Automatic identification of client-side JavaScript libraries in web applications

Hanyang Cao, Yuxian Peng, Jing Jiang, Jean-Rémy Falleri, Xavier Blanc

Automatic Identification of Client-Side JavaScript Libraries in Web Applications

Hanyang Cao
University of Bordeaux
LaBRI, UMR 5800
F-33400 Talence, France
cao.hanyang@labri.fr

Yuxian Peng
Beihang University
Beijing, China
yuxianpeng@buaa.edu.cn

Jing Jiang
Beihang University
Beijing, China
jiangjing@buaa.edu.cn

Jean-Rémy Falleri
University of Bordeaux
LaBRI, UMR 5800
F-33400 Talence, France
falleri@labri.fr

Xavier Blanc
University of Bordeaux
LaBRI, UMR 5800
F-33400 Talence, France
xblanc@labri.fr

ABSTRACT

Modern web applications often use JavaScript libraries, such as JQuey or Google Analytics for example, that make the development easier, cheaper and with a better quality. Choosing the right library to use is however very difficult as there are many competing libraries with many different versions. To help developers in this difficult choice, popularity indicators that pinpoint which applications use which libraries are very useful. Building such indicators is however challenging as popular web applications usually don't make their source code available. In this paper, we address this challenge with an approach that automatically browses web applications to retrieve the client-side JavaScript libraries they use. By applying this approach on the most famous websites, we then present the trends we observed, and the recommendations that can be provided.

CCS Concepts

•**Information systems** → **Web applications;**

Keywords

Library migration; JavaScript; Web Application

1. INTRODUCTION

Software projects often use third party libraries. With Java projects for example, 70% of the projects use at least 4 third party libraries, and 10% use more than 10 libraries [1]. For web applications, the growing interest for library management systems, such as NPM with NodeJS, shows the importance of this topic[2]. Further, it is well known that third party libraries such as *jQuery* are almost used by all web applications [3].

http://dx.doi.org/10.1145/3019612.3019845

Using a third party library provides many benefits as reusing high quality code prevents errors and speeds up productivity[4]. However, it comes with the main problem of the choice of the best library from a software development perspective [5]. Indeed, there are so many libraries with so many versions that it becomes too complex for a developer to choose which one to include in a software project. This is even more difficult for JavaScript client-side libraries, because of the popularity of JavaScript,[1], the outstanding pace of JavaScript libraries production, and the fact that web applications are doomed to evolve at the Internet speed to be used and not to become deprecated[6].

As an example, let us consider that a developer wants to include the famous *jQuery* library within its project. By browsing the web, he will then notice that there are two major versions for *jQuery* (1.x.x and 2.x.x) and several minor versions. If he now tries to understand which is the best version, he will end up into StackOverflow with several different and sometime opposite answers[2]. As another example, if a developer wants to use a MVC framework such as *AngularJS*, *Backbone* or *Polymer*, and asks to the web for the best one, he will then get many different answers, each saying that one of these framework is the best. As a consequence, he won't be able to decide the one to choose.

To help developers to chose the right library to include, many existing research approaches aim to provide popularity indicators on libraries[7, 8, 5, 9, 10, 11]. These approaches observe large sets of applications with the objective to detect the most popular libraries. However, all these approaches are based on the analysis of the development artifacts (source code files or deployment descriptors), and therefore rely on the observation of open source projects that

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Here is the answer at the time of the 20th of August 2015, http://stackoverflow.com/questions/22289583/what-version-of-jquery-should-i-act ually-use

make such artifacts available. This is however not possible with web applications as most of the famous web applications are closed-source applications, and their development artifacts are not available at all. The development artifacts of a commercial web application are only available on-line by browsing it through its root url.

In this paper, we overcome this difficulty and exhibit which are the popular client-side JavaScript libraries. Our proposal performs an online observation of popular web applications with the underlying assumption that the libraries they use are most probably the ones that should be used. The main difficulty is therefore to identify which are the libraries they use just by browsing them. To get significant results we decide to observe the Alexa global top 100 websites[3]. Our approach then browses these web applications and recognizes the client-side JavaScript libraries they use. Based on such observations we can then output trends and give recommendations.

We make the following contributions:

- We provide an automatic and efficient approach that browses web applications and detects their used client-side JavaScript libraries (see Section 2). Our approach uses both syntactical and dynamical analysis of the online resources of the web applications and returns high precision results (see Section 3 and Section 4).
- By applying our approach on the 100 most popular websites, we provide statistics of the use of JavaScript libraries. Such statistics confirm the fact that libraries are largely used in web applications (see Section 5).
- We then present how our observations can be used to provide trends and recommendations. As an illustration, we present our observation by focusing on *jQuery* and *AngularJS*, *Backbone* and related libraries (Section 5).

2. METHODOLOGY

This section first provides definitions for web applications and client-side libraries. It then presents three strategies that can be used to detect the client-side libraries used by a web application. Then it presents our global approach that uses these three strategies.

2.1 Definitions

First, we consider that a web application is a pair (w, url) where w is the name of the web application and *url* is its root url. For instance, $(\text{DropBox}, \text{www.dropbox.com})$ is a web application. Second, we consider that a JavaScript (JS) library is a pair (l, V_l) where l is the name of the library and V_l its corresponding ordered set of versions. For instance $(\text{JQuery}, \{1.1, 1.2, 1.3\})$ is a library.

As we briefly presented in Section 1, our objective is to automatically identify the client-side JS libraries used by web applications. Table 1 shows a tiny example of the result we want to obtain, called the *library usage matrix*. The library usage matrix contains a set of web application names as rows, and a set of JS library names as columns. The goal of this table is to indicate which library is used by which web application, and to give the corresponding version. Therefore the value of a cell corresponding to an application w and a library l belongs to the set $V_l \cup \{?, \perp\}$, where V_l is the set of all versions of library l, as previously defined, and where the symbol ? means that w is using an unknown version of l, and where $\perp$ means that the application is not using the library. Table 1 states for instance that *DropBox* uses only the version 1.3.1 of *jQuery*, *Alipay* uses an unknown version of *AngularJS* and *Pixnet* does not use any library. We will use the library usage matrix to compute which libraries are popular, and which particular versions of the libraries are popular, with the main objective to exhibit trends and to give some recommendations.

[3] http://www.alexa.com/topsites

Table 1: Library usage matrix.

	JavaScript Libraries	
Web applications	jQuery	AngularJS
`Alipay`	$\perp$	?
`Dropbox`	1.3.1	$\perp$
`Pixnet`	$\perp$	$\perp$

2.2 Recognition Strategies

Filling the library usage matrix requires to browse web applications and to recognize which libraries they use. It is done by using several *recognition strategies* that are programs that take as an input a given web application, and produce as an output a row of the library usage matrix. The remainder of this section presents the three different recognition strategies we propose that, when combined, provide good results (see Section 4).

2.2.1 Comment Strategy

The idea behind the *Comment Strategy* is to search for names of libraries in the header comment of the JS files used by a web application. This strategy is quite efficient since library files often contain this information. For instance, Figure 1 shows the three first lines of a Modernizr JS library file. We clearly see in this header that this file cites the name of the library (Modernizr) and its version (2.8.3).

To fetch this information, the comment strategy begins by browsing the root URL of the web application. Browsing this URL returns all the webpage content, including a set of linked resources. We retain from this set of resources only the ones that correspond to JS files, by using the *content-type* information. We then use several regular expressions to extract the library name and version from the comments, as shown in Figure 2. More precisely, one regular expression is generated for each sought library by substituting the variable *name* by the name of the sought library. These regular expressions are then all checked, in global mode, against a JS file until one matches, ignoring the case. This regular ex-

```
/*!
 * Modernizr v2.8.3
 * www.modernizr.com
```

Figure 1: The header of a Modernizr JS library file.

pression is robust enough to match comments such as *jQuery JavaScript Library v1.11.3* or *Modernizr v2.8.3*.

The comment strategy has the main advantage to be very efficient since executing a regular expression is quite fast. Furthermore, this strategy can be easily extended to support the search of new libraries just by adding new library names. Unfortunately, several web applications remove all the origin comments of a library file, which prevent this strategy from working. The detailed performance results are presented in the Section 4.

2.2.2 File Matching Strategy

The intuition behind the *File Matching Strategy* is to check if a JS file used by the web application is similar to a file that is known to be a JS library. Similarly to the first strategy, this strategy starts by retrieving all the JS files used by the web application. It also uses a so-called *knowledge base* which contains the files of all versions of all JS libraries. This knowledge base is large: it contains more than 2000 files in our current implementation.

Asking for an extensive comparison with all the files of the knowledge base would take too long. To perform such a comparison in a reasonable time, we use a two-step processIn the first step, we use simhashes, which are short hashcodes computed on a large text, and that can be used for quick comparisons [12]: the more similar the two texts, the more similar their simhash. Therefore, prior to any comparison, we compute a simhash for all the files of the knowledge base. When analyzing a JS file that is used by a given web application, we then generate its simhash and compare it, using the Hamming distance [13] to all simhashes of the knowledge base. We retain only the files from the knowledge base that have a Hamming distance $d \leq 3$. The thresholds of our strategy will be discussed in Section 4.

Since the multiple versions of a same library are usually very close, this process generally retains several files from the knowledge base. To retain only one result, we perform a more detailed comparison in the second step. We then compute the Dice coefficient for each candidate file on the bigrams they contain[14]. We retain only the files having a Dice coefficient $c \geq 0.8$, and among these files we retain only the file with the greatest coefficient. If there are several files with a same maximum coefficient, we distinguish two cases: 1) if the files do not come from the same library, we return no library and 2) if the files come from the same library, we return the file associated to the greatest version.

The file matching strategy has the main advantage to be robust to small modifications of the library, as it is sometimes done by web developers. It is quite easy to extend as it only requires the set of library files. Its first drawback is its cost in time. However, thanks to our optimization, the total comparison cost time is reasonable as we will measure it in Section 4. Its second drawback is that some web applications modify the source code of the library or merge several

```
name\s(.*\s)?v?([0-9]+)(\.[0-9]+)*
```

Figure 2: The regular expression used to extract library names and version from comments.

Table 2: Key URLs and objects for several libraries.

	Element	
JavaScript library	Key URLs	Key objects
jQuery	jquery.com	*window.jQuery*, *window.$*, *window.$jq*
Modernizr		*window.Modernizr*
Facebook SDK	connect.facebook.net	*window.FB*
Twitter Platform	platform.twitter.com	*window.twttr*

libraries into one JS file. In this situation this strategy fails.

2.2.3 Sensor Strategy

The *Sensor Strategy* aims at inserting at runtime a sensor in the web application with the objective to dynamically detect which JS libraries are deployed. Such a sensor is a JS plug-in that is executed by the browser. To detect a library, the sensor uses two elements. First it monitors the requests made by the browser in order to detect URLs (called key URLs) associated to a known library. Then it checks for the existence of JS objects (called key objects) that are associated to a known library.

For instance, if the sensor detects that the web application is requesting the *connect.facebook.net* URL, it means that it is using the *Facebook SDK* library. Similarly, if the sensor detects that a *window.$* object exists, it means that the application is using *jQuery*. Table 2 shows key URLs and objects for several well known libraries.

When a library is detected thanks to a key URL or object, our sensor then calls a specific function that aims to detect the version of the library. Such a function uses the internal knowledge of the library to recognize the version, i.e. objects specific to a particular version or functions that return the version. Further, the function returns ? when no version can be detected.

The sensor strategy has the advantage to be quite efficient. Its main drawback is that it requires a lot of configuration that has to be provided by an expert that knows the internal of a library. Therefore adding a new library in the knowledge base is expensive. Moreover, sometimes the libraries do not provide a mean to distinguish between its different versions (for instance same URLs are requested, and the same objects are defined whatever the versions). Further, as it relies on executing JS code coming from the web applications and the plugins, it sometimes fails due to some unexpected runtime error.

2.3 ARJL Combined Strategy

The three strategies we presented above give different results according to the input web application and to the sought library. For instance, Table 3 shows the results obtained by these three strategies on several couple of applications and libraries. In this table, we see that the library *jQuery* is detected with the same version by the three strategies for the Microsoft website. However, in the 360 website, *jQuery* is detected only by the two first strategies. Also in the Microsoft website, *Sizzle* is not detected by the sensor strategy, and detected in two different versions by the comment and

Table 3: Comparison of three recognition strategies.

Web application / JavaScript library	Strategy		
	Comment	File Matching	Sensor
`Microsoft / jQuery`	1.7.2	1.7.2	1.7.2
`360 / jQuery`	$\perp$	1.7.1	1.7.1
`DropBox / Modernizr`	2.8.3	$\perp$	2.8.3
`DropBox / Underscore.js`	1.8.3	1.8.3	$\perp$
`Microsoft / Sizzle`	1.9.4	1.9.2	$\perp$

file matching strategies.

Ssection 4 gives much more information regarding the precision of each of the strategy. However, we clearly see that there is no silver bullet: no strategy always return a correct result.

We can see that the results obtained by the three strategies should be merged. We therefore introduce ARJL (Automatic Recognizer of JS Libraries) that integrates the three strategies mentioned above, as follows. First the comment, file matching and sensor strategies are applied separately on a given web application. For each library l, we therefore get three results in the set $V_l \cup \{?, \perp\}$. Recall that V_l is a totally ordered set, i.e. $1.0 < 1.1 < 1.2$. We extend the total order of V_l by considering $\perp$ as the smallest element, and ? the second smallest, i.e. $\perp <? < 1.0 < 1.1 < 1.1$. Using this total order, for a library and a given web application, the ARJL strategy returns the greatest element in $V_l \cup \{?, \perp\}$ that has been computed by the strategies. For instance, in Table 3, for the *Microsoft* application and *Sizzle* strategy, we have $\perp < 1.9.2 < 1.9.4$ therefore 1.9.4 is returned.

3. IMPLEMENTATION

Our approach has been implemented using the JavaScript (JS) language on top of the SlimmerJS[4] scriptable and headless (without GUI) web browser. SlimmerJS is in charge of applying the three strategies. In particular, we use it to retrieve the JS files from the web applications (for the comments and file matching strategies), and also to look for the key URLs and objects (for the sensor strategy). Then, we apply the ARJL strategy on top of the obtained results.

Since SlimmerJS can experience a heavy network traffic when crawling a web application, or even crash when executing the remote JS code, we analyze each web application in a separate thread. A monitor watches all the threads and when one is running for too many time (we have a configurable threshold set by default to five minutes), it kills and relaunches it. Fortunately, we always succeed in analyzing all the web applications we wanted to analyze as the crashes rarely happen.

Since there exists a huge amount of JS libraries to detect, we had to choose a reasonable subset of them to test our approach. We therefore chose to select a limited set of famous JS libraries from two well-known sources. First, we chose to include the 14 JS libraries stored by the Google CDN[5]. Second, as Wikipedia[6] provides a list of notable JS libraries, we choose this list but remove the libraries without any update in the last five years or the ones that are totally abandoned. By merging these two lists, we obtain a list of 52 JS libraries, with an average of 43 versions per library.

To build the knowledge base of all files of all versions of these libraries, we use the CDNJS[7] library hosting website. To elaborate the key objects and URLs for the sensor strategy, we reused the source code of Library Detector[8], a Chromium plugin that detects the libraries used by a web application. We then have extended it to handle our sensor strategy.

For the list of web applications, we use the Alexa website that ranks websites according to their popularity. Alexa contains websites that are available on several domains (such as *google.com*, *google.co.in*, *google.co.jp*). We therefore remove the domain information from the URL, and select the 100 most popular ones.

To allow researchers and developers to replicate our results, the source code of our approach is available on GitHub[9].

4. EVALUATION

In this section we evaluate the strategies described in Section 2. Firstly, we describe how we set up the two thresholds of the file matching strategy. Then, we analyze the precision of our approach. We also compare the results of the different strategies. Finally, we measure the time performances of the strategies.

4.1 Thresholds of the File Matching Strategy

As described in the previous section, the file matching strategy uses two thresholds: the maximum Hamming distance d between simhashes, and the minimum Dice similarity s between the text of the files. To compute these thresholds our objectives was that the strategy should avoid at all costs false positives, and should return the largest set of identified libraries. In other words, the precision should be 100%, and the recall should be as big as possible (close to 1).

We then used a manual process that started with the stronger threshold (where $d = 0$ and $s = 1$), and aimed to release the thresholds to get more identified libraries without having any false positive. In other words, we tried to get the two thresholds with a precision of 100% and with the biggest recall. We ran that process on a set of 10 applications randomly chosen from the 100 most popular application from Alexa. Further one author of the paper manually inspected the detected libraries and classified each of them as either a true or a false positive. The Table 4 shows the results of our process where the candidate values for d are $\{0, 1, 2, 3, 4\}$, and $\{1, 0.9, 0.8, 0.7, 0.6\}$ for s. For each couple of thresholds (c, s) in the Table, there is an associated couple (t, f), where t (resp. f) is the number of true (resp. false) positives. According to our considerations, we therefore selected the thresholds $d = 3$ and $s = 0.8$ as it returned the more libraries (12) without any false positive (0).

4.2 Precision of the Strategies

[4] https://slimerjs.org/
[5] https://developers.google.com/speed/libraries
[6] https://en.wikipedia.org/wiki/List_of_JavaScript_libraries
[7] https://cdnjs.com/
[8] https://github.com/johnmichel/Library-Detector-for-Chrome
[9] https://github.com/kenmick/WebCrawler

Table 4: Hamming distance and Dice similarity thresholds, with the associated true and false positives.

	Dice coefficient				
Hamming distance	1	0.9	**0.8**	0.7	0.6
0	(4,0)	(9,0)	(9,0)	(9,0)	(9,0)
1	(4,0)	(10,0)	(10,0)	(10,0)	(10,0)
2	(4,0)	(10,0)	(10,0)	(10,1)	(10,2)
3	(4,0)	(10,0)	**(12,0)**	(12,2)	(12,4)
4	(4,0)	(10,0)	(12,1)	(12,5)	(12,7)

We consider two kinds of precision, called the *library-level* and the *version-level* precisions. The *library-level* precision focuses on library name, and ignores the versions. In such a level, a true positive is when a strategy returns a library that is truly used by the web application, whatever the version returned by the strategy and the one that is truly used by the application. A false positive is when a strategy returns a library that is not used by the web application. The *version-level* precision focuses on versions. In such a level, a true positive is when a strategy returns the version of a library that is truly used by the web application. A false positive is when a strategy returns a version that is not used by the web application. When the strategy does not return the version (unknown version), we don't consider that as a result, so it is not a true nor a false positive.

To evaluate these two precisions, we drew at random 20 web applications from the top 90 applications of Alexa (we excluded the ones used in the threshold experiment). We then ran our strategies on these applications and collected the results. One of the authors then checked if the results were true or false positives. To perform this check, the author just used all the development tools of Mozilla Firefox with the intent to check whether the return library is really used by the web application.

Table 5 shows the precision of our strategies. Regarding the *library-level*, both the comment and the file matching strategies have a 100% precision. The sensor strategy has a 93.5% precision mainly because few web applications use objects that have the same name than key objects. Regarding the *version-level*, the precision of all the strategies is very close to 90%. All together, the precision of our approach is 94.4% for the *library-level*, and 92.6% for the *version-level*, which is quite good.

4.3 Comparison of the Strategies

In this section, we perform a comparison of the results obtained using the different strategies. For this purpose, we ran the comment, file matching and sensor strategies on the 100 most popular applications of Alexa. In this experiment, we removed the version information retrieved by the strategies, so if a strategy returns $\{(jQuery, 1.2)\}$, it is transformed as $\{(jQuery, ?)\}$. Using this transformed data, we construct the Venn diagram of Figure 3 that shows the intersections of the results of the strategies. This diagram clearly shows that each strategy is useful because it identifies libraries that are not detected by the other strategies. The sensor strategy seems to outperform the comment strategy that in turns seems to outperform the file matching strategy as they find respectively 78%, 40% and 22% of all libraries. The set of uniquely detected libraries represent 53% of all libraries for the sensor strategy, 13% for the comment strategy, and 8% for the file matching strategy.

Table 5: Precision of the strategies.

	Strategies			
Precision	Comment	File Matching	Sensor	ARJL
Library-level	100%	100%	93.5%	**94.4%**
Version-level	88.9%	94.4%	90.9%	**92.6%**

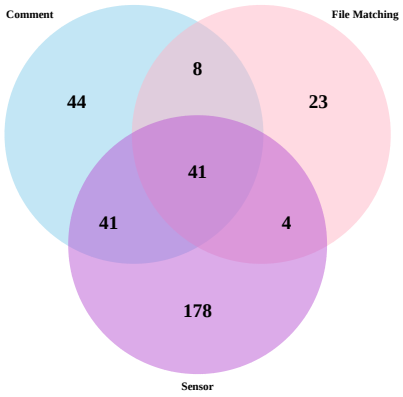

Figure 3: Venn diagram for 3 strategies

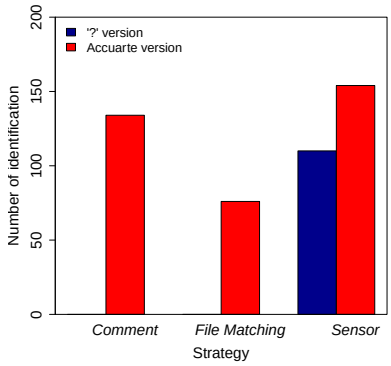

Figure 4: Comparison of '?' and accurate version for each strategies

Finally Figure 4 shows the comparison of '?' and accurate version for each strategy. It shows the number of accurate and unknown versions returned by a strategy. This figure confirms that the comment and file matching strategies never return unknown version, as expected. Only the sensor strategy returns unknown versions. This Figure also shows that the sensor strategy return more versions than the other ones, which also explain the Venn diagram of the Figure 3.

4.4 Efficiency

In this section, we assess the time performances of our strategies. First, SlimmerJS require 3 hours to browse all the web applications from top 100 applications of Alexa. Then, we measure the total time taken by each strategy to analyze

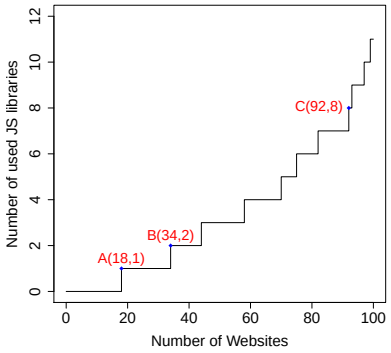

Figure 5: Statistics for Top 100 web applications on Oct. 20, 2015

all these applications. This time has been measured using a Intel Core i7-4770 CPU @3.40GHz×8, 16GB of RAM, and Ubuntu 14.04.2 LTS x86 64. We did not measure the time taken to compute the ARJL strategy because it only combines the results of the others in a few milliseconds. The fastest strategy is the comment one (0.1 hours). The sensor strategy is the longest one, since it takes two hours and a half to process all sources (2.5 hours). This is because runtime errors that can be experienced when running JS code. Finally, the file matching strategy takes 1 hour. In conclusion, the top 100 websites can be processed in less than 7 hours.

5. OBSERVATIONS AND SUGGESTIONS

This section presents our observations and provides some suggestions regarding the use of JavaScript (JS) client-side libraries in the context of web development. We first present some statistics on how famous web applications are using libraries. Then, we present the suggestions that can be provided looking at a recent snapshot of the library usage of the most famous web applications. Finally, we present our observations on the library usage during a long period of time. Such observations yield interesting insights regarding the pace of library evolution.

5.1 Statistics

Figure 5 presents the global usage of JS client-side libraries by the Alexa top 100 web applications. It shows that most of these web applications use several JS libraries: 82% of them use at least 1 library, 66% use at least 2 libraries, and 8% use at least 8 libraries (see the points A, B and C).

By looking into these web applications, we observed that top ranked web applications use few libraries, even if their company develops and maintains several famous ones. For example, the *Google* web application ranks first but does not use any library, even if the Google company provides widely used libraries such as *GoogleAnalytics* or *GoogleAPI*. On the contrary, web applications that have a lower ranking use more libraries. We have validated this observation by using a Spearman correlation test. The results of this test is $\rho = 0.37$ with a p-value of 0.0006. There is therefore a medium correlation between the rank of a web application in the top 100 and the number of library it uses.

Table 6: JS library usage frequency for Alexa global top 100 web applications on Oct. 20, 2015

Snapshot-2015-10-20	Library name	Frequency
1	jQuery	63
2	GoogleAnalytics	25
3	Modernizr	19
4	jQueryCookie	18
5	Underscore	12
6	jQueryUI	12
7	Facebook SDK	11
8	RequireJS	9
9	SWFObject	8
10	Twitter	7
...		
	Backbone	7
	Bootstrap	3
	AngularJS	0
	Polymer	0

Table 7: JS.ORG rank on Oct. 20, 2015

JS.ORG rank	Library name
1	AngularJS
2	D3
3	jQuery
4	RevealJS
5	React
6	ImpressJS
7	ThreeJS
8	Backbone
9	jQueryFileUpload
10	SocketIO
...	

These statistics reinforce our hypothesis that the libraries used by the famous web applications are the ones to use. Indeed as famous web applications use only a few libraries, we claim that they do use the libraries that provide a very strong added-value. However, this phenomenon questions the number of popular web applications that have to be considered to perform valuable observations that yield useful suggestions. We currently choose 100 web applications but we are not sure that this number is representative enough. Therefore this number could be increased or decreased depending on the objective: analyzing only widely used web applications, or analyzing also less famous web applications. We investigated how the results change when using 1000 applications, and there was a very limited impact on the results.

5.2 Analysis of the October 2015 Snapshot

Table 6 lists the JS library usage frequency for Alexa global top 100 web applications on Oct. 20, 2015. We can observe that *jQuery* is widely used by web applications and ranks first (63 web applications use it). In the opposite MVC (Model View Controler) libraries such as *AngularsJS*, *Backbone* and *Polymer*, are rarely used by famous web applications. These results contradicts the trends provided by JS.ORG[10], which gives statistics about popular JS projects on GitHub. Table 7, which presents the statistics of JS.ORG, states that *AngularsJS* ranks first which is very different from the results of our study.

We claim that our results confirm our hypothesis, and that they can be used to give some suggestions. In particular, we claim that libraries such as *jQuery* are essentials. On the contrary, libraries such as *AngularJS*, *Backbone* and *Polymer* should be avoided as they are possibly not yet mature enough to be included in web applications.

[10] http://stats.js.org/

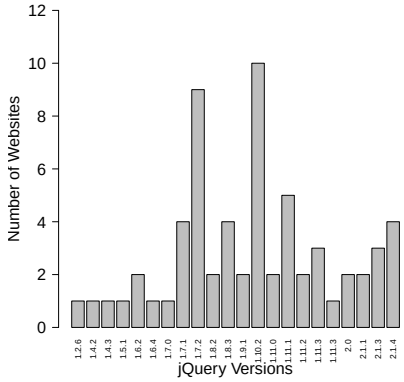

Figure 6: jQuery Version Distribution on Oct. 20, 2015

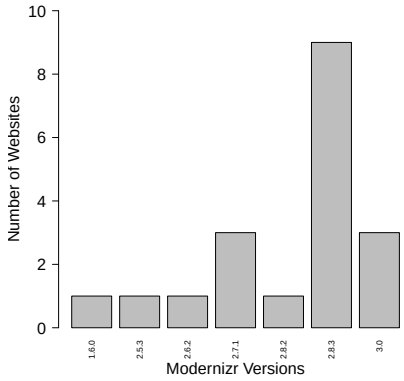

Figure 7: Modernizr Version Distribution on Oct. 20, 2015

Our observations that focus on versions can be used to give precise suggestions on a particular library. As an example, Figure 6 shows the version distribution of the *jQuery* library among the top 100 web applications on Oct. 20, 2015. According to this figure, *jQuery-1.7.2* and *jQuery-1.10.2* are the most popular versions. Moreover, most of the famous web applications prefer to use the version 1.x.x of *jQuery* than the version 2.x.x. Figure 7 presents our observations for the versions of the *Modernizr* library. This figure shows that the version 2.8.3 is the preferred one, and therefore may be used preferentially.

5.3 Analysis of a Three Years Period

We used *Wayback Machine*[11], the internet archive website, to get older versions of the Alexa top 100 web applications. Thanks to *Wayback Machine*, we are therefore able to observe which libraries were used through the history of these web applications. We focused on a three years period (from Oct. 20, 2012 to Oct. 20, 2015) to observe library usage evolution. Figure 8 shows such observations for the top 10 used JS libraries as computed from the October 2015 snapshot. It shows that *jQuery* is the first used library for three years, and exceeds to a large extent the other libraries. The *GoogleAnalytics* library has a steep increasing slope during 20 Oct'12 and 20 Oct'13, but tends to decline since 20 Apr'15. Further, in the beginning *Modernizr* is not widely used by web applications, but accumulates popularity and exceeds *jQueryCookie* to rank third around October 2014.

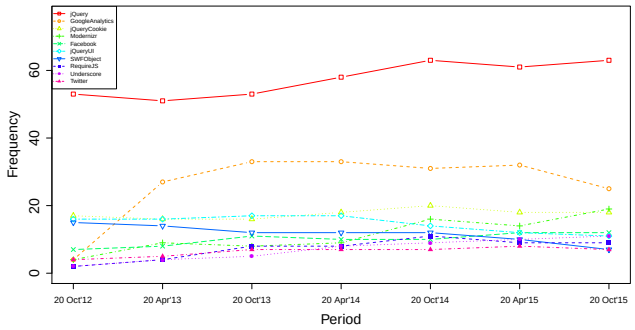

Figure 8: Evolution of top 10 JS libraries for three years

As a main conclusion, Figure 8 shows that the usage of client-side JS libraries evolves, but not that fast. During a three years period, which is quite long for web applications, there are few evolutions.

6. RELATED WORK

To the best of our knowledge, there is only one survey, which is done by the W3Techs company, that provides statistics on the client-side library usage among popular web applications.[12] However, W3Techs does not communicate its underlying methodology. On the contrary, our methodology is fully described, making it possible to reproduce or extend it. In the remainder of the section, we present the work done in the fields of analysis of third-party library usage, identification and recommendation.

In [1], Teyton et al. analyze a large set of open source software systems to mine their library migrations (replacement of a library by a competing one). Their approach is based on the analysis of the source code and build system artifacts, and cannot therefore be reused in our context.

In [15], Lämmel et al. perform a large-scale study on AST-based API-usage over a large set of open-source projects. We didn't use the same approach, as analyzing a JavaScript (JS) AST for mining library usage is very complex, because of dynamic typing and high order functions.

In [16], Mileva et al. perform an analysis of the Maven configuration files of 250 Apache projects to mine usage of libraries and their versions. Such an approach cannot be used in the context of web application as the configuration files are not available.

Seifert et al. [17] present a static method to detect malicious code in web applications. The method checks the initial HTTP response and HTML code to find static attributes. Our sensor strategy has been inspired by this approach.

Finally, Zhong et al. [18] and Thung et al.[19] provide approaches to automatically recommend library methods and

[11] https://archive.org/web/

[12] http://w3techs.com/technologies/overview/javascript_library/all

code snippets. Further, Thung et al. [10] also describe an new approach that can automate the recommendation of a library depending on the developers' needs. These approaches are however based on the source code of the projects, which is not available in our context of web applications.

7. CONCLUSION

In this paper we propose an approach to automatically identify client-side JavaScript (JS) libraries used web applications. Our approach combines three different strategies that respectively aim to (1) look for any name of a library in the comment part of the JS files linked to the application, (2) compare these same linked JS files with reference files of libraries, and (3) execute a sensor JS plug-in to dynamically identify library usage.

Our approach has the advantage to be highly precise (more than 92%) once the sought libraries are included the knowledge base of our approach. We used our approach on the 100 most popular websites referenced by Alexa. Our intent is to identify the popular client-side JS libraries, and to give some suggestions to the developers that have trouble to choose a library to include within their own web application.

Based on our observations, we can state that there are some essential libraries (such as *jQuery*, *Modernizr* or *Underscore*), and other ones that are rarely used (such as *AngularJS* and *Polymer*). This contrasts with ranking that are provided by websites dedicated to software development, and that states that *AngularJS* is very popular. We argue that our approach is focused on popular and commercial web applications, and therefore is resilient to technological buzz.

As a last result, the observations we made on a three years period shows that the library usage evolves but not that fast. Most, if not all, the libraries that were used three years ago, are still used nowadays. As a future work, we think about partitioning web applications into several business domains to check whether some libraries are more used depending on these domains.

8. ACKNOWLEDGEMENT

This work is supported by the National Natural Science Foundation of China under Grant No.61300006.

9. REFERENCES

[1] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in Java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, Nov. 2014.

[2] A. Mardan, *Practical Node.Js: Building Real-World Scalable Web Apps*, 1st ed. Berkely, CA, USA: Apress, 2014.

[3] B. Bibeault and Y. Katz, *Jquery in Action*. Greenwich, CT, USA: Manning Publications Co., 2008.

[4] M. T. Baldassarre, A. Bianchi, D. Caivano, and G. Visaggio, "An Industrial Case Study on Reuse Oriented Development," in *Proceedings of the 21st IEEE ICSM*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 283–292.

[5] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining Library Migration Graphs," in *19th WCRE Conference, 2012*, IEEE, Ed., Kingston, Ontario, Canada, Oct. 2012, pp. 289–298.

[6] R. Baskerville, B. Ramesh, L. Levine, J. Pries-Heje, and S. Slaughter, "Is Internet-Speed Software Development Different?" *IEEE Software*, vol. 20, no. 6, pp. 70–77, Nov. 2003.

[7] V. Bauer, L. Heinemann, and F. Deissenboeck, "A structured approach to assess third-party library usage," in *28th IEEE ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 483–492.

[8] R. Kula, D. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, Mar. 2015, pp. 520–524.

[9] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic Discovery of Function Mappings between Similar Libraries," in *20th WCRE Conference 2013*. IEEE, 2013, pp. 192–201.

[10] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *20th WCRE Conference, 2013*, Oct. 2013, pp. 182–191.

[11] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, 2005.

[12] C. Sadowski and G. Levin, "Simhash: Hash-based similarity detection," 2007.

[13] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[14] L. R. Dice, "Measures of the amount of ecologic association between species," *Ecology*, vol. 26, no. 3, pp. 297–302, 1945.

[15] C. McMillan, M. Linares-Vasquez, D. Poshyvanyk, and M. Grechanik, "Categorizing software applications for maintenance," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 343–352.

[16] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM, 2009, pp. 57–62.

[17] C. Seifert, I. Welch, and P. Komisarczuk, "Identification of malicious web pages with static heuristics," in *Telecommunication Networks and Applications Conference, 2008. ATNAC 2008. Australasian*. IEEE, 2008, pp. 91–96.

[18] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP 2009–Object-Oriented Programming*. Springer, 2009, pp. 318–343.

[19] F. Thung, S. Wang, D. Lo, and J. Lawall, "Automatic recommendation of api methods from feature requests," in *28th International Conference on Automated Software Engineering (ASE), 2013 IEEE/ACM*. IEEE, 2013, pp. 290–300.