Text-based Document Similarity Matching Using `sdtext`

Clay Shields
Department of Computer Science
Georgetown University
Washington, D.C., 20057
clay@cs.georgetown.edu

I. Abstract

Forensics examiners frequently try to identify duplicate files during an investigation. They might do so to identify known files of interest, or to allow more rapid review of documents that appear to be similar. Current forensic tools for detecting duplicate files operate over the low-level bits of the file, typically using hashing. While this can be a fast and effective method in many cases, it can fail due to differences in file format. We introduce `sdtext`, a tool developed to identify similar files based on their textual contents, which is robust to changes in format. We show that `sdtext` is far more accurate than existing tools in matching files that contain the same text in different formats.

II. Introduction

As technology improves and prices fall, the amount of digital media found in a forensic examination increases. For examiners in the field, this can create problems as they may recover a large amount of digital media requiring triage to determine what is interesting or relevant. One method of isolating interesting media is to identify which media contains files similar to those already known to be of interest. Similarly, identifying near-duplicate material can allow an examiner to more quickly work through large amounts of data by examining similar documents at the same time rather than piecemeal as they are encountered.

One approach to finding similar files is done on the basis of hashes that operate over the bits of the file [1], [2]. In many cases, however, this approach can fail to identify file containing similar content in different formatting or encodings. An example of this is a Microsoft Word document that has been exported as a PDF. While humans would perceive the same text, hashing will likely not identify documents as being similar. Other examples might be different classes of documents with similar text, like web pages from the same source or documents with similar boilerplate text.

To improve the state of the art in matching documents that contain similar text, we have developed a tool named `sdtext`, which is available under an open-source license at sdtext.com. It operates by extracting text from files, constructing a dictionary of statistically important terms, then parsing files and recording the presence or absence of those terms. Digests are either hashes of the set of dictionary terms or a bit vector representing which terms are in the dictionary. Digests are portable and can be shared between investigators, and they can be matched against each other to find files that have similar content.

Below, we describe how existing approaches work, and then provide more detail about the operation of `sdtext`. We then present an experimental comparison of `sdtext` and `sdhash` : first, in a general case; next, over files containing text; and finally over files consisting of extracted text. We show that `sdtext` is able to identify similar files that are textual with much higher accuracy than existing tools, particularly in cases in which the same text is present in different file formats.

A. Similarity Hashing

Existing forensic approaches to duplicate document detection operate based on the bits that encode the file. In the simplest case, cryptographic hashes are used to identify files that are bit-for-bit identical. This is used most often to identify known child pornography and to identify known system and application files to exclude from further examination [3]. These techniques do not work if even a single bit is changed in the file and consequently they fail to identify similar files that have been subjected to even small edits.

There have been a number of approaches taken to allow hashing to match file based on similarity. All involve breaking the file into smaller pieces and hashing those pieces, then comparing the piecewise hashes to find matches. The difference is in how pieces are created for hashing.

The simplest approach is to use pieces of fixed size, as is used in `dcfldd` [4]. This allows an examiner who is taking a forensic image to verify that image piecewise; if some piece is corrupted, the integrity of other pieces is maintained. When applied to document recognition, this approach suffers from an alignment problem. Should a single bit be added or deleted from the file, that block and all following blocks will have differing hashes.

A better approach is called context triggered piecewise hashing (CTPH), and was built into a forensic tool named `ssdeep` [1]. First used for spam detection [5], CTPH uses a rolling hash to determine when to trigger computation of a

piece hash. The effect of similar files is to trigger the rolling hash in the same places even if some portions are altered. In practice enough similarities remain that many of the piece hashes match.

An issue with CTHP is determining how often to create piece hashes; this frequency is related to parameters controlling the rolling hash. Rather than attempting to determine an optimal frequency, a more accurate approach is to compute hashes for many different average piece sizes at once, then to compare among pieces computed using the same parameters. This approach is used in multi-resolution similarity hashing [2], which operates over a byte stream and creates multiple hashes on that stream with differing rolling-hash parameters. The resulting piece hashes are placed in Bloom filters [6], one for each parameter chosen. Later, when other files are compared, their hashes are compared against previously computed hashes in the Bloom filter.

There are several advantages to using hashes to find similar documents. They are file-format independent, and can be used across many different file types. They are fast to compute and do not require significant storage as well.

Their disadvantage is that they do not reflect the way computer users see files, which is based on file content and not their encoding. For example, a Word document can be saved in a variety of file formats, such as text, PDF or HTML. Those files would be considered identical to a person, but would not be similar enough on a bit level to be seen as a match using existing tools. This creates problems for the forensic examiner, who might be trying to match similar files between systems. For example, on one system the file might be an email message in text or `.eml` format; on the other it might be part of an HTML cache file holding the page from a webmail server. Current similarity tools might miss this, and we show many instances where the probability of such a miss is high.

To help rectify this, we introduce `sdtext`, which matches files based solely on content, allowing a much higher accuracy.

III. Similarity Matching with `sdtext`

The operation of `sdtext` is based on principles from information retrieval, the general area of computer science research that powers search engines such as Google and Bing [7]. The techniques used are those used in duplicate document and plagiarism detection [8] and are novel only in their application to forensics problems. While the IR community has studied duplicate document detection, most work has been on finding similar documents in a single collection of files. This differs from some forensic applications in that an examiner may require a portable digest that can be shared with other examiners without sharing the full set of files from which the digest was first created.

A. Overview of operation

`sdtext` operates by performing some statistical analysis of the files that are to be compared or of files which are expected to be similar to those being compared. This analysis consists of computing the *inverse document frequency* (IDF) of each term found in a file, which, for each term T in a set of documents D, where a document containing term T is denoted as D_T, is defined as:

$idf_T = log \frac{(|D|)}{1+|D_T|}$

The result of this analysis is a *dictionary* that contains a list of statistically significant terms. As described below, this dictionary is then used in creating file digests. Each digest contains some information about the file as well as a bit array representing information about which dictionary terms were found in the file.

To approximately compare two digests, `sdtext` uses cosine similarity. This treats each bit array as a vector in space where the number of dimensions is the same as the number of bits in the array. The two digests are considered similar if the cosine of the angle of the vectors falls within a specified range. This range is a user-selectable parameter that allows the examiner to tune the results they see. This is analogous to looking down the list of results on a search engine; the lower-ranked results might also be a match, though the lower the score the less the likelihood.

`sdtext` therefore operates in three distinct rounds. First, a dictionary must be created, which requires reading files and performing statistical analysis. Second, digests of files are constructed using this dictionary, which can also be shared with other examiners so that they can create digests as well. Third, digests are compared in order to find matches. We describe each of these operations in more detail below.

B. Text Extraction

The operation of `sdtext` is predicated on being able to extract text from files found during an investigation. While simple in concept, the number of file formats that exist can make this difficult in some cases. Fortunately, however, there are variety of approaches that can be used, particularly for common file formats.

For files that contain text but have a proprietary file format, a technology like Oracle's OutsideIn [9] can be used to extract the text. We have included the necessary hooks in `sdtext` to operate with OutsideIn, but do not distribute OutsideIn with `sdtext` due to licensing limitations.

For more common file formats, such as Microsoft Office documents or PDF files, there are a wider variety of choices for text extraction. One can use native versions of programs that can read the files and write them as text. A scripting language that can interface with the program can make bulk translation easier; for this paper, we used Applescript and Microsoft Office to convert hundreds of Office files to text and other processable formats.

In addition, there are other programs that have reverse-engineered file formats and allow text extraction. We used several, including OpenOffice and unoconv, a command-line tool that uses OpenOffice's libraries, to extract text from Microsoft Office Files; pdftotext, a command-line linux program that extracts text from PDF files; links, a command-line browser that can dump text from HTML files and web sites; and unrtf, a command-line tool that extracts text from RTF files.

In cases in which other methods fail, it is possible to extract text from printable files using optical character recognition. In our experiments below, we used the open-source OCR program tesseract-ocr to extract text from PDF files.

We also note that not all text is the same as there exist a variety of possible encodings and line endings. Some of the most commonly encountered include ASCII, ISO 8859-1, and UTF-8. Different operating systems also use different markers to indicate line endings. As part of our experiments, we varied the encoding and line endings. We used iconv to convert between encodings; dos2unix to change line endings; and fold to shorten long lines by adding additional line endings.

It is important to note that not all text-extraction mechanisms produce identical text. Optical character recognition adds many additional characters as noise, and other methods treat special characters, like ligatures or smart quotes, differently. Because of this, any method that works to recognize text needs to be robust against small differences in the text. We show below that sdtext is robust in this way, whereas sdhash - due entirely to design differences - is not.

C. Tokenization

During processing, each file that is read is *tokenized*. The tokenization process splits the text into smaller segments, typically words or lines, each of which is one of the units used in making digests. We note that it is necessary to extract text from files during this process, and described approaches to recovering text in section III-B, above.

The tokenization to be performed can and will vary, particularly by language. Tokenizers are generally used as a set; each file that is read is passed through all the tokenizers. In sdtext, each tokenizer has a name that reflects its function, and the first tokenizer in the list must be able to read the file to be processed. The current tokenizer set includes file-reading tokenizers that can read from text and gzipped text files; can use Oracle's OutsideIn for text extraction from a variety of file formats; and which can use Apache Lucene [10] for processing Chinese and Arabic language documents.

Other tokenizers then filter this text by performing such operations as splitting tokens on punctuation or numerals or filtering tokens based on length, which can be useful in removing tokens that are a byproduct of text extraction, such as when unencoded or base64 encoded text is part of a document. Some languages, such as English, benefit from *stemming*, which is the process of removing word endings but leaving the root. This helps ensure that words like "ensured" match "ensuring". sdtext includes the frequently-used Porter Stemmer [11] to accomplish this.

D. Dictionary creation

Creating a dictionary requires a sample of files that are expected to be similar to those being matched. This can be the entire set of documents to be compared, rather than some prior collection, but a set of files in the same language can suffice. Dictionaries are created by processing all files and discovering what tokens appear in which documents. We then extract statistically important terms using the inverse document frequency, or IDF, as described above.

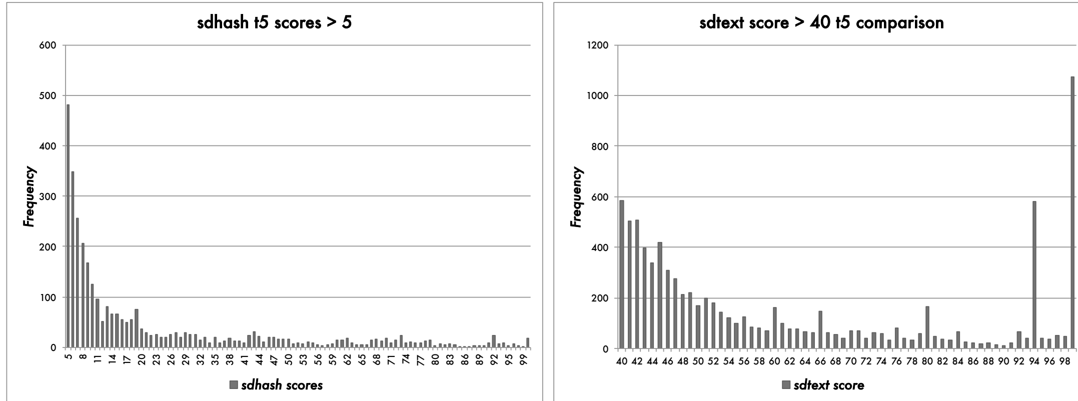
sdtext then trims the set of gathered tokens by IDF, dropping the low IDF and the high IDF terms. The reason for this is that low-IDF terms are very common and require storage space while not adding significantly to the process of differentiating files. Similarly, high-IDF terms are removed as they are so uncommon as to appear in only one or a few different documents in the collection. While they can help identify those few files that contain those terms, they are useless otherwise and can be dropped to preserve space.

Determination of appropriate IDF settings can be made by experimentation, or the examiner can use a heuristic. sdtext includes the ability to test a variety of IDF ranges to determine which would produce the most accurate results, though this can be computationally expensive. Alternatively, it is possible to use language-based heuristics. For example, a normalized IDF range of approximately 0.3 to 0.6 often works well for English language documents, and is used in our tests below.

E. Digest creation

Once the dictionary has been created, text digests can be created. Each document is tokenized to extract tokens of interest. From this set of tokens, we currently support two different types of digests. First, sdtext supports hash-based digests that are made by taking all dictionary terms that are present, alphabetizing and concatenating them, then hashing the resulting string. This is the approach used in I-Match [12], and produces compact hashes which can be used only for exact matches between files.

A better approach to similarity matching, however, uses bit vectors. Each bit in the vector represents the presence or absence of a dictionary term in the file being processed. In a sense, these are a semantic representation of the file in a lossy format. These digests allow more flexibility in matching, as files that contain a sufficient number of matching terms with a smaller number of non-matching terms will still result in a high similarity score.



(a) `sdhash`, score $>$ 5　　(b) `sdtext`, score $>$ 40

Figure 1: All-pairs matching of the T5 data set

F. Digest Matching

Digests can be matched against each other. For bit-vector digests we use a cosine similarity measure, which is the measure of the cosine of the angle between two vectors represented in n space. Investigators can specify a parameter that represents the maximum cosine of the angle between the two vectors that can be considered a match, making it possible to control the precision and recall of their searches to a degree. Increasing it results in higher precision but a lower recall. An investigator can use this fact to perform initial, highly precise searches then expand the scope of the searches by decreasing the matching parameter. These later searches will return a wider variety of matches which are less accurate but contain more possibilities.

G. Score Interpretation

As output, `sdtext` and other similarity matching tools produce an integer number between 0 and 100. While this number has mathematical meaning, the fact that two files have a 100 score does not guarantee that they are identical, and in fact there can be significant differences between such files. It is common for files that have a specific boilerplate to match, though the topics may be different. For example, in our data set, described below, we find that web pages from the same site often match, as do calls for grants from the same agency as they contain standard text.

A better way of thinking about the scores relates to search engine performance. Matching in `sdtext` is essentially performing a query based on the terms recorded in the digest against the terms in other digests. A high score means it is likely the match will be of interest; a lower score less so. In general, for English text, an `sdtext` score of above 60 seems to be worth examining. For our experiments with `sdhash`, we use the cut-off of 20 as a reasonable value as cited by Roussev [13].

IV. Matching Performance

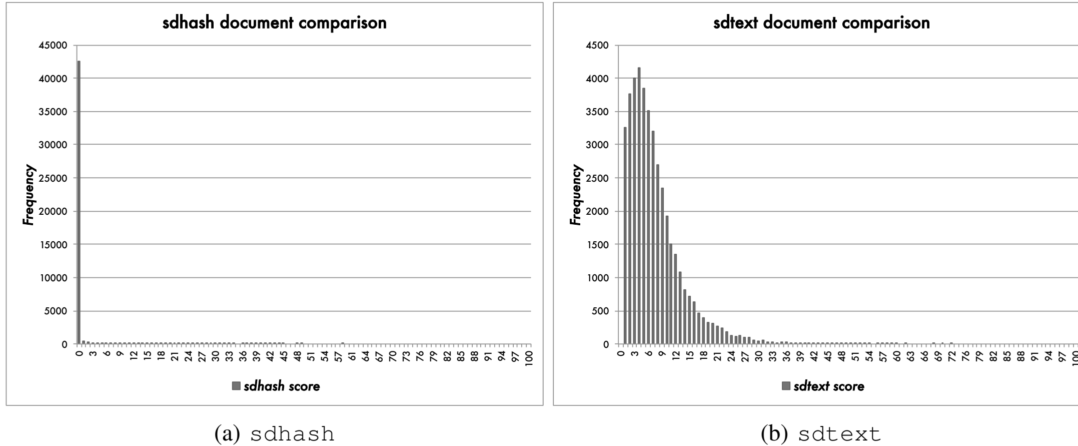
We first compare the performance of `sdtext` against that of `sdhash` using the methodology and data corpus described by Roussev [13] using a variety of text extraction techniques. We show that in the general case composed of a corpus that contains a significant proportion of non-textual files, such as images, `sdtext` does not perform as well as `sdhash`, as one would expect. We then move on to files that are primarily textual and show that `sdtext` outperforms `sdhash` in terms of accurate matching of files that contain similar text in different formats.

We do not include performance comparisons in this paper. In performance, `sdhash` is far superior. Where as `sdtext` may need to read the files twice - once for dictionary creation and once for digest creation - `sdhash` only has to read the files once. Once digests are created, `sdtext` must individually compare all digests of interest, resulting in an $O(n)$ or $O(n^2)$ operation. In contrast, `sdhash` can place all digests in a Bloom Filter and do an $O(1)$ operation to determine if there are any matches. That being said, `sdtext` is highly threaded and text processing is a relatively fast operation. Our experience with `sdtext` shows that it runs fast enough, even over large data sets, to be of practical use.

A. t5 corpus

Past work by Roussev has compared the performance of similarity tools [13]. We add to that body of work by performing the same tests using the t5 data corpus he produced and comparing them to the results from the most recent version of `sdhash`. We do not expect `sdtext` to perform well on this dataset, as it includes a variety of files including many that do not contain text.

We show the results in this section as a histogram of scores of individual files from the test set. In general, the more high scores the better the result; as a heuristic, a `sdhash` score of 20 or greater would indicate a likely



(a) `sdhash` (b) `sdtext`

Figure 2: All-pairs matching, doc corpus

match, for `sdtext` a score of 60 or greater might. The results for `sdhash` are shown in Figure 2a, and those for `sdtext` in Figure 2b.

For this experiment, we used version 3.3 of `sdhash` and `sdtext` version 0.4 with automatic text extraction produced using OutsideIn. In configuring `sdtext`, we used minimal tokenization in that we removed tokens with punctuation, limited the length of tokens to between 3 and 10 characters, and performed Porter Stemming. We then compared created digests of each file in the t5 corpus and did an all-pairs comparison between digests and recorded the resulting scores. We performed a similar operation for `sdhash`, setting the minimum score to report to zero and selecting the option to generate an all-pairs comparison, again recording the scores.

For each tool, the vast majority of pairs do not generate a matching score. For `sdhash`, 1,073 of 9,965,880 score above the `sdhash` cut-off score of 20, which is about 0.01%. In comparison, `sdtext` reports fewer total matches, providing scores for only 5,095,020 pairs. The reason for this is because many of the files in the t5 corpus produce no text when OutsideIn processes them. Files that do not produce text do not produce digests, as such digests would be useless, so they are not included in the matching process. Of the scores reported, 3,389 were above the `sdtext` cut-off score of 60, which is about 0.07%. Figure 1a shows the results for `sdhash`; Figure 1b shows those for `sdtext`.

Surprised at the seeming effectiveness of `sdtext` in a situation where we did not expect it to perform well, we took a closer look at the results. We noted that the high scores in the 99 and 94 columns consisted almost entirely of JPEG images. Checking the text extracted, we found that OutsideIn was reporting standard language EXIF headers that were causing a match. Removing the image files from comparison gave us the results shown in Figure 3. Examining the remaining hits showed reasonable results: for example, the top hit was between web pages from the same site; the second hit data sets in the same format.

Overall, however, `sdtext` doesn't perform as well in the general case as `sdhash`, which is fully expected. In particular, `sdtext` was unable to create digests for many files due to the lack of text to be extracted. In the next section we describe experiments on textual files and show that `sdtext` outperforms `sdhash` in this specific, textual domain.

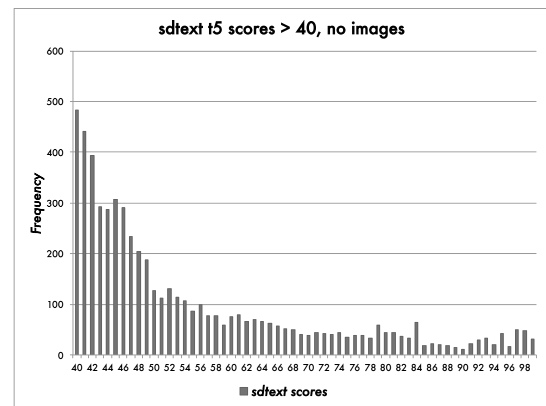


Figure 3: `sdtext` scores over T5 data set, score $>$ 40, images removed

B. Experiments with Textual Files

As `sdtext` is designed to operate effectively over text files, we designed and conducted experiments to demonstrate performance on pure text files. We developed a test corpus, as described below, and converted the text in each Microsoft Word document to PDF, HTML, OCR'ed text, and text. We then compared each Word document against all other versions of the same document using `sdtext` and `sdhash`. Performing experiments in this manner allows us to have a solid ground truth. Each different format of the same file should, ideally, match each other, as they contain the same

text when viewed by humans with the appropriate viewing software.

Corpus Creation: We first selected a random subset of 500 Microsoft Word files from the GOVDOCS [14] corpus. From this set, we removed documents that were only one page, as many of those were blank forms with little text. We then briefly examined each document to ensure it was in English and that when saved as PDF it created a single document; many of the .doc files created two or more PDFs when printed. Though we had a goal of 300 text documents, we ended up with 299 as late in testing it was determined one was in Spanish, so its results were removed. This resulting set of documents, which we will refer to as the doc corpus, are available for download from anonymized-for-review.

Format conversion: For each document, we converted it from Word format to PDF by using the option to print as PDF through Microsoft Office on a Macintosh desktop through a small program written in Applescript. We similarly converted the documents to HTML in the same manner, keeping only the HTML portion and not the corresponding subdirectories. The PDF files were then run through `tesseract-OCR` to produce a text version through optical character recognition.

1) All pairs doc files: Our initial test was to compare each doc file against all others. The results are shown for `sdhash` in Figure 2a and for `sdtext` in Figure 2b. For `sdtext`, we used OutsideIn's text extraction, removed tokens with punctuation, limited the length to 3-10 characters and performed Porter Stemming. This is a demonstration that, for the most part, the set of files extracted are not generally self-similar. It also shows that `sdhash` identifies matches that `sdtext` does not.

To determine why this was the case, we examined all `sdhash` matches that scored above 20 and all `sdtext` matches that scored above 60. For `sdhash`, there were 108 such matches. Of those, a brief examination of each match revealed that 49 appeared to be legitimate, mostly due to cases in which the documents were produced by the same agency and had identifying header text and embedded logo image. The other 59 matches were apparent false positives. Interestingly, most of these false positives involved the same 10 or so files. A hypothesis is that there may be some embedded code that is not visible, perhaps something like a font definition, that is triggering those matches.

In contrast, `sdtext` had 4 matches above a score of 60, three of which were documents from the same organization, and the fourth of which (and the lowest scored) was a false positive. This shows that each approach has a strength, as `sdhash` found many matches that `sdtext` did not at the cost of many false positives. Meanwhile `sdtext` found one match `sdhash` did not, but was more precise in what it reported.

2) Comparison between doc and PDF: We next moved on to comparison of the same text in different formats. We tested each document only against its re-formatted copy. As mentioned above, this experiment provides a strong measure of ground truth, as each comparison should be true. We started by comparing each doc file against the corresponding PDF file. For this, and all the comparisons between formats below, we used OutsideIn's text extraction, removed tokens with punctuation, limited the length to 3-10 characters and performed Porter Stemming. The results are shown for `sdhash` in Figure 4a and for `sdtext` in Figure 4b.

It is clear that something about the PDF conversion process badly affects the ability of `sdhash` to match a Microsoft Word document to its corresponding PDF file, as none of the matched pairs returned a score other than zero. In contrast, `sdtext` performed quite accurately, with only two pairs failing to meet a reasonable score, though they were close. This is a 99.33% success rate.

3) Comparison between doc and HTML: As a conversion to HTML can be a common transformation, we compared each doc file to its corresponding HTML file. We note that the HTML as produced by Microsoft Word is quite verbose, and that a different conversion method or post-processing of the HTML to reduce its complexity might produce different results. The results are shown for `sdhash` in Figure 5a and for `sdtext` in Figure 5b.

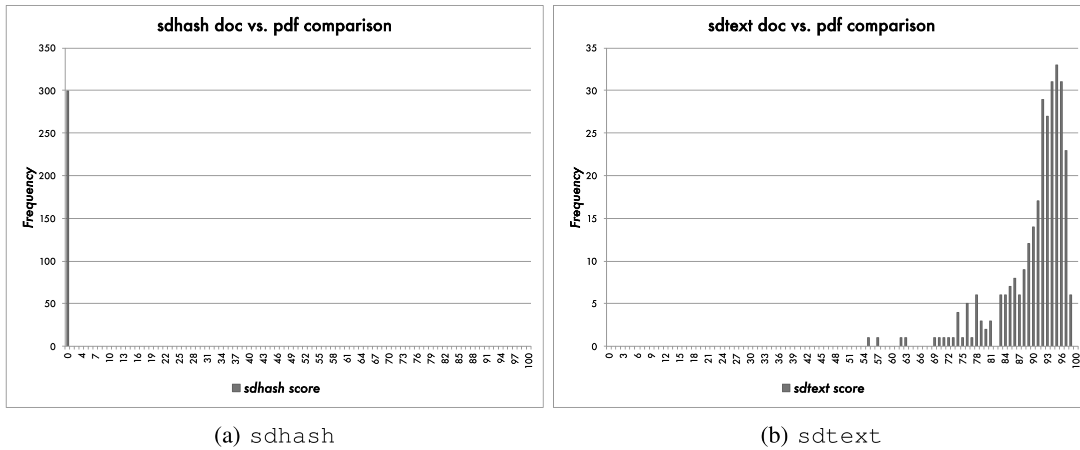
Again, `sdhash` performed poorly, matching only 1 pair out of 299, for a success rate of 0.33%. `sdtext` did much better, using the same tokenization as above with a 100% success rate.

4) Comparison between doc and OCR text: Though text extraction is possible in many common electronic formats, in the case where the files are in image form or exist on paper text can be retrieved using optical character recognition. This process, while often effective, can be very noisy and error prone. We note that our OCR corpus was produced using existing PDF files, and is likely "cleaner" than copies that were scanned from paper. Examination of the resulting text files does, however, show that there are a significant number of additional characters introduced into the file.

Again, as shown in Figure 6a and Figure 6b, `sdhash` performed poorly in this specific case, showing a 0% success rate, while `sdtext` missed only 3 matches for a 98.99% success rate.

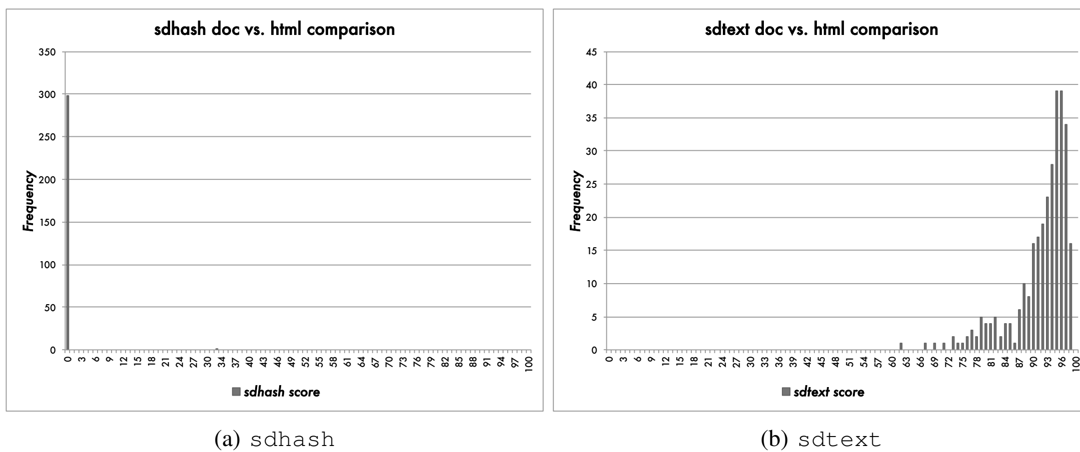
5) Comparison between doc and alternate text extractions : Finally, we compared doc files against text files that had been created using the "Save as text" function of Microsoft Word. The text produced this way differs from that produced by OutsideIn, which was the default for `sdtext` in this experiment. The results are shown below in Figure 7a and Figure 7b.

In this test, `sdhash` was able to match a good portion of the files, scoring 220 out of 299 as matches for a success rate of 73.6%. Again, `sdtext` did well with a 100% success rate, showing that it is robust to small differences in text



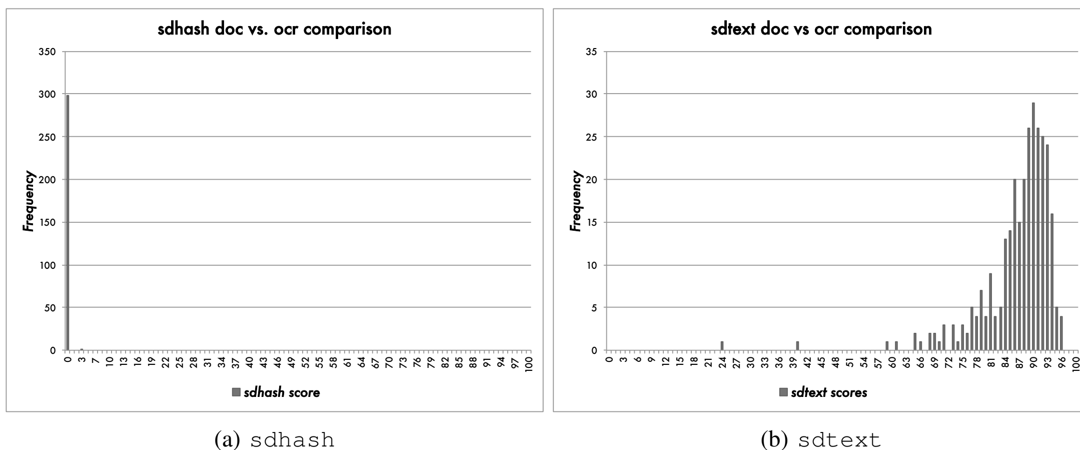
(a) `sdhash` (b) `sdtext`

Figure 4: Matching doc files to PDF files



(a) `sdhash` (b) `sdtext`

Figure 5: Matching doc files to HTML files

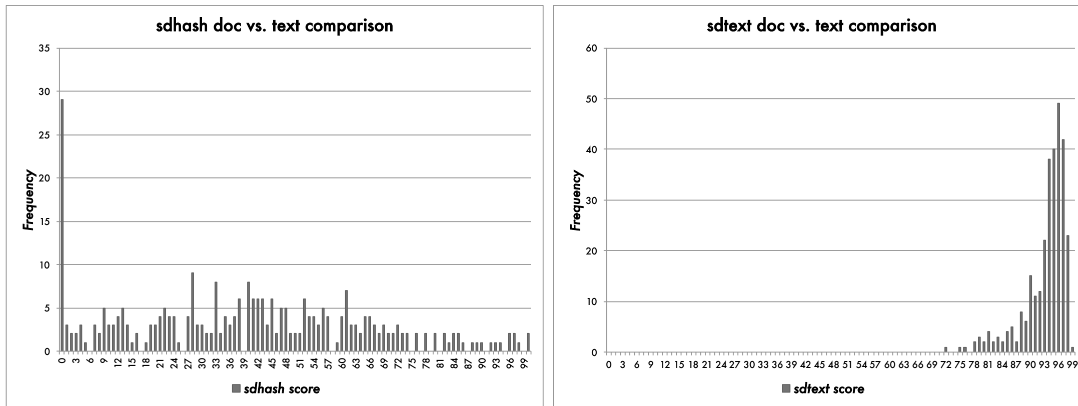


(a) `sdhash` (b) `sdtext`

Figure 6: Matching doc files to OCR'ed text files

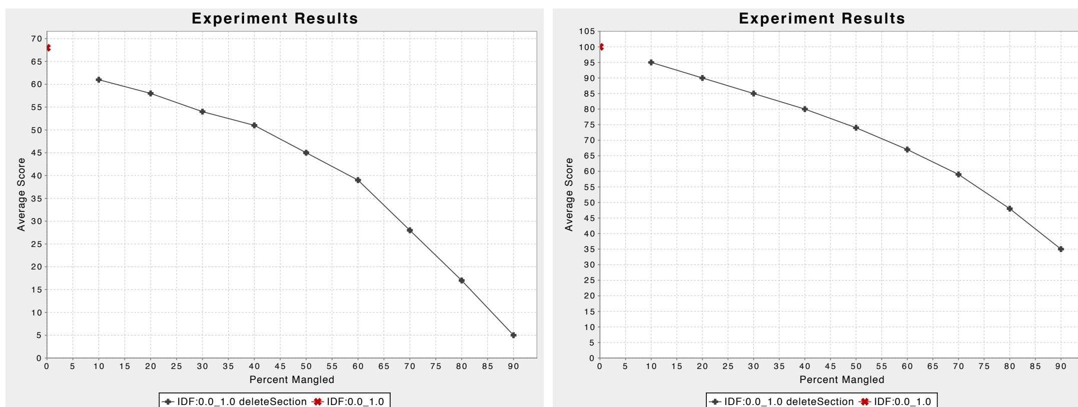
extraction.

6) Results: The results show that `sdtext` is highly accurate at matching files containing text across a variety of formats. While `sdhash` is an excellent tool in general use, it performs less well in this specific case.



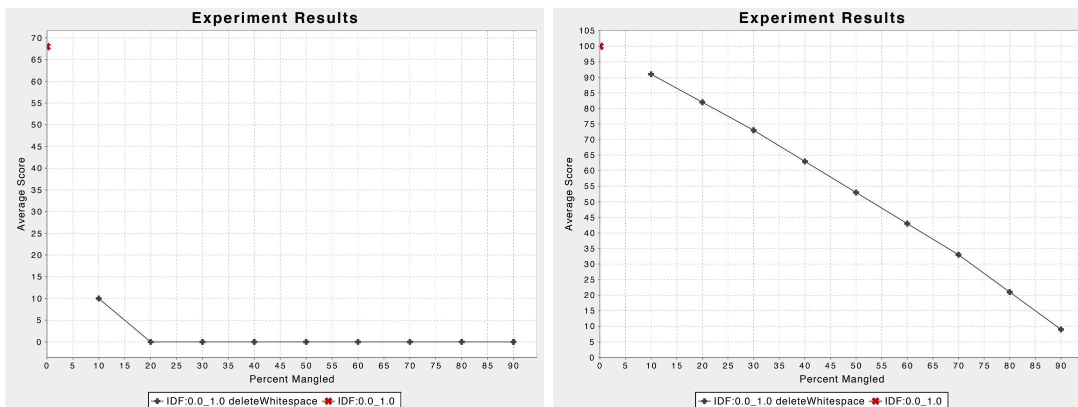
(a) `sdhash` (b) `sdtext`

Figure 7: Matching doc files to text files



(a) `sdhash`with Deleted Sections (b) `sdtext`with Deleted Sections

Figure 8: Matching text to text with deleted sections

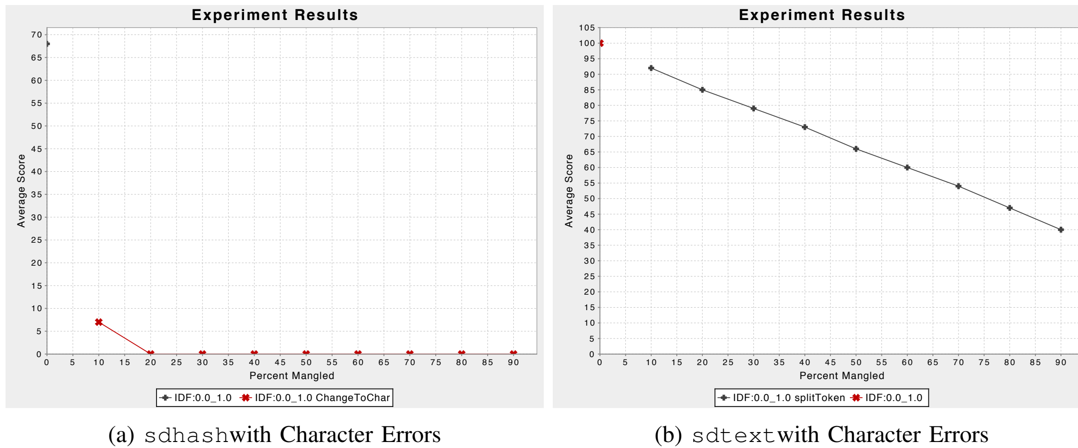


(a) `sdhash`with Deleted Whitespace (b) `sdtext`with Deleted Whitespace

Figure 9: Matching text to text with deleted whitespace

C. *Using* `sdhash` *on extracted text*

Given that `sdtext` operates over extracted text while `sdhash` needs to operate over all formatting, the question naturally arises if the text extraction is what provides the advantage. Would `sdhash` do as well if provided the extracted text to operate on?



(a) sdhashwith Character Errors

(b) sdtextwith Character Errors

Figure 10: Matching text to text with deleted whitespace

To determine if this is the case, we ran a series of experiments over extracted text. For sdhash, we used english-language text and normalized it by making the text lower case; removing punctuation; and replacing all white space with a single space. We refer to sdhash working over extracted text as sdhash-E. For sdtext, we extracted text; split tokens on whitespace; stripped digits and punctuation; removed short and long tokens; then applied Porter stemming.

We then also applied automatic error insertion into the text at varying levels. These manipulations included deleting a contiguous chunk of the text; deleting whitespace tokens with a percentage chance; or changing individual characters to random other characters with some percentage chance. These edits were intended to represent different situations that would occur in forensic situations. Recovering partially overwritten files might result in only recovering a small portion of the file. OCR of documents might result in erroneous characters, or missed whitespace characters. We show that each of these affects the operation of sdtext and sdhash-E differently, but that sdtext is generally more resilient to errors in the text.

1) Deleting Sections: In the first experiment, we matched a large number of source plain texts files from the Enron data set [15] against the same file with a contiguous section deleted. The results are shown in Figures 8a and 8b, which recount the average score over multiple trials for each deletion size. The 95% confidence interval is calculated, but is too close to the plotted points to be visible. As a reminder, scores above 20 are reasonably interpreted as matches for sdhash , and scores above 60 are reasonable matches for sdtext.

The graphs show that both sdhash-E running over extracted text and sdtext do a good job at matching segments of files to the original file. In this case, sdhash-E is able to provide a high confidence score to a slightly larger deletion percentage, on average matching documents that had 77% of the text removed sdtext was behind, matching on average with about 70% of the text removed. This indicates that sdhash-E can perform slightly better in the case of matching contiguous fragments of text to files.

2) Deleting or Changing Characters: We also considered other edits typical of the errors that can occur in matching noisy text, like that is created by OCR or recovered with other noise. In this case we tried two different experiments. In the first, we randomly deleted whitespace characters in the file. This is one type of OCR error that is common. The results are shown in Figures 9a and 9b. In this case, sdhash-E performs very poorly, failing to match files with 10% of white space removed. sdtext performs much better, matching the files on average with as much as 40% of the whitespace removed.

Similarly, we examined what would happen when OCR-type errors altered individual characters. In this experiment, individual characters were randomly changed with the specified probability. The results are shown in Figures 10a and 10b. Again, sdtext is robust against about 60% of the characters being randomly changed; sdhash-E fails with even 10% of characters changed.

3) Results: The results show the different approaches of sdtext and sdhash cause different performance. Because sdhash creates hashes over sections of text, any change in that section means it will not match the original section. However, given a clean hash of a small section, it will still match the original document. This gives it better performance than sdtext in matching segments, thought the difference is not major.

On the other hand, sdtext is far more resilient to errors in characters in a file than sdhash. This indicates that it is likely preferable in situations when the type of error that might be present in recovered text is not known

sdtext. We used the base set of text files created using Microsoft Word as described above. We then compared these

files to the set of OCR'ed files, called the OCR set; a set of files that was created using the unoconv utility on linux, called the unoconv set; and the base set of text files that had line endings changed with the dos2unix utility and the line length wrapped to 80 characters using the fold utility, called the folded set.

V. Limitations of sdtext and Future Work

While sdtext is very accurate at matching text files, it has limitations compared to existing tools. First, the processing overhead is higher. To match a new set of files, sdtext must read the files once to create the dictionary, then again for the signatures. For large data sets this can induce a significant overhead. Second, there is an additional overhead required to load the dictionary. In some cases, this can be expensive, particularly when sdtext is run repeatedly to create fingerprints, as the dictionary must load each time. Third, the digests are not hashes and require more computation to match. The hashes produced by sdhash and ssdeep can be inserted into a Bloom Filter [6] that later allows very fast checking of the presence of a matching hash. In contrast, the digests produced by sdtext must be checked individually to determine the score between them. Finally, sdtext is written in Java, which does not have the inherent performance of C or C++ which other tools are coded in.

Our planned future work intends to address some of these issues. First, we plan to place all tokens from each file as it is read into a Bloom filter. Instead of reading the file a second time, the Bloom filter will be probed to determine which tokens from the dictionary were present, removing the need to read each file twice. Second, we are planning to implement an improved algorithm for all-pairs digest scoring [16]. This approach limits the number of comparisons that need to be made. Third, while sdtext is already heavily threaded, we are planning on examining other options for parallelism, including Apache Spark.

VI. Conclusion

Similarity matching is an increasingly useful technique for working with large amounts of acquired evidence. Most similarity tools work over the bitstream of the file, which, while useful, does not do well in finding documents with similar text in differing formats. We introduced sdtext, which matches similar text files on the basis of their contents. We showed that while sdtext is not as suitable for finding duplicates in the general case, it is more accurate at identifying textual file matches than current tools.

VII. Acknowledgements

I would like to thank Lindsey Neubauer and Evan Bloomberg for their work on sdtext.

References

[1] J. D. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, no. Supplement-1, pp. 91–97, 2006.

[2] V. Roussev, G. R. III, and L. Marziale, "Multi-resolution Similarity Hashing," in *Digital Forensics Research Conference (DFRWS)*, 2007, pp. 105–113.

[3] N. I. of Standards and Technology, "National software reference library," http://www.nsrl.nist.gov/, August 2003.

[4] N. Harbour, "DCFLDD," http://dcfldd.sourceforge.net/, March 2002.

[5] A. Tridgell, "Spamsum readme," http://samba.org/ftp/unpacked/junkcode/spamsum/README, 2000.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[7] D. Grossman and O. Frieder, *Information Retrieval*, 2nd ed. Springer, 2004.

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 1, pp. 1–16, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2007.9

[9] Oracle, "Oracle Outside In Technology," http://www.oracle.com/technology/products/content-management/oit/oit_all.html.

[10] "Lucene homepage," Available at http://lucene.apache.org/.

[11] M. F. Porter, "An Algorithm for Suffix Stripping," *Readings in Information Retrieval*, pp. 313–316, 1997.

[12] A. Chowdhury, O. Frieder, D. Grossman, McCabe, and M. Catherine, "Collection statistics for fast duplicate document detection," *ACM Transactions on Information Systems*, vol. 20, no. 2, pp. 171–191, 2002.

[13] V. Roussev, "An Evaluation of Forensic Similarity Hashes." in *In Proceedings of the Eleventh Annual DFRWS Conference*, New Orleans, LA., August 2011, pp. 34–41.

[14] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing Science to Digital Forensics with Standardized Forensic Corpora," in *In Proceedings of the Nineth Annual DFRWS Conference*, Montreal, Canada, August 2009.

[15] B. Klimt and Y. Yang, "Introducing the Enron Corpus," in *First Conference on Email and Anti-spam (CEAS)*, 2004.

[16] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proceedings of the 16th International Conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 131–140.