Finding similar document

Ettore Celozzi
E-mail address
ettore.celozzi@stud.unifi.it

Luca Ciabini
E-mail address
luca.ciabini1@stud.unifi.it

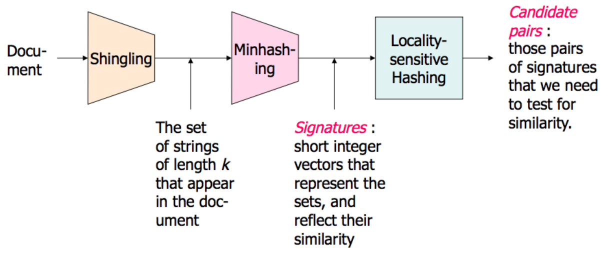
Abstract

Plagiarism in scientific articles and in documents in general is a difficult problem to deal with both because the scope of this phenomenon is increasing but also because of the number of documents to examine is massive and difficult to handle.
Copyright infringement occurs when the author of a new article uses an important part of previously published articles. Clearly trying to recognize such behaviors just comparing line by line and word by word each documents in a brute force way is simply impossible in terms of computational cost.
In this paper we present an implementation of a series of techniques which aim to identify pairs of documents from a bunch of PDF files that are candidates of copyright infringement.
The results obtained using this algorithm however are subject to false positives.

1. Introduction

The problem of finding PDF that have an important part of text in common is hard and what we want to do is check if two documents are similar, but without, of course, comparing every line.
So the idea in steps is to:

- preprocess every document extracting only the text and deleting every stop word, image, math expression etc.;
- create for each document a signature using the min Hash procedure that depends on the text it contains;
- get the candidate pairs using the Locality Sensitive Hashing from starting from the signature matrix.


Docu-ment
Shingling
Minhash-ing
Locality-sensitive Hashing
Candidate pairs : those pairs of signatures that we need to test for similarity.
The set of strings of length k that appear in the doc-ument
Signatures : short integer vectors that represent the sets, and reflect their similarity

2. Use cases

There are three main use cases that this program aim to satisfy:

the first use case is finding the documents that are most similar inside a bunch of PDF; similarly in the second use case we want to find the most similar pairs of documents between two given groups of PDFs: detect which PDF of the first group is similar to one (or more) of the second group.
These two scenarios are quite slow especially because of the preprocessing needed for each PDF.
Finally in the last use case, given a database of files and a single PDF that we want to check, our software discovers which PDF or PDFs in the database are most similar to the input one; In this case the procedure is pretty fast since the databases has already been processed.

3. Pdf mining

PDFs are not such as other documents, they are more like a graphic representation, in fact is not possible to retrieve a logical structure that describes the text position. To recover something similar to a structure we use PDFminer library. Thanks to this we are able to perform two operations on PDFs: **tokenization** and **shingling**.
Both methods return a list of elements: in the first case the list items are the PDF's words, in the second the list items are the shingles. A shingle is a subset of a word with a fixed

length k. For instance if we have the word "parsing" and $k = 4$ the world will became the list: [pars,arsi,rsin,sing].

3.1. Tokenization

To perform the **tokenization** we used three functions:

- `LAParams()` function is used to build the tree structure of each page and set the parameters for analysis whit that informations.
- `TextConverter()` is used to transform PDF files into other text formats (such as HTML).
- `PDFPageInterpreter()` function is used to process the page contents and return the text of the PDF.

A problem that can arise is that word can be hyphenated, in this case the library yield two separate words, so before accepting the output we check at the end of each word if there is a hyphen, if there is, we glue the first part with the other part of the word. Finally before saving the result in a txt file we delete the useless part of the text, such as stop words, punctuations and maths expressions:

- The stop words are taken from the Natural Language Toolkit (NLTK)
- Punctuations symbols are taken from a list that include "(" , "@" , "," , ":" etc.
- Math expressions symbols are taken from:
  - numbers;
  - all the greek aphabet;
  - symbols like "$<$" , "$! =$" , "$\geq$" etc.

3.2. Shingling

Since to generate the shingles is necessary to execute the same preventive operation on PDF's word, we retrieve this from the token file. A sliding windows of length k slides from the begin to the end of the line, storing at each iteration the window content (In our case $k = 10$).
The shingle are necessary to build the sparse matrix which has the list of documents on the columns and the **hash** of shingle on the rows. The hashing is performed through this formula:

$$h = \sum_{i=0}^{k} alphabet[s[i]] * 31^i \; mod(m)$$

where $s[i]$ is the shingle's letter, $alphabet[s[i]]$ return a numeric value corresponding to the alphabet character(a=1,b=2,...) and $m = 1000003$.

4. MinHashing

The $minHashing$ phase take a sparse matrix (we will go deeper in the sparse matrix implementation in the next paragraph) and the number of permutations we want to make as input. Then we perform a permutation of the rows of the matrix using a random Hash function in order to get for each loop iteration a different index permutation.
Once the permutation is done, for each column of the matrix (meaning for each document), we search for the first row index that has a one; when it is found the row index is written in a list witch will be part of the $minHash$ matrix.
Is also important to notice that at the end of this phase the $minHash$ matrix and all the (a, b) pairs used for the hash will be stored in the memory for reasons that will be clear in the next rows.
The i-th column of the $minHash$ matrix will be the signature of the i-th document.
But this phase can change in the case we want to compare a single document to a group of, already evaluated, documents. In this case in fact, we only want to "minHash" a single document, and of course we want to use the same (a, b) pairs used to "minHash" the rest of the documents. So the function takes also the (a, b) list and will use them for the hash function.

4.1. MinHash hash

The hash function for the "minHash" phase takes a and b that can either be randomly created or taken from the memory, the value x to be hashed, a prime number p and the number of shingles (meaning the row of the sparse matrix) n.
The algorithm perform the operation $h_{a,b}(x) = ((a * x + b) \; mod(p)) \; mod(n)$. Using the $mod(n)$ we can be sure that the hash table will make legal permutation of the rows. As p we use the number 10000013

5. Sparse Matrix

The result of the shingling phase is a very sparse matrix. To well represent it and avoid waste of memory we implemented the class $sparseMatrix$.
It uses a dictionary that has as keys the index of the rows that contains at least a 1. The value of a dictionary item is a list that contains the index of the columns that has a 1 in correspondence of the row indicated by the key.
The class also offers functions like:

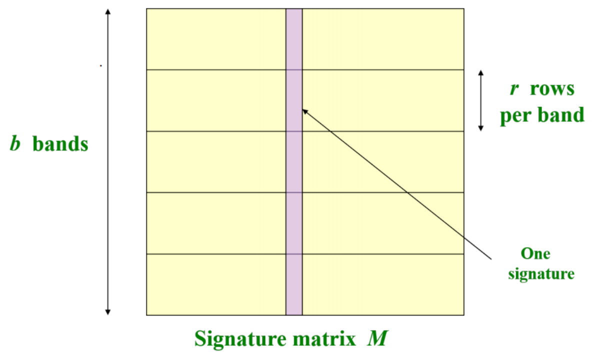
- `addValue( rowIndex, columnIndex):` that add a new column index in the correspondig key and also add a new key (rowIndex) if it does not already exists.
- `addValueWithCheck(rowIndex, columnIndex):` that add the a column index

in corrispondence of the key ”columnIndex” only if it is not already present in the list.

- `isEmpty(rowIndex)`: that return true if the passed row index is not present in the dictionary
- `getColumns(rowIndex)`: that return the column index list of a given row Index
- `getKeys()`: that return all the keys of the dictionary

6. Locality Sensitive Hashing

The LSH phase takes three parameters: $minHashes$ which is the $minHash$ matrix, $numberOfBands$ which is the number of sets in which we want to split each document signature and $numberOfDocuments$.
The procedure divides the $minHash$ matrix in pieces based on the number of bands we want.



After that the band is hashed with the function $listHash$ (that will be explained in the next subparagraph) and if two (or more) bands with the same index (the $i-th$ band of the $j-th$ document with the $i-th$ band of the $z-th$ document) collide in the hash table then the documents whom those bands belongs are candidate: meaning they probably have a similarity.

6.1. List Hash

The $listHash$ function takes a band and a value m (chosen to be higher of the $numberOfDocuments$ and not to be a power of two) and perform the following operations: first of all it convert the list into a number simply using positional notation (in this way the list [1,4,2] becomes the number 142) and then we proceed with $val\ mod(m)$.

7. Experiments and results

7.1. Machine technical specification

- Processor: Intel(R) Core(TM) i7-4750HQ CPU @ 2.00GHz, 1995 MHz, 4 Core(s), 8 Logical Processor(s)
- Ram: 8GB (7,89GB available), DDR3, 1600MHz
- Disk: SSD GOODRAM 256GB, 750MB/S

7.2. Experiments

For the experiments we decided to proceed in this way:

- First we choose randomly a document from the database end create five copies of it. Each copy will represent a different similarity:
  - 80% similar to the original (meaning that the 80% of the rows are the same)
  - 60% similar to the original (meaning that the 60% of the rows are the same)
  - Analogue thing for 50%, 40% and 20%.
- For each copy we loop in the rows and with a probability equals to the similarity we decide if we want to change the row (For instance in the 80% similar document for each row there is a probability of 20% that it will be changed).
- If the row has to be changed then we proceed choosing randomly a document inside the database and a row inside the document (again, randomly). Then we swap the tokens of that random row with the tokens in the copy.
- Finally we check if our algorithm is able to retrieve the original document starting from the copied and modified one.

Using a different number for bands and row per band we iterate this procedure for 50 times (50 times for each different pair $(bands, rowsPerBand)$). In particular we use four different pairs of $(bands, rowsPerBand)$: $(50, 2)$, $(25, 4)$, $(20, 5)$ and $(10, 10)$.
The number of bands and the rows per band are key values because they affect the probability of finding a document with certain similarity. The formula $1-(1-t^r)^b$ where t is the similarity, r is the rows per band number and b is the number of bands gives us the probability that two documents with a certain similarity will be found by the algorithm (see table 1).

Clearly if we want to find documents with a lower percentage of similarity a large number of documents will be found because they will need a lower number of similar lines.

7.3. Results

For the experiments we used a database of 393 documents; in particular scientific papers coming from two different conferences: one in the 2017 and one in the 2018.

Table 1. Ideal probability table

Similarity	r=2
	b=50
		r=4
		b=25
			r=5
			b=20
				r=10
				b=10

0,8	1	0,99	0,99	0,67
0,6	1	0,96	0,80	0,05
0,5	0,99	0,80	0,47	0,01
0,4	0,99	0,47	0,18	0
0,2	0,87	0,03	0	0

Table 2. Real probability table with 50 test iterations

Similarity	r=2
	b=50
		r=4
		b=25
			r=5
			b=20
				r=10
				b=10

0,8	1	0,96	0,81	0,1
0,6	0,98	0,68	0,42	0
0,5	0,94	0,34	0,11	0
0,4	0,86	0,1	0,06	0
0,2	0,62	0,06	0	0

The results obtained executing 50 tests for each different pairs of $(bands, rowsPerBand)$ are reported in table 2.
The total disk space used for the PDF database is over 300MB and for the shingles is 102MB while for the memorization of the minHash matrix only 120KB are occupied.
For what concern the execution times the program can be divided in three parts:

- The most time consuming part of the program is the tokenization and the creation of all the shingles that takes 36 minutes in our machine (fortunately once they are created there is no need to execute this part again).
- The creation of the minHash matrix also require a lot of time: 4 minutes to perform 100 permutations (as above even the minHash matrix is stored in the disk so this part of the code will not be executed again).
- Finally there is the LSH phase that takes only few second depending on the number of the bands.

In the end the total time including also the tests with 50 iterations takes more than 3 hours.

8. Conclusion

This software analyze *substantial similarity* of documents. The purpose is to detect if, some sections or portions of text, are copied from another document without referring to this.
To discover similarity is necessary to confront PDFs. Comparison can be performed between one PDF and a bunch of these, or among bunches belonging to different context.
The ideal value showed in table 1 are an upper bound of probability to identify similarity, our results are lower but proportionate to them. We expect that as the number of tests grow the real probability values become closer to ideal ones.