regular expressions, but a number were flagged when I used much more permissive regexes (e.g. allowing a few exceptions to the format). This is a very good sign that much of the automatic redaction is working, but it also indicates that there may be a few heavily corrupted unredacted ones.

There are also likely things we missed, like numbers that have a few digits incorrectly OCRed more than just one. These are extremely difficult to detect, but could still potentially be pieced together to identify people so they should be redacted if its reasonably possible. I also briefly investigated some of the false positives, and it appeared many were from sequential digits that happened to pass the Luhn test, perhaps indicating that these were from lists or spreadsheets with line numbers.

Project 4: LLM Summarization Sentiment Analysis

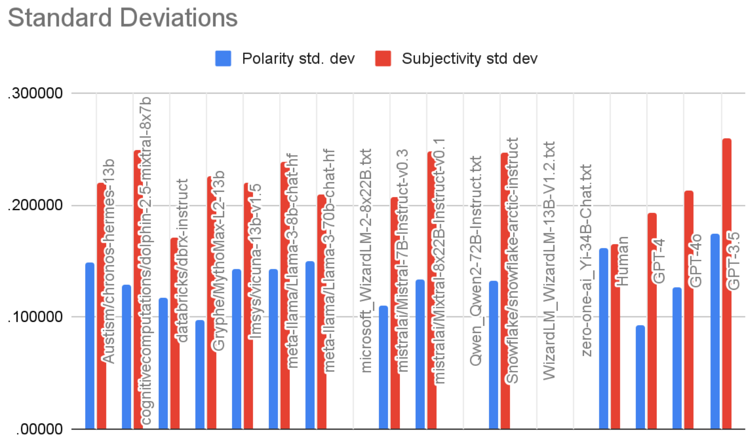
Introduction

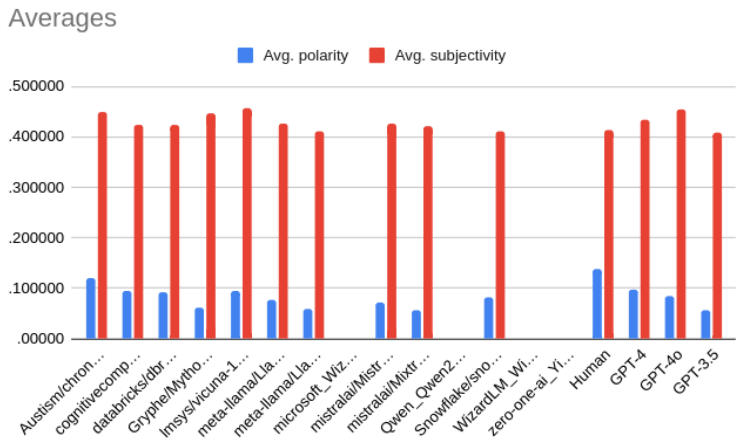
The final research project I worked on was evaluating the bias of LLM summarization with sentiment analysis. This project was relatively brief, but I think the results were quite interesting. One potential application of LLMs is to generate summaries of documents, allowing researchers to better search through them for key information. However, it's possible that these LLM summaries are biased one way or another, which wouldn't be good news for the integrity of research and data. Thus, I investigated the bias of LLM summaries of JUUL documents and compared them to human summaries.

Method

To generate the LLM summaries, I used TogetherAI models and their API to automatically send the OCR of documents in. I instructed the LLMs to generate one to two sentence summaries. I also tested GPT-4, GPT-4o, and GPT-3.5 through Versa Chat, which are the LLMs most likely to be used by UCSF.

To measure the sentiment of summaries, I used TextBlob, a Python library that packages the Natural Language Toolkit (NLTK) for ease of use. Its sentiment analysis measures the polarity and subjectivity of documents. Polarity is on a scale of -1 to +1, where +1 would be a very positive comment, while a -1 would be extremely negative. Subjectivity is measured on a scale of 0 to 1, with a 0 score being perfectly neutral, while a score of 1 would be totally subjective. Average sentiment scores of the LLM summarizations of about 60 JUUL documents were measured and compared to the sentiment scores of a human reviewer.





Results

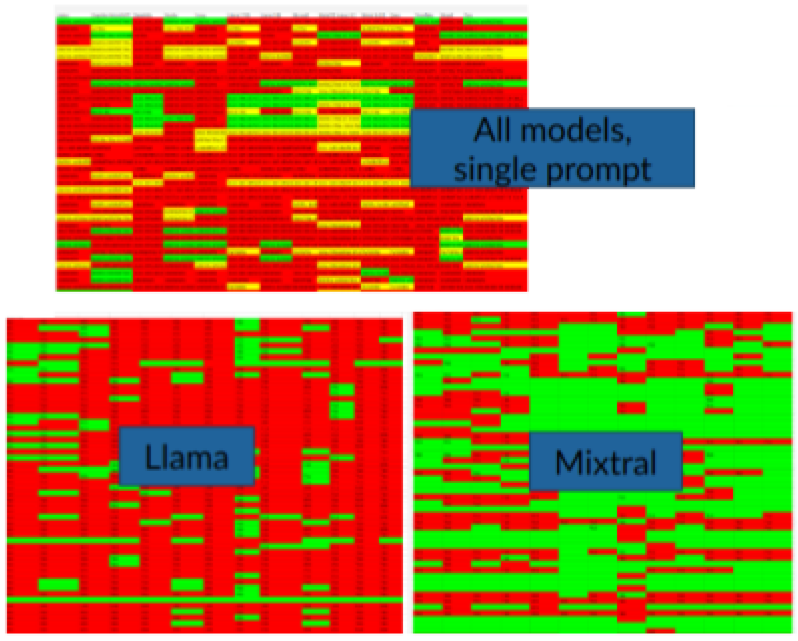
As seen in the charts above, The LLMs were generally slightly positively polar, meaning that they generally talked in favorable terms about the JUUL documents. They were also generally consistent with their sentiment scores, not wildly varying from one document to the next. Interestingly, the LLMs were every rarely negative, and when they were, they were only very very slightly negative. This is acceptable, especially given the significantly higher polarity of the human summaries. As for subjectivity, the LLMs were somewhat subjective (average of about 0.4), generally about as subjective as the human summaries. This is a good sign, as they tend to be more objective than subjective, an important metric in this context.

Meta's Llama models and GPT-3.5 (via Versa Chat) performed particularly well, although it is interesting to note that GPT-3.5 was also less consistent than the other models. However, all in all, most of the LLMs' summaries were about as good if not better than the human summaries, which is to be expected from models trained to imitate human text and responses. It is important to note, however, that this project did not measure the truth, accuracy, or helpfulness of the summaries. It's possible, although unlikely, that the LLMs generated totally wrong summaries of the documents, and further analysis would have to be done to catch this.

Conclusion

In summary, I largely worked on four projects all centered around the applications of natural language processing and data science to the JUUL Labs Collection and the IDL digital collections as a whole. In addition to these, I also helped develop AWS workshops specifically for UCSF researchers in the style of Software Carpentry (available on my website) and looked into the LLM-based categorization of documents.

There are a few key things I learned from this fellowship. For one, I learned that not every project idea is successful. I worked on the LLM categorization of documents for a week or two (where LLMs would label the documents as fitting into certain categories of interest), but I was never able to achieve very good results. To illustrate, you can see a small overview of the results of some LLM categorization schemes to the right (red is bad, green is good). While using certain models and asking a series of yes/no questions to the model helped, their accuracy was still far too low to be useful. I think the root problem was that the categories and their descriptions were a bit ambiguous and difficult for humans to categorize, so there



wasn't even a good metric for verifying the LLMs' performance.

On top of learning multiple new technologies and libraries (such as AWS, regular expressions, TextBlob, and TogetherAI embedding models), I also learned the benefit of hands-on testing and how to deal with large datasets and slow running times, recurring themes in my projects. The IDL database in its entirety is huge, containing about 20 million documents, meaning that running many programs on my laptop would be infeasibly slow. Thus, I had to learn about multithreading, parallelism, asynchrony in API requests, and (de)compression schemes to handle the many gigabytes of data.

I also gained experience writing about my projects in an academic yet informative style, and presenting them at the mid-fellowship Zoom meeting. I think the ability to communicate technical topics is just as important as the work itself. After all, what's the use of research if nobody else can understand it?

In the future I hope to continue research like that I here at UCSF, especially focusing on the applications of NLP to diverse fields like environmental sustainability, policy, and the digital humanities.

Contact Information and Final Thoughts

More specific details about my projects and the live embedding visualizations are hosted on my website at https://generic-account.github.io/. Much of the code is hosted on my GitHub. If you have any suggestions, corrections, potential collaboration ideas, or would just like to talk, feel free to reach out through LinkedIn or the email addresses I have listed on the website!

Thank you again to all my friends and mentors at UCSF. Your support, guidance, and wisdom were incredibly valuable. I loved every moment of my time here and I genuinely cannot thank them enough.