Fingerprinting jar files

Sasha Mahdavi-Hezavehi Seyyed Mohammad Hadi Sajjadpour

November 2010

COMP 5900

School of Computer Science,

Carleton University

Ottawa, ON

Abstract

As the size of software grows, it becomes harder and harder to manage all of them and detect plagiarism. Hence, we introduce a method of creating identifiers, which we call fingerprints, for jar files to identify jar files. These fingerprints will then have enough information for us to compare them against other jar files and give us an approximate percentage of how similar two jar files are. We take into account a few different components of a jar file and mix them together to create a fingerprint. One of the components we use are text files that are used in documents. For this we use winnowing. Winnowing is a very effective algorithm for document detection. We also look at file extensions, file paths, file name and also file sizes.

Introduction

Problem

Given that the number of software and software developers and work produced by them is on the rise, software, just like any other industry, has to face the challenges of original work. As more and more software and documents are being produced, the need for work identification grows. Hence in this paper, we will discuss our solution to overcome this problem via creating fingerprints from jar files. Although every piece of software will not fall into this category, however, we believe that it is a step forward to solving the greater problem.

Motivation

Given that companies and individuals spend a decent amount of time and money to produce software, a work stolen could potentially be a big blow to companies and individuals. On top of the time/money factor, credentials and reputation is also stolen. It is not trivial to prove that a company or individual has stolen a software or document without solid proof.

Goals

The goal is to build a software for proving any plagiarism and copying. As a result, this could be lead to reducing software plagiarism and copying, hence developers get the correct credentials and potentially, they do not lose extra money. It is worth mentioning that companies and individuals must pay some attention to licensing, hence another of our goals is to help in enforcing these licenses.

One of our goals is to take different components of jar files, create separate fingerprint from each component of a jar file, then combine all these prints together and create a fingerprint for a jar file. We also let our software be open to other components that we were not able to implement.

Objective

We want to make these identifiers that give away a lot of information on jar files. This information should be enough to compare them against other jar files and while comparing, get a good quantitative value as of how similar the jar files are. We call these identifiers, fingerprints. These so called fingerprints should only be a string and nothing more. Any fingerprint is encoded by five different components. These coded components could be encoded and disintegrated for comparison with other fingerprint's components' elements. However, we leave the room open for other components.

Outline

We will first give a little background on this topic, and then we will discuss our approach and design decisions. After that we will move on to how our implementation reacts in different scenarios. We wrap it up with our conclusion and by giving some ideas on how making what me made more effective.

Background

A good fingerprint should reveal sufficient information such that two similar jar files are detected. We have not found many articles or papers regarding specifically fingerprinting jar files. However, there are different algorithms for different elements of the jar file that have been previously researched on, namely document fingerprinting. We will point out to a few of these algorithms for different components of a jar file.

Although the goal of document plagiarism is different from jar files, it helps quite a bit as jar files do have documents in them.

In general a good copy detection algorithm for documents should have the following properties, taken from [2]:

1. *Whitespace insensitivity:* In matching text files, matches should be unaffected by such things as extra whitespace, capitalization, punctuation, etc. In other domains the notion of what strings should be equal is different—for example, in matching software text it is desirable to make matching insensitive
to variable names[2].

2. *Noise suppression* Discovering short matches, such as the fact that the word *the* appears in two different documents, is uninteresting. Any match must be large enough to imply that the material has been copied and is not simply a common word or idiom of the language in which documents are
written[2].

3. *Position independence* Coarse-grained permutation of the contents of a document (e.g., scrambling the order of paragraphs) should not affect the set of discovered matches. Adding to a document should not affect the set of matches in the original portion of the new document. Removing part of a document
should not affect the set of matches in the portion that remains[2].

It is easy to achieve property 1, but not all algorithms achieve 2 and 3. As you will see in our approach, we use the algorithm in [2] called winnowing which satisfies all three. Schemes based on fingerprinting k-grams satisfy property 2[2]. We will discuss what k-grams are in our winnowing section.

There are also algorithms that compare source code, for example, baker's algorithm. For two different source codes, it recognizes their variables and renames them, hence comparing becomes much easier. They use a program called *dup* and locate instances of duplication or near duplication in a software system. Testing their program on millions of lines of code, they claim that their algorithm is both fast and effective [3].

However, fingerprinting jar files is not limited to text based documents. Another work we found, was that of [1], where they give weights to different components of a jar file, namely file names. We make use of their style in [1].

Approach

Our Design

For designing the fingerprint, first, we tried to categorize different elements of the jar files into separate components. As we know, a jar file includes class files plus different text based files for documentation or description of an application/software. On the one hand, it is possible to generate a fingerprint by just decompiling the class files and using different techniques for finding some equivalencies between java files which can been generated by decompiling them. On the other hand, it is also possible to create a fingerprint by using other characteristics of a jar file. Separating all text files in a jar file, and using their path, size, extensions, and contents to build a fingerprint. We use the latter approach. Note that we also take all the mentioned properties of all other non text based files, except for their content.

This approach tries to separate and classify all files which exist in a jar file. Then make a fingerprint with these files which includes the properties related to paths, sizes, extensions, names, and text content of text based files(.txt, .html...). The result which the software produces is a composition of the information related to all mentioned properties. Also, our software is capable of giving an approximate percentage of how close different jar files are. For finding the similarity of the components of a fingerprint with another, our technique uses weights for each property. We store each jar file produced in a database via a text file.

Different jar files might have different kinds of files and the number of these files might be different. Hence, it is possible to change the weights of each property to reach a more suitable and logical fingerprint. In details, by changing the values of weights, it is possible to concentrate

on special parts of the jar files for comparing with another jar file. However, this has some drawbacks as we will not be able to keep in a uniform database, and hence avoid it in our implementation except for one case which we will see later on in this paper.

Our fingerprint calculation is similar to the scheme in [1], however with some differences. Our fingerprint has information regarding the following information:

1) Text based files (.html, .xml, .rtf and .txt) by using winnowing with 40% weight.
2) File extensions with a weight of 24%.
3) Size of files with 15% weight.
4) File Names with 4% weight.
5) File Paths with a weight of 17%.

Top Level Implementation Details

Different classes have been created for building this software. FileExtensionFingerprint, FileNameFingerprint, FileSizeFingerprint, FilePathFingerprint, and WinnowingFingerPrint are classes for generating the components of each property of a fingerprint. Also, there are some helper classes which use these classes for generating a fingerprint or comparing two fingerprints and for reading fingerprints from files or writing them to files, for merging different properties and generating a unique identifier, or for decomposing a fingerprint or identifier and comparing each part of a fingerprint with the same part of another fingerprint. The formula for calculating the result of the comparison uses the average of both properties.

FileReader and FileStructure are classes for managing the content of jar files and text files. They use some data structures for holding some properties of files, for example, name of a file, and extension of a file, path, and size of a file plus the content of a file. WriteToFile is a class for writing contents in a file. If it is necessary, it will create a file. The class FingerPrintCoordinator coordinate different classes for producing a unique fingerprint for a jar file. This class uses those five classes which are the generator of fingerprint properties plus some operation for coding these components and locating them in a string with special tokens.

Special tokens are used to make distinction between different components of a fingerprint. The software is able to decode all these properties for making a comparison between two related properties from different fingerprints.

Winnowing on Text based files

In this part, we take text files and perform an algorithm taken from [2], which is called winnowing. We thought that instead of reading content from .class files, which will take a considerable amount of time, we will check text files instead. We gave this 40% of our fingerprint weight, as it is the only content based information we can get. Note that we avoided giving it more than 50%, as if a plagiarizer eliminates all text based files, we still give a percentage above 50% for the difference.

Winnowing

Winnowing is a method of creating fingerprints for files. We will first discuss how the algorithm works, then talk about its properties and our method of implementation.

Before we get started, here are some definitions that we will be using throughout the explanation of winnowing.

K-gram: A K-gram is a contiguous substring of length k[2].

Window of size w: W consecutive hashes of k-grams in a document, where w is a parameter set by the user.

The algorithm does the following, let us take a sample from[2]:

(a) A do run run run, a do run run.

(b) Eliminate all undesired features. In this case it will be:

Adorunrunruadorunrun

(c) Generate a sequence of k-grams. This means that for each character obtained in b, take that character and the k-1 following characters. In this example we will get: adoru dorun irunr runru unrun nrunr runru unrun nruna runad unado nador adoru dorun orunr runru unrun

(d) Now hash the terms found in (c). In our example, it will (given a simple hash) give us : 77 74 42 17 98 50 17 98 8 88 67 39 77 73 42 17 98

(e) Take windows of length w. In this case let w = 4, hence we get:

(77,74, 42, **17**) (74,42,17,98)(42,17,98,50)(17,98,50,**17**)(98,50,17,98)(50,17,98,**8**) (17,98,8,88)

(98,8,88,67) (8,88,67,39)(88,67,**39**,77) (67,39,77,74) (39,77,74,42)(77,74,42,**17**) (74,42,17,98)

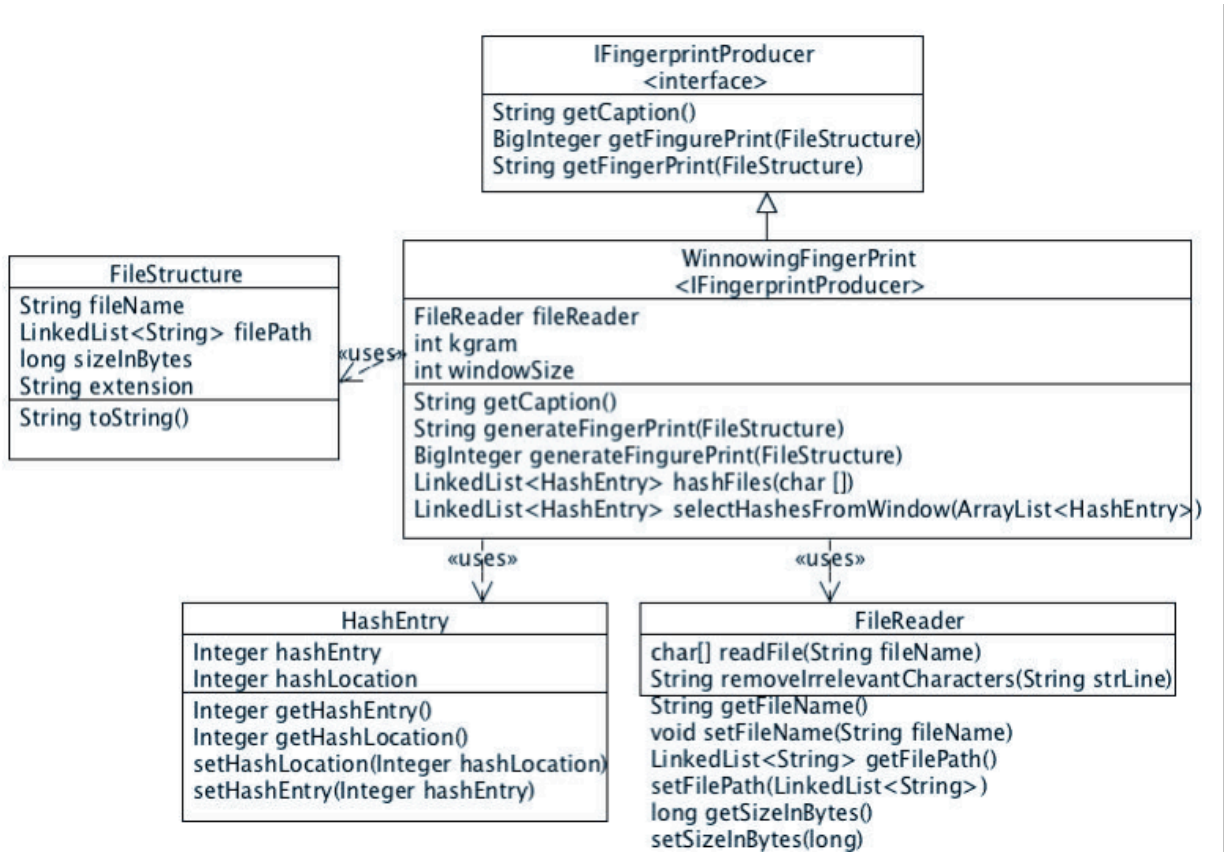
(f) In each window, select the minimum hash value. If there is more than one minimum in a given window, select the rightmost. Note that if a hash is selected in one window and it still remains the smallest in another windows (note the same hash, not the same value but from a different window) then it is still selected. In this example we get:

17 17 8 39 17

(g) Store the fingerprints with their location:

[17,3] , [17,6] , [8,8], [39,11], [17,15]

Our Implementation of Winnowing



In our implementation, we first look for .xml, .html, .txt and .rtf files. Then we follow the following steps:

1) Throw everything in the file into a string

2) Take away all the spaces of that String using Java's Pattern and Matches classes

3) Convert the String into an array of char

4) Hash each array k characters in a row as seen above, and then we throw each into a linked list of HashEntries(a helper class)

5) Convert the linked list to an array list. Note that the above step was done to preserve the order of the list

6) Finally select hash entries with lowest value in windows of size w

7) Along with their location, send them to the fingerprint coordinator

Winnowing Properties Implementation

As you have probably observed so far, winnowing is not a very complicated algorithm. However simple it is, it maintains the following properties, which makes winnowing very effective for copy detection on .html pages, and in our case any text based file. These are the same properties we explained in the background, here we just point out how they are achieved in our implementation as it has been in claimed by their paper in [2]

1) *Whitespace Insensitivity:* In matching text files/creating fingerprints, fingerprints should be made independent of capitalization, punctuation, white spaces.. We implement this by replacing all spaces by no space, removing all punctuations, dashes and any other character we found irrelevant.

2) *Noise Suppression:* Words such as, "the", "of", " an" etc must not affect the fingerprints. We implement this by choosing a k-gram of size greater than 4 or 5. An alternative would be to eliminate all such words from the text as we did in 1 with white spaces..

3) *Position Independence:* Permutation of words or scrambling the order of paragraphs should not affect the fingerprints. This is achieved by the nature of the algorithm, by taking the minimum in a window of size w.

File Extensions

Given what usually is put in a jar file, .class files, it is not very common to change the extension of files names in a jar file, nor is it practical as it well change the functionality of softwares. Hence we added this to our overall fingerprint. We gave it a high percentage of 24%. When we read the jar file from disk and un jar it, we also store the extensions of each file. Hence we keep track of how many files are in each extension by adding the extension as our key of a hash map, which then links them to a linked list of files.

The catch here is that, text based files, such as .txt, .xml … could be changed without major damage being done to the actual content of a file or change of functionally. However, we have already outsmarted the plagiarizers by performing winnowing on text based files. Even if two text based files are merged, our winnowing algorithm still recognizes the words.

Size of Files

In a jar file, once again, the main components are .class files. Even if variable names are changed, the size of .class files aren't affected a lot. Changing file size could also have impact functionality.

Hence we file sizes reveal some information whether two works are the same or not. However, we do not give a lot of weight to this, as file size in essence does not by itself give a lot of information about a jar file.

File Names

Changing file names could potentially bring a lot of structural changes to the content of a jar file, and would require a lot of changes to documents etc. For example if .html names are modified, then all links to that html page must be modified, hence it will require a lot of changing around. To a lesser extent, it will also require changing names/classes in the source code, however with development environments such as eclipse, this is task is easily done.

Overall, since changing files names is a bit easier than changing extensions, size.. we do not give it as much weight as the other components, hence we give it only 4%.

File Path

Given a well-designed software that is compressed in a jar file, it will be extremely difficult to change the path of files, as this will require a lot of changes to the actual software.

We compare the file path of two jar files, by counting how many directories it takes to get to any given file. Note that we do not check for the name of the path, as in for example " C: \comp5900\project\packageA\a.class", instead here the .class file goes through

C:\ -> Comp5900→ project -> package -> Destination, hence the length of path here is 4.

Fingerprint Production and comparison

In details, the class FingerPrintCoordinator calls the other classes FileExtensionFingerPrint, FileNameFingerPrint, FilePathFingerPrint, FileSizeFingerPrint, and WinnowingFingurePrint. The FingerPrintCoordinator also takes all text files with rtf, html, xml, and txt extensions from the jar file that have been previously written on to the hard disk when un jarring into the program's memory for winnowing operation. Then, it will delete those files from the hard disk for preventing the redundancy of unused information. The FingerPrintCoordinator will also produce a string as a unique identifier or the fingerprint of a jar file by locating and concatenating all the sub components of the fingerprint which are produced by calling each related class, which in this case are the above mentioned. For example calling FilePathFingerPrint will create a string that will tell us about the file paths in the jar file.

Furthermore, the FingerPrintCoordinator concatenates each of the strings produced by calling the classes mentioned above, delimiting them by a *. We use the sign * because it is not possible to use this sign in the file names or file extensions. Note that it is possible to use the star character in text files but the winnowing comparison compares all contents sequentially. Thus, it will not create any problems in case of separating them with this sign because all members have been hashed into numbers. Then, the class FingerPrintCoordinator puts all these components between |nameOfTheComponent| as a first token for identifying this component and ?nameOfTheComponent? as the termination of this component in the string. For example, for distinguishing winnowing component in the string of a fingerprint two tokens |WINNOWNIG| and ?WINNOWING? are placed in the beginning and end of this component.

The fingerprint is produced in a way that we can decompose and disintegrate all members for comparison with another fingerprint. This is achieved by using the class FingerPrintCompare. First, this class decomposes the fingerprint to its sub components. Then, it disintegrates all components to its members. Afterwards, FingerPrintCompare compares two finger prints. Hence, after each comparison, there would be a number of matching which shows how many members of each component is equal to the same component of the other fingerprint. For calculating the final result of comparison, both jar files have equal weight in the final formula.

Validation

Our application was verified by running several times and by choosing different several jar files with different sizes and different content. From the general point of view, there are two main

different types of files for our solution. Jar files that contain text based files and jar files which do not contain text contents or have very little of it. The host computer, in using our application, should allow reading/writing to the current directory, as we need to write the files on the hard disk to read them and figure out their size, we delete once we are done.

Jar files with text contents

This application can recognize the similarity of two different jar files properly by comparing their finger prints. However, there are some drawbacks in the exact accuracy of this application for finding the very precise result. Although we compare all text files, however, changing the type of any text file to the other file could impose changing the header of the file or even generating a new header for a text file. Thus, the accuracy can become lower. For example, changing a normal text file with the extension of txt to a html file could impose different headers and footers or tags. As a result, the algorithm in our approach will produce the final comparison result with a bit lower accuracy.

Also, the application does not take too much time for running. By examining different files with different sizes, the running time was up to 20 seconds. However, there is no guarantee for this period of time. A jar file with a extremely large text content may take more time for processing and comparing.

Jar files without text content

Our approach has five different factors for any fingerprint. By eliminating the text, one of the most important properties of the comparison with the greatest weight has been omitted. Thus, the application may not answer even with a low level of accuracy. In this case, it is better to change the formula by eliminating the winnowing component from this formula. The other four components could make some estimation about the comparison of two fingerprints of jar files.

As a result, the result of the fingerprints can be produced by extensions of files, file sizes, file paths, and the name of the files. In this situation, the file paths and size could be much more important than other components. In particular, it is more complicated to change the size and the paths of all classes when someone wants to use a big open source code in another program without using open source licenses. Thus, it is possible to pay more weight on these two characteristics.

Conclusion

Goal and Contributions

This project's goal is to find an effective and pragmatic match between two fingerprints which is produced by our application, each fingerprint containing information on different components of

a jar file. In this case, the fingerprint producer software especially concentrates on the available text based documentation in jar files when these kinds of documentation exist. Thus, using the winnowing algorithm tightly dependent on the documentation in jar files. This approach works almost properly in a reasonable period of time. However, if jar files do not have any documentation or the documentations are very short, it would be better to assign less weight to the winnowing algorithm or even eliminate this component when there is no documentation in any jar file for comparison.

Future work

This approach does not use the content of class files for making fingerprints. Thus, it would be possible to decompile the class files and produce java files. Then we can add the content of java files as another component of the jar file to our fingerprint. There is an algorithm which changes the name of all variables in any java files to some special tokens and holds the general structure of the code. Then, it is possible to have a comparison between java codes of both jar files based on their general structure without being concerned about the effects of the name of different variables in the source code. This algorithm is called Baker's algorithm. By using Baker's algorithm, the final result of fingerprint comparison could be more precise than this existing result. However, note that this might add to the time it takes to produce fingerprints for very large jar files.

References

[1] Cate Hutson, Fingerprinting Jar Files using Winnowing , Ottawa

[2] Schleimer, S, Wilkerson, D. D, Aiken, A. Winnowing: Local Algorithms for Document Fingerprinting. SIGMOD 2003

[3] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In L. Wills, P. Newcomb, and E. Chikofsky, editors, *Second Working Conference on Reverse Engineering*,Los Alamitos,California, 1995