Using the MinHash Algorithm to Detect Plagiarism in the ACL Anthology Reference Corpus

Alex Weatherhead

1. INTRODUCTION

The work makes a preliminary attempt at identifying plagiarism in scholarly work. Our approach to the problem is to use the MinHash algorithm [2], which excels at identifying near-duplicate sentences [6]. Though our procedure could be applied to any corpus of articles, we choose to examine the ACL Anthology Reference Corpus (ARC).

2. THE ACL ARC

The ACL ARC [1] is a collection of 22,878 articles on the topic of computational linguistics scraped from the original PDF files via the Optical Character Recognition (OCR) software and parsed into a logical document structure using ParsCit [3]. Unfortunately, from a glance, it seems that ParsCit makes regular errors, meaning that the text that we use (namely that in the <bodyText> and <listItem> tags) is not always the best representation of the actual article.

In order to use this data for our purposes, six preprocessing steps were performed. First, we removed in-text citations, as citations can be shifted around in a sentence without any change to that sentence's content. Unfortunately, there is no consistent citation style used throughout the corpus. Thus, for the sake of time, we opted to simply remove any text with brackets around it. In future work, a less naive approach involving many cases could easily be implemented with time. Second, we decoded the various html entities embedded throughout the corpus by the OCR software. Related to this, we discovered what seems to be an issue in the ParsCit code which causes """ to be written as """ during the parsing process, which prevents proper decoding to take place. A simple replace all was thus used to rectify this issue. Third, we removed any punctuation which does not represent the end of a sentence. In doing so, we kept in mind some obvious special cases. For example, when removing a hyphen we checked if a newline immediately followed said hyphen and, if not, inserted a space. However, again, as time was a constraint, we did not have the opportunity to consider every possible case, and so this is another avenue for future work. Fourth, all of the text was converted to lower case, because the case of a letter can impact a shingle's hash values despite there being no difference in the letters themselves. Fifth, we removed digits and other symbols from the text, as their use is inconsistent. For example, some articles use a bullet when listing items, while others use numbers. Sixth, all whitespace characters are converted to a regular space character to maintain consistency throughout the corpus.

Shingle	Sentence A	Sentence B
s2	1	0
s1	0	0
s3	1	1

Table 1: The shingles s1, s2, and s3 permuted according to the imaginary hash function f. For each document, the MinHash value is the first row in which there is a 1.

Shingle	Sentence A	Sentence B
s3	1	1
s1	0	0
s2	1	0

Table 2: The shingles s1, s2, and s3 permuted according to the imaginary hash function g. For each document, the MinHash value is the first row in which there is a 1.

3. THE MINHASH ALGORITHM

The central ideas behind the MinHash algorithm are that hashing a vector is roughly the same as permuting it (see Tables 1 and 2), and that the probability that two sentences permuted in the same way yield the same first shingle is equivalent to the Jaccard Similarity of the two sentences (see

Hash Function	Sentence A	Sentence B
f	s1	s3
g	s3	s3

Table 3: The signature matrix generated from the data in Tables 1 and 2. We can use this matrix to obtain an estimate of the Jaccard Similarity of the two sentences by calulating the number of rows in which Sentence A and Sentence B contain the same value.

Table 3). Thus, if we apply several different permutations, and take the first shingle of every sentence each time, we can approximate the Jaccard Similarity.

In practice, the algorithm follows four steps. First, N distinct hash functions are generated using some common hashing scheme. These hash functions are then applied to each shingle within a sentence. The smallest value given by each function is kept as a MinHash signature.

Second, M different bands of K hash functions are selected. Each of these bands thus contains an assortment of rows from the signature matrix, and the rows of the band are concatenated column-wise to form a new MinHash signature for each of the sentences. The choice of M is significant, as the probability of detecting a Jaccard Similarity of s or higher between two documents is $1 - (1 - s^K)^M$ [2]. Thus, it is often crucial to consider multiple bands. Clearly, the choice of K also impacts this probability, but in addition, it reduces the chance of false positives, as it is far easier to match a single MinHash signature than to match multiple MinHash signatures.

Third, sentences sharing the same signature are grouped into candidate pairs. Duplicates can of course arise when two sentences have more than one MinHash signature in common, and so some filtering is often done to remove such pairs in order to save on unnecessary computation.

Fourth, for each candidate pair, we compare the N original MinHash signatures of the two sentences to determine whether or not their similarity is actually s or higher. This comparison involves simply counting the number of rows in the signature matrix in which the two sentences agree and dividing by the total number of sentences.

4. SPARK IMPLEMENTATION

We began by working from the MinHash implementation of previous CS 651 students [4], but soon decided to start from scratch, as their code was not easy to follow, and missed several important aspects of the algorithm; mainly, the use of multiple bands and the filtering of false positives. While neither of these aspects are strictly necessary, they significantly improve the results of the algorithm.

Other than the filtering of pairs based on target and maximum Jaccard Similarity, our implementation closely resembles the MapReduce implementation described by Weissman [6]. We do however, still make use of [4]'s multiply-shift hashing utility.

For ease of use and efficiency we provide a separate Spark program for combining each pair of sentence ids with the appropriate pair of sentences. Those pairs which contain the exact same sentence are filtered out. Though this may seem counter-intuitive, the rationale for this decision is that many of these articles come from the same journals and thus contain the same taglines. Upon some initial examinations of the outputs, it was realized that these sorts of pairs are ubiquitous and make it difficult to find pairs of interest. Furthermore, plagiarism in scholarly works is unlikely to take the form of a simple copy-paste, so removing pairs of identical sentences is unlikely to result in the loss of meaningful outputs. However, one again, this naive approach is simply a quick means to achieving a result, and more sophisticated approaches should be considered in the future to avoid missing any instances of plagiarism.

Hyperparameter	Value
Target Jaccard Similarity (s)	0.60
Number of Bits in Hash Values	60
Number of Hash Functions (N)	20
Number of Bands (M)	10
Number of Hash Functions per Band (K)	10
Number of Characters per Shingle	12
Minimum Number of Shingles to Consider	75
Maximum Number of Shingles to Consider	600

Table 4: The values used for each hyperparameter in our MinHash implementation.

5. RESULTS

We ran the MinHash algorithm using the hyperparameters given in Table 4. The final output consisted of 46,719 pairs. At a glance, many of these matches are still taglines, just with very subtle differences. For instance, consider "... do not necessarily reflect the view of the darpa afrl or the us government" versus "... do not necessarily reflect the view of darpa afrl or the us government". The sentences are identical other than the insertion of a "the" before "darpa". However, there is nothing interesting about these sentences. With such a large number of outputs, it is quite difficult to parse what are and what are not meaningful near-duplicate sentences.

6. CONCLUSION

This preliminary work has accomplished a great deal, in that it has provided an initial insight into the problem at hand; plagairism detection in scholarly works. Specifically, it has allowed for us to perform an initial examination of the data, as well as to create a suitable implementation of the MinHash algorithm for tackling this problem further.

There are many avenues for moving forward with this project, but we believe they can be aptly summarized as follows. First, we believe that it would be beneficial to revisit the logical parsing of the articles in ARC. ParsCit is now outdated, and its authors now recommend using Neural-ParsCit [5]. Better parsing would likely also help reduce the number of taglines that are included in the text. Second, as discussed, more sophisticated preprocessing is possible with time, and should be explored in order to maintain a high quality of data. Third, we are interested in applying additional algorithms, including Machine Learning techniques, for identifying meaningful pairs of sentences.

7. REFERENCES

[1] S. Bird, R. Dale, B. Dorr, B. Gibson, M. Joseph, M.-Y. Kan, D. Lee, B. Powley, D. Radev, and Y. F. Tan. The acl anthology reference corpus: A reference dataset for bibliographic research in computational linguistics. In *LREC 2008*, 2008.

[2] A. Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1997.

[3] M.-Y. Kan, M.-T. Luong, and T. D. Nguyen. Logical structure recovery in scholarly articles with rich document features. *Int. J. Digit. Library Syst.*, 1(4):1–23, Oct. 2010.

[4] Z. Meng and M. Zheng.

[5] A. Prasad, M. Kaur, and M.-Y. Kan. Neural parscit: A deep learning based reference string parser. *International Journal on Digital Libraries*, 2018.

[6] S. Weissman, S. Ayhan, J. Bradley, and J. Lin. Identifying duplicate and contradictory information in wikipedia. In *Proceedings of the 15th ACM/IEEE-CS Joint Conference on Digital Libraries*, 2014.