

# Supervised Categorization of JavaScript<sup>TM</sup> using Program Analysis Features

Wei Lu

*Singapore-MIT Alliance, E4-04-10, 4 Engineering Drive 3, Singapore, 117576*

Min-Yen Kan

*Department of Computer Science, School of Computing, National University of Singapore, Singapore, 117543*

---

## Abstract

Web pages often embed scripts for a variety of purposes, including advertising and dynamic interaction. Understanding embedded scripts and their purpose can often help to interpret or provide crucial information about the web page. We have developed a functionality-based categorization of JavaScript, the most widely used web page scripting language. We then view understanding embedded scripts as a text categorization problem. We show how traditional information retrieval methods can be augmented with the features distilled from the domain knowledge of JavaScript and software analysis to improve classification performance. We perform experiments on the standard WT10G web page corpus, and show that our techniques eliminate over 50% of errors over a standard text classification baseline.

*Key words:* Information Retrieval, Machine Learning, JavaScript, ECMAScript, Program Comprehension, Source Clone, Program Pattern, Software Metrics, Program Classification, Automated Code Classification

---

## 1 Introduction

As the web has become more interactive, tasks such as form processing, uploading, scripting, applets and plug-ins have transformed the web page into a dynamic application. While a boon to human users, such dynamic aspects of web page scripting and applets impede the machine parsing and understanding of web pages. When

---

*Email addresses:* luwei@nus.edu.sg (Wei Lu), kanmy@comp.nus.edu.sg (Min-Yen Kan).

such functionality is present, these crucial functionalities are lost if not processed. Pages with JavaScript, Macromedia Flash and other plug-ins are only starting to be indexed and analyzed by web crawlers and indexers. Interactivity in web pages is increasing as more web content is provided using complex content management systems, which use scripting to create a more interactive experience for the human user. Asynchronous Javascripting and XML (AJAX) is an example of such an emerging technique. If automated indexers are to keep providing accurate and up-to-date information, methods are needed to glean information about the dynamic aspects of web pages.

To address this problem, we consider a technique to automatically categorize uses of JavaScript, a popular web scripting language. In many web pages, JavaScript realizes many of the dynamic features of web pages. We chose to focus on JavaScript as 1) its code is inlined within an HTML page and 2) embedded JavaScript often interacts with other static web page components (unlike applets and plug-ins). We leverage on both of these key properties in our analysis.

An automatic categorization of JavaScript can assist both indexing software to accurately model web pages' functionality and requirements and browsers to select allow certain scripting functions to run and to disable others. Pop-up window blocking, which has been extensively researched, is just one of the myriad uses of JavaScript that would be useful to categorize. Such software can assist automated web indexers to report useful information to search engines and allow browsers to block annoying script-driven features of web pages from end users.

We start with a system that employs traditional text categorization metrics as a baseline. Although the resulting baseline system performs reasonably, we pursue a machine learning framework that draws on features from text categorization, program comprehension and code metrics to improve performance. We show that the incorporation of features that leverage knowledge of the JavaScript language together with program analysis improves categorization accuracy. We conduct evaluation of our methods on the widely-used WT10G corpus (Hawking, 2004) to validate our claims and show that the performance of our system eliminates over 50% of errors over the baseline.

In the next section, we examine earlier attempts at source code categorization and discuss how features of the JavaScript language and techniques in program analysis can assist in categorization. We then present our methods that distills features for categorization from the principles of program analysis. We describe our experiment and analysis, and conclude by discussing our manual analysis and future directions of our work.

## 2 Background

A survey of previous work shows that the problem of automated computer software categorization is relatively new. We believe that this is due to two reasons. First, programming languages are generally designed to be open-ended and largely task-agnostic. Languages such as FORTRAN, Java and C are suitable for a very wide range of tasks and attempting to define a fixed categorization scheme for programs is largely subjective, and likely to be ill-defined. Second, the research fields of textual information retrieval (IR) and program analysis have largely developed independently of each other. We feel that these two fields have a natural overlap which can be exploited.

Unlike natural language texts, program source code is unambiguous to the compiler and has exact syntactic structures. This means that syntax plays an important role that needs to be captured. This has been largely ignored by text categorization research.

(Ugurel et al., 2002; Krovetz et al., 2003) are perhaps the first work that uses IR methods to attack this problem. They employ support vector machines for source code classification in a two-phase process consisting of programming language classification followed by topic classification. In the second, topic classification task, they largely relied on each software projects' README file and comments. From the source code itself, only included header file names were used as features. We believe a more advanced treatment of the source code itself can assist such topic classification. These features, including syntactic information and some language-specific semantic information, could be important and useful for classification. Other recent work in categorizing web pages (Wong and Fu, 2000) has revived interest in the structural aspect of text. One hypothesis of this work is that informing these structural features with knowledge about the syntax of the programming language can improve source code classification.

Program analysis, in which formal methods are employed, has developed into many subfields. The subfield of program comprehension develops models to explain how human software developers learn to comprehend existing code (Mathias et al., 1999). These models show that developers use both top-down and bottom-up models in their learning process (von Mayrhauser and Vans, 1994). Top-down models imply that developers may use a model of program execution to understand a program. Formal analysis via simulated code execution (Blazy and Facon, 1998) may yield evidence for automated categorization techniques.

Comprehension also employs code metrics, which measure the complexity and performance of programs. Of particular interest to our problem scenario are code reuse metrics, such as (Krsul and Spafford, 1995; Kontogiannis, 1997; Kapsler and Godfrey, 2004; Cory Kapsler, 2005), as JavaScript instances are often copied wholesale

or modified from standard examples. In our experiments, we assess the predictive strength of these metrics on program categories.

We believe that a standard text categorization approach to this problem can be improved by adopting features distilled from program analysis. Prior work shows that the use of IR techniques, such as latent semantic analysis can aid program comprehension (Maletic and Marcus, 2000). The key contribution of our work shows that the converse is also true: the formal and exact results of program analysis are good features that assist in the IR frameworks for source code categorization.

### 3 JavaScript categorization

The problem of JavaScript program categorization is a good proving ground to explore how these two fields can interact and inform each other. JavaScript is a niche language – mostly confined to web pages and performing limited number of tasks. We believe this is due to the restrictions of HTML and HTTP, and because web plug-ins are more conducive an environment for applications that require true interactivity and fine-grained control. We feel this property makes the text categorization approach well-defined, in contrast to categorization of programs in other programming languages, a hypothesis that we verify later in Section 6. Secondly, JavaScript has an intimate relationship with the HTML elements in the web page. Form controls, divisions and other web page objects are controlled and manipulated in JavaScript. As such, we can analyze how the web page’s text and its HTML tags, in the form of its document object model (DOM), affect its categorization.

What type of categorization is suitable for JavaScript? Recalling our purpose, the categories should indicate to the end user the usefulness or functionality of the script. We examined two previous source code classifications to judge whether they can be used for our work:

- (1) **Ugurel *et al.* (Ugurel et al., 2002)** proposed 11 topic categories for source code topic classification task, including *circuits*, *database*, and *development*. They assessed their work on different types of languages, such as Java, C, and Perl. Their work are based on large software systems and therefore these categories are designed for topical classification of general software systems and does not fit our domain well.
- (2) **www.js-examples.com** is an existing JavaScript categorization from a well-known tutorial site. This site has over 1,000 JavaScript examples collected worldwide. To allow developers to locate appropriate scripts quickly, the web site categorizes these examples into 54 categories, including *ad*, *encryption*, *mouse*, *music* and *variable*.

While a good starting point, the *js-examples* categorization has two weaknesses that made it unusable for our purposes. First, the classification is in-

tended for the developer, rather than the consumer. Examples that have similar effects are categorized differently if the implementation is different. In contrast, we intend to categorize JavaScript functionality with respect to the end user. Second, the classification is used for example scripts, which are usually truncated and for illustrative purposes. Real-world JavaScript does not illustrate these niceties.

Table 1  
JavaScript functional categories, sorted by frequency in our corpus (discussed in Evaluation). Number of instances indicated in parenthesis in the description field.

Category	Description (# units in corpus)
Dynamic-Text Banner	Displays a banner that changes content with time(264)
Initialization	Initializes/modifies variables for later use (123)
Form Processing	Passes values between fields, or computing values from form fields (119)
Calculator	Displays and manipulates a calculator (88)
Image Pre-load	Pre-loads images for future use (87)
Pop-up	Pops up a new window (80)
Changing Image	Changes the source of an image (79)
HTML	Generates HTML components, such as forms (68)
Web Application	Runs web applications such as games & e-commerce (62)
Background Color	Changes or initializes the background color (50)
Form Validation	Validates a forms data fields (50)
Page	Loads a new page to the browser window (49)
Plain Text	Prints some text to the page (46)
Multimedia	Loads multimedia (43)
Static-Text Banner	Displays a banner which does not change with time(42)
Static Time Information	Displays static system time (41)
Loading Image	Loads and displays images (39)
Form Restore	Restores form fields to default values (37)
Last Modified Time	Displays the last modified time of the document(36)
Dynamic Clock	Displays a clock that changes with system time(35)
Navigation	Navigates a site (32)
Browser Information	Checks browser identification or information (32)
Cookie	Stores or retrieves data on server about client (26)
Trivial	Performs a simple one-liner task (24)
Interaction	Serves to assist user interaction with the page (24)
Warning Message	Displays static warning message (16)
Timer	Displays a timer which is running (10)
Greeting	Displays a greeting to the user (10)
CSS	Changes the Cascading Style Sheet of the page (7)
Client-Time based Counter	Displays interval between current time and another time relevant to page (7)
Visiting Browser History	Visits a page from the browser's history (6)
Calendar	Displays a calendar (5)
Others	Performs multiple functions or has too few training instances (133)

To deal with these shortcomings, we decided to modify the *js-examples* scheme based on a study of JavaScript instances in actual web documents. We see that JavaScript that natively occurs in actual web pages is different. Actual web pages often embed multiple JavaScript instances to achieve different functionality. Also, scripts can be invoked at load time or by triggering events that deal with interaction with the browser. For example, a page could have a set of scripts that performs browser detection (that runs at load time) and another separate set that validates form information (that runs only when the text input is filled out). In addition, some scripts are only invoked as subprocedures of others. All these aspects make these real-world JavaScript instances more difficult to handle than the ones on the example site.

As such, we perform categorization on individual JavaScript functional units, rather than all of the scripts on a single page. A *functional unit*, or simply *unit*, is defined as a JavaScript instance, combined with all its (potentially) called subprocedures. Given a web page, JavaScript code fragments are segmented from one another by how they are invoked (automatically upon loading or by a particular user action, like a mouse click on a form input). All relevant variable declarations, function calls, objects and HTML that are associated with the invoked code fragment form the unit. Figure 1 shows an example.

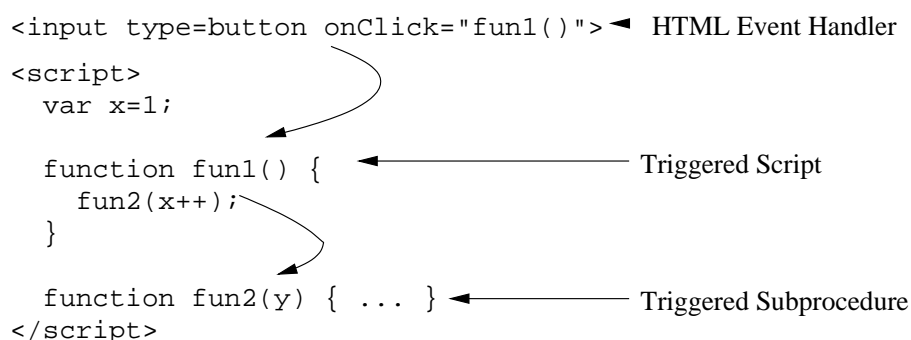


Fig. 1. A JavaScript unit, the basic element used for our classification.

We base our categorization of JavaScript on these automatically extracted units. Based on our corpus study, we created a classification of JavaScript into 33 discrete categories, shown in Table 1. These categories are based on functionality rather than by their implementation technique. A single *other* category is used for scripts whose purpose is unclear or contains more than one basic functionality.

We have made our dataset, annotations and categories freely available for research use and encourage others to make use of this resource. The dataset consists of the complete set of 18,683 documents which contain JavaScript, taken from the WT10G corpus. Web pages with erroneous JavaScripts are removed or corrected. We also provide the 1,637 JavaScript *functional unit* instances used in our evaluation, along with a single set of gold standard annotations made by the first author. We have also made our system prototype freely available online for research pur-

pose. Details of these resources will be presented at the conclusion of the paper.

## 4 Methods

Given such a categorization scheme, a standard text categorization approach would tokenize a training set (already classified) into units and use the resulting tokens as features to build a model. New, unseen test units are then tokenized and the resulting features are compared to the models of each category. The category most similar to the test unit is inferred as its category.

A simple approach could use a compiler’s own tokenization, treating the resulting tokens as separate dimensions for categorization. Using the compiler’s tokenization is important, as whitespace may not indicate all token separations (*e.g.*, “`x=2`”). An  $n$  dimensional feature vector results, where  $n$  is the vocabulary size (*i.e.*, total number of unique tokens that occur in all training unit instances). Usually, tens of thousands of features result, and the positive instances of a typical feature are rare, perhaps limited to a few training example each.

We improve on this text categorization baseline in three ways. We first show how tokenization can be improved by exploiting the properties of the programming language. Second, we show that certain code metrics can help. Third, features distilled from program comprehension in the form of static analysis and dynamic execution allow us to analyze how objects interact with each other; evidence which can further improve a unit’s classification.

### 4.1 *Using language features for improved tokenization*

We can improve the tokenization by leveraging the syntax of the programming language. A syntactic analysis of JavaScript types the unit’s tokens and can distinguish the tokens as numeric constants, string constants, comments, language-specific reserved keywords, arithmetic operators, regular expression variable or method names. These types are illustrated in Table 2.

As JavaScript’s is aimed at Web constructs, we further distinguish string constants as URLs, file extensions of images and multimedia, HTML tags and color values. Tokens of these types are tagged as such and their aggregate counts are used as additional features for categorization, as illustrated in Table 3.

Variable and method names are special as they often convey the semantics of the program. However, for convenience, programmers frequently use abbreviations or short forms for these names. For example, in the JavaScript statement `var`

Table 2

Examples of JavaScript tokenization and type-binding. **KEY**:keyword; **VAR**:variable; **SYM**:symbol; **NUM**:number; **STR**:string; **CMT**:comment; **REG**:regexp

Original JavaScript code	Segmented and tagged JavaScript code
<code>var x=1</code>	<code>var [KEY] x[VAR] =[SYM] 1[NUM] ;[SYM]</code>
<code>x *= 1.23e4;</code>	<code>x[VAR] *=[SYM] 1.23e4[NUM] ; [SYM]</code>
<code>alert("x="+x);</code>	<code>alert[VAR] ([SYM] "x=" [STR] +[SYM] x[VAR] ) [SYM] ;[SYM]</code>
<code>x++//cmt</code>	<code>x[VAR] ++[SYM] //cmt [CMT]</code>
<code>var re = /mac/i;</code>	<code>var [KEY] re[VAR] =[SYM] /mac/i[REG] ; [SYM]</code>

Table 3

Examples of string token normalization.

Example	Pattern	Sub-type of String Token
<code>var color[0] = "#0FF0F0"</code>	# header(optional) + (6-digit Hex value)	Color Value Token
<code>document.write("&lt;SCRIPT"&gt;")</code>	String contains HTML tag	HTML Tag Token
<code>parent.location = "../4.html"</code>	Is a well formed relative URL string	URL Token
<code>document.btn1.src = '6.GIF'</code>	Has image file extension <i>e.g.</i> .JPG .GIF	Image File Extension Token
<code>this.value = v + ".wav"</code>	Has multimedia file extension <i>e.g.</i> .mid,.wav	Multimedia File Extension Token

Table 4

Examples of name token normalization.

Example	Transition Pattern	Result (with expansion)
<code>curMsg</code>	single lowercase $\Leftrightarrow$ single uppercase	current message
<code>IPAddress</code>	consecutive uppercase $\rightarrow$ lowercase	ip address
<code>thisweek</code>	no transition and length $\geq 6$	this week

`currMon = mydate.getMonth()`, `currMon`, `mydate` and `getMonth` are short forms for “current month”, “my date” and “get month” respectively.

To a classification system, tokens such as `currMon` and `curMonth` are distinct and unrelated. To connect these forms together, we need to normalize these non-standard words (Rowe and Laitinen, 1995). We normalize such words by identifying likely splitting points and then expanding them to full word forms. Splitting is achieved by identifying case changes and punctuation use. Tokens longer than six letter in length are also split into smaller parts using entropy reduction, previously used to split natural languages without delimiters (*e.g.*, Chinese). An expansion phase follows to map commonly abbreviated shortenings to their word equivalents (*e.g.*, “`curr`” and “`cur`”  $\rightarrow$  “current”) using a small (around 20 entries) hand-compiled dictionary.

## 4.2 Code metrics

Complexity metrics measure the complexity of a program with respect to data flow, control flow or a hybrid of the two. Recent work in metrics (Kapsner and Godfrey, 2004; Krsul and Spafford, 1995) has been applied to specific software families and most metrics are targeted to much larger software projects (thousands of lines of code) than a typical JavaScript unit (averaging around 28 lines) in our corpus. As such, we start with simple, classic complexity metrics to assess their impact on categorization. We also designed two additional metrics that capture some of the unique properties that we encountered in JavaScript.

### 4.2.1 Standard metrics

#### (1) Counting metrics

**Logical lines of code (LLOC)** simply counts the number of lines of code.

We obtain this number by counting the number of program lines after pretty-printing.

**Number of attributes (NOA)** counts the number of fields declared in the class or interface. For JavaScript, we interpret this as the count of declared variables and newly-created objects in the source code.

**Number of procedures (NOP)** counts the number of potential procedures invoked during execution.

#### (2) Structure metrics

**Total cyclomatic complexity (TCC)** is a variation of the classic control flow metric, cyclomatic complexity (CC). Formally stated, the cyclomatic complexity of a graph  $G$  is defined as  $E - N + 2$ , where  $E$  is the number of edges in the control flow graph and  $N$  is the number of nodes in the same graph. In practice, it is the number of test conditions in a program. We define TCC as  $Sum(CC) - Count(CC) + 1$ , where  $Sum(CC)$  is the sum of individual  $CC$  value over all procedures, and  $Count(CC)$  equals the number of procedures.

**Decision density (DECDENS)** shows the average cyclomatic density within the JavaScript unit, defined as  $DECDENS = CC/LLOC$ , where  $CC$  is the cyclomatic complexity,  $LLOC$  refers to the logical lines of code.

**Depth of conditional nesting (DCOND)** . Whereas cyclomatic complexity deals with the absolute number of branches, nested conditionals counts how deeply nested these branches are.

**Depth of looping (DLOOP)** equals the maximum level of loop nesting in a procedure.

#### (3) Information flow metrics

**Informational fan-in (IFIN)** measures the amount of data used within a class, interface, constructor or method. It is defined as  $IFIN = P + R + G$ , where  $P$  is the number of procedures called,  $R$  is the number of parameters read,

$G$  is the number of global variables read. In our implementation, we treat a single JavaScript unit as a whole block,  $P$  is the total number of procedures in the unit,  $R$  is the sum of the number of parameters read for all these procedures, and  $G$  is the number of global variables read for all codes in the unit.

The above listed are classic software metrics employed in software measurement tasks and are generally applicable for different general-purpose programming languages.

#### 4.2.2 JavaScript language-specific metrics

As stated earlier, software metrics are now being developed for specific languages and for specific task domains, we have developed some additional metrics which attempt to capture idiosyncratic phenomenon that occur in our corpus.

**Similar/Repeating Statements (SS)** counts the number of statements with highly similar/repeating structure. The similarity measure of structure is examined by abstract syntax trees given in 4.2.3. This is helpful in detecting scripts that perform similar tasks in a conditional manner. For example, displaying different greetings based on the time (*e.g.*, morning, afternoon evening).

**Number of Built-in Object References (BO)** counts the number of built-in Java objects (*e.g.*, math, date) referenced by the unit.

**Number of Object Function Invoked (OF)** counts the number of object-function invoked in the code. Object functions are a unique feature of JavaScript. One can define and instantiate an object via a function. Figure 2 gives an example of function object. This is helpful in detecting scripts involving complicated tasks like games or e-commerce.

```
function PC(brand,price){
  this.brand = brand;
  this.price = price;
}
var myPC = new PC("ibm",1789);
```

Fig. 2. Example of an object function

#### 4.2.3 Code Reuse using Edit Distance

We can also measure code reuse (also referred to as clone or plagiarism detection). This is particularly useful as many developers copy (and occasionally modify) scripts from existing web pages. Thus similarity detection may assist in classification. Dynamic programming can be employed to calculate a minimum edit distance between two inputs using strings, tokens, or trees as elements for computation.

We employ a standard string edit distance (SED) algorithm to calculate similarity

<pre> if ( x&gt;1 ) {   alert ("hi"); } </pre>	<pre> alert ("hi") </pre>	<p><b>SED</b> there is common subsequence</p>
<pre> BLOCK ├── IFNE │   ├── GT │   │   ├── NAME x │   │   └── NUMBER 1.0 │   └── BLOCK │       ├── STMT │       │   └── CALL │       │       ├── NAME alert │       │       └── STRING hi </pre>	<pre> STMT └── CALL     ├── NAME alert     └── STRING hi </pre>	<p><b>TED</b> different roots results in large edit distance</p>
<pre> if[KEY] ([SYM] x[VAR] &gt;[SYM] 1[NUM] ) [SYM] ([SYM] alert[VAR] ([SYM] "hi"[STR] ) [SYM] ;[SYM] )[SYM] </pre>	<pre> alert[VAR] ([SYM] "hi"[STR] ) [SYM] </pre>	<p><b>LED</b> more reasonable and accurate</p>

Fig. 3. String based edit distance(SED), tree based edit distance(TED) and lexical-token based edit distance(LED)

between two script instances. We use the class of the minimal distance training unit as a separate feature for classification. However, this measure does not model the semantic differences that are introduced when edits result in structural differences as opposed to variable renaming. A minimal string edit distance may introduce drastic semantic changes, such as an addition of a parameter or deletion of a conditional statement.

To address this we again turn to program analysis. In program analysis, abstract syntax trees (ASTs) (Baxter et al., 1998) are often used to model source code and correct for these discrepancies. An AST is a parse tree representation of the source code that models the control flow of the unit and stores data types of its variables. Therefore we can use the AST model to define a tree-based edit distance (TED) measure between two JavaScript units. TED algorithms are employed in syntactic similarity detection (Yang, 1991). However, as is shown in Figure 3, the given two code fragments are of the same functionality, but have different syntactic structures. Hence, syntactic difference does not necessarily imply functionality similarity and vice versa. In this manner, a standard TED algorithm used in syntactic similarity detection is not likely to outperform a simple SED algorithm for our task. We can also measure similarity using a token-based edit distance (lexical-token based edit distance, LED), in which source code is parsed into a stream of tokens for distance computation (Kamiya et al., 2002). Edit costs are assigned appropriately depending on token types and values. We have implemented all three models and have assessed each approach’s effectiveness.

### 4.3 Program comprehension using the Document Object Model

So far we have considered JavaScript units as independent of their enclosing web pages. In practice, since JavaScript units may be triggered by HTML objects and may then manipulate these HTML objects in turn, a JavaScript unit has an intimate relation with its page and is often meaningful only in context. These objects are represented by a document object model (DOM)<sup>1</sup>. In fact, a unit which does not interact with a DOM object cannot interact with the user and is considered uninteresting. Many variables used in JavaScript are DOM objects whose data type can only be inferred by examining the enclosing HTML document. Figure 4 illustrates two examples where the script references DOM objects.

Fig. 4. Units that reference their HTML context.

<pre> window.document. getElementById(   'seminar'). choice[2].value; </pre>	<p>Accesses the value of the second radio button in a form "seminar"</p>
<pre> top.newWin.document. all.airplane. img2.src; </pre>	<p>Accesses the source of an image "img2" in the form "airplane", embedded in a window "newWin".</p>

We classify references to DOM objects into four categories: *gets*, *sets*, *creates*, and *calls*. These are illustrated in the JavaScript unit in Figure 5: on line 1, `doIt()` gets a reference to a form object; on line 4, the input object represented by `frm.txt` is set to a value `'ok'`; on line 2, a new window object is created by the method `window.open`; and on line 3, the object `document` calls its `write` method. The count of each of these DOM object references is added as an integer feature for categorization.

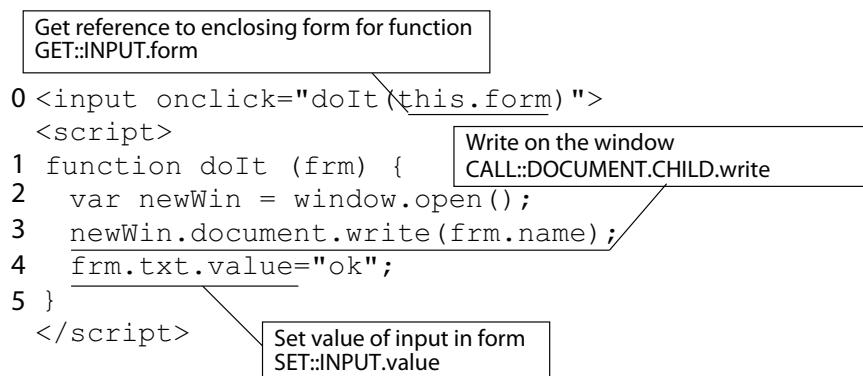


Fig. 5. Types of DOM object references.

<sup>1</sup> Although the browser object model (BOM) is distinct from the DOM, we collectively refer to the two models as DOM for readability.

### 4.3.1 Static analysis

Certain aspects of the communication between the DOM objects and the target JavaScript can be done by an inspection of the code. We extract two types of information based on this static analysis: triggering information and variable data types.

Certain classes of JavaScript are triggered by the user's interaction with an object (*e.g.*, a form input field) and others occur when a page is loaded, without user interaction. This triggering type (interactive, non-interactive) is extracted for each unit by an inspection of the HTML. For units triggered by interaction, we further extract the responsible DOM object and event handler. We also extract the lexical tokens from the enclosing web page elements for interactive units. For example, an input button with a text value "restore" is likely to trigger a unit whose class is *form restore*; likewise, button inputs with text labels such as "0", "1", and "9" are indicative of the class *calculator*.

DOM object settings and values may flow from one procedure to another. We recover the data type of objects by tracing the flow as variables are instantiated and assigned. This is done with the assistance of the abstract syntax tree described in 4.2.3. A variable and its data type form a single unified token (*e.g.*, `newWin`  $\mapsto$  `WINDOW`) used for categorization. In addition, all the JavaScript unit's interaction with DOM objects are then traced statically and recorded (*e.g.* `GET::INPUT.value`) as static analysis features for categorization. Table 5 shows some example DOM objects together with their respective methods and attributes that are traced during static analysis.

Table 5

Partial table describing DOM objects references traced in static analysis

Example DOM object	Methods	Attributes
<code>document</code>	<code>write()</code> , <code>open()</code> ...	<code>cookie (SET/GET)</code> <code>bgColor (SET/GET)</code> ...
<code>window</code>	<code>open()</code> , <code>prompt()</code> ...	<code>top (GET)</code> <code>status (SET/GET)</code> ...
<code>input (radio)</code>	<code>select()</code> , <code>click()</code> ...	<code>name (SET/GET)</code> <code>value (SET/GET)</code> ...

### 4.3.2 Dynamic analysis

Static analysis is not able to recover certain information that occurs at run time. Dynamic analysis (*i.e.*, simulated execution of code) can extract helpful features along a default path of execution. Although dynamic analysis is incomplete (in the sense that it only examines a single execution path), such analyses can determine exact values of variables and may help by discarding unimportant paths.

We illustrate how dynamic analysis can yield additional features for categorization in Figure 6. This sample JavaScript unit creates a dynamic text banner that scrolls in window's status bar. The function `window.setTimeout()` displays the string represented by ``banner(''+index+'')` after 100 milliseconds, which



approaches that can speed up this feature generation step.

Our experiments aim to measure the performance difference using different sets of machine learning features. In all of the experiments, the baseline model tokenizes units and passes the tokens as individual features to the learner.

Table 6

Component Evaluation Results. Error reduction (ER) is measured against the text categorization baseline. \* and \*\* indicate the improvement over the previous feature set is statistically significant by the one tailed T-test at the 0.05 and 0.01 level, respectively

Features used	Accuracy	ER
Most frequent class baseline	16.12%	–
Text categorization baseline	87.47%	–
L. All lexical analysis	89.61%**	17%
L <sub>c</sub> . Language token counting	88.57%	8%
L <sub>n</sub> . Function/variable normalization	87.66%	1%
M. All software metrics	77.76%	–
M <sub>s</sub> . Standard classic metrics	20.46%	–
M <sub>j</sub> . w/ new metrics (M <sub>s</sub> +SS+BOR)	25.60%	–
M <sub>e</sub> . String-based edit distance	73.85%	–
M <sub>a</sub> . AST-based edit distance	72.69%	–
M <sub>t</sub> . Token-based edit distance	74.89%	–
P. All program comprehension	87.29%	–
P <sub>s</sub> . Static analysis	79.78%	–
P <sub>d</sub> . Dynamic analysis	71.22%	–
L+M	90.04%*	21%
L+P	92.36%**	39%
L+M+P	93.95%**	52%

Table 6 shows the component evaluation in which we selected certain combination of features as input to the SVM classifier. Here, we can see a simple majority class categorization performs poorly, as this dataset consists of many classes without a dominating class. However, a simple text categorization baseline, in which strings are delimited by whitespaces performs very well and is accurate on 87% of the test instances. When informed lexical tokenization is done and combined with features from software metrics, static and dynamic analysis, we are able to improve categorization accuracy to around 94%. Perhaps unsurprisingly, using only software metrics and program comprehension features fail to contribute good clas-

sifiers. However, when coupled with a strong lexical feature component, we show improvement.

A good baseline performance may seem discouraging for research, but many important problems exist which exhibit the same property (*e.g.*, spam detection, part of speech tagging). These problems are important and small performance gains are quite relevant. As such we also calculate the error reduction that is achieved by our methods over the text categorization baseline. By this metric, almost half of the classification errors are corrected by the introduction of our techniques. We now examine the individual feature types.

**Lexical Analysis:** We hypothesized that token features and variable and function name normalization would enhance performance. The results show that simple typing of tokens as keywords, strings, URLs and HTML tags is effective at removing 8% of the categorization errors. Variable and function name splitting, expansion are proven to be less effective on our corpus. When both techniques are used together, synergy results removing 17% of errors. This validates our earlier hypothesis that program language features do positively impact program categorization.

**Metrics:** We also break down our composite metric feature set into its components to assess their predictive strength. Our results also show that edit distance alone is not sufficient to build a good categorizer. Such a code reuse metric is not as accurate as our simple text categorization baseline. A finding of our work is that applying published software metrics “as-is” may not boost categorization performance much, rather these metrics need to be adapted to the classes and language at hand. Only when collectively used with lexical analysis is performance increased.

**Program Comprehension:** Static and dynamic features alone do not perform well, but their combination greatly reduces individual mistakes (29% and 51% for the static and dynamic analyses, respectively). The combined feature set also does not beat the simple lexical approach, but does serve to augment its performance.

## 6 Annotation evaluation

Recall that the goal of our JavaScript categorization is to separate useful JavaScript units from those that are useless or annoying to the end user. To do this, we argued that a functionality-based categorization is a good starting basis and we used a single set of annotations to train a classifier to achieve such categorization. Two issues emerge from this argument: 1) do the proposed functionality-based classes actually correlate with end users’ perception of usefulness? and 2) do people agree with our proposed classification scheme and are able to replicate our annotation?

To answer the first question, we conducted a survey to establish the “usefulness”

(“helpfulness”) of each of our proposed categories. 16 subjects, all of who were daily computer and Internet users, took part in the survey. The average rating among subjects is shown in Table 7, where the rating “+5” indicates the most useful and “-5” indicates the most irritating or useless. A “0” score indicates indifference or neutrality. A few categories (e.g., *initialization*) were not given to subjects to rate, as their functionality is largely hidden and/or correlated with other categories.

Table 7

Usefulness ratings of our proposed functionality classes from user assessments over 16 subjects, sorted by usefulness.

Category	Rating	Std. Dev.	Category	Rating	Std. Dev.
Trivial	-	-	Cookie	+0.88	2.60
Initialization	-	-	Calendar	+0.56	3.22
Image Pre-load	-	-	Dynamic Clock	+0.25	2.49
CSS	+3.38	1.75	Interaction	+0.19	2.99
Calculator	+3.06	1.77	Client-Time based Counter	+0.13	3.10
Form Restore	+2.94	2.02	Multimedia	0	3.62
Form Validation	+2.81	2.64	Greeting	-0.10	3.14
Form Processing	+2.68	1.62	Warning Message	-0.29	2.34
Navigation	+2.63	1.89	Timer	-0.37	2.53
Last Modified Time	+2.62	1.67	Web Application	-0.38	2.53
Loading Image	+2.44	2.19	Browser Information	-0.44	2.68
Changing Image	+2.44	2.19	Static Time Information	-0.44	2.68
Visiting Browser History	+2.13	1.75	Background Color	-0.75	2.65
Page	+2.13	1.75	Static-Text Banner	-1.25	2.35
HTML	+1.88	2.28	Dynamic-Text Banner	-1.25	2.35
Plain Text	+1.88	2.28	Pop-up	-3.38	2.03

We see that the proposed JavaScript categories indeed have different utility and helpfulness to end users. Although the standard deviation (given in the second column of Table 7) is large, the average correlation between raters is .414, showing a moderate trend of our subjects to rate in the same manner. Out of all sixteen subjects, only one had a slight inverse correlation with a few other subjects.

People largely agree with each other on the usefulness of most categories. Functionality such as *pop-up*, *dynamic-text banner* and *static-text banner* are considered annoying by most users, while some others like *form processing* are considered useful. A couple of them (e.g. *background color*) are neither considered annoying nor helpful to end users, which we conclude as neutral functionality.

To settle the second issue, we needed to examine the reliability of our categorization scheme. We compiled an annotation guide which explained each of the categories and a web-based application to ask subjects to label data. Subjects were required to read the tutorial on JavaScript and the entire annotation guide before starting their annotation, and comparison notes among different categories were also provided during annotation.

As the entire 1.6K corpus is too large to expect volunteers to annotate, we randomly selected two documents to represent each class (excluding the *other* class), giving a total of  $2 \times 32 = 64$  instances for annotation. This distribution was not known to our additional annotators.

We then used the Kappa statistic (Cohen, 1960) to measure the reliability of our annotation. The kappa statistic measures the agreement between multiple raters on ordinal data, correcting for chance agreement, and is defined as  $\kappa = \frac{P(A) - P(E)}{1 - P(E)}$ , where  $P(A)$  is the proportion of times the annotators agree, and  $P(E)$  is the proportion of times they would agree by chance.  $\kappa$  ranges between 1 (perfect agreement between annotators) and 0 (meaning no agreement, but merely chance agreement). We have asked 4 people (including one of the authors) to do the annotation. The kappa for our task based on these 4 annotators is 0.607; higher than .4 which is sometimes used as the border between an ill-defined task and one that is replicable.

A closer study of the disagreements in annotation show that a few categories lead to a disproportionate amount of disagreements. General classes such as *trivial*, *interaction*, *form-processing* were often mislabeled with more specific classes. Classes dealing with the same general functionality were also sometimes confused (the different classes for time related functions, or form related functions). Despite these systematic errors, we feel that our kappa score of 0.6 indicates that the classes are largely replicable and well-defined.

## 7 Conclusion and future work

We have presented a novel approach to the problem of program categorization. In specific, we target JavaScript categorization, as its use is largely confined to a small set of purposes and is closely tied to its enclosing web page. A key contribution of our work is to create a functional categorization of JavaScript instances, based on a corpus study of over 18,000 web pages with scripts. To encourage other researchers to use our dataset as a standard reference collection, we have made our dataset, annotations and resulting system prototype freely available<sup>2</sup>.

Although previous work (Ugurel et al., 2002; Krovetz et al., 2003) has examined the use of text classification approaches to classify source code, our method is the first method that employs the source code in a non-trivial way. Different from previous work which classified code into topic categories, our work attempts to achieve a more fine-grained functional categories with less data. In this work, rather than treating the problem merely as a straightforward text categorization problem, we incorporate and adapt metrics and features that originate in program analysis. While our baseline does well, performance is greatly improved by utilizing program anal-

---

<sup>2</sup> <http://wing.comp.nus.edu.sg/~luwei/SMART/>

ysis. By the use of careful lexical analysis, static analysis and mock execution, a 52% overall reduction of categorization error is achieved. We believe this result provides strong evidence that program categorization can benefit from adapting work from program analysis. We have also validated our hypothesis that functionality based classification is feasible and useful by carrying out two human evaluations which examined our classification scheme's usefulness and replicability.

We have currently deployed our system as part of a smart JavaScript filtering system, that filters out specific JavaScript units that are irrelevant to the web page. The aim is to assist users in filtering such material and to summarize such information for users to make more informed web browsing choices. JavaScript code continues to impact the design and interactivity of the Web today in new and difficult guises (e.g., AJAX). We hope to extend our analysis further where possible to discover additional functionality to help automated indexers.

In future work, we can expand our current work to a larger dataset, adding in more entries in the hand-compiled dictionary (currently limited to 20 entries) for token normalization. We also hope to extend our techniques to subject-based classification on a wider range of computer languages.

### **Acknowledgments**

The authors sincerely thank the reviewers from the 2nd Asia Information Retrieval Symposium and *Information Processing and Management* for their valuable comments and Geunbae Lee for inviting us to publish in this special issue.

### **References**

- Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: ICSM '98: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, p. 368.
- Blazy, S., Facon, P., September 1998. Partial evaluation for program comprehension. *ACM Computing Surveys* 30 (3).
- Cohen, J., 1960. A coefficient of agreement for nominal scales. Vol. 20. pp. 27–46.
- Cory Kapser, M. W. G., 2005. Improved tool support for the investigation of duplication in software. In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 305–314.
- Hawking, D., June 2004. Web research collection. <http://es.csiro.au/TRECWeb/>.
- Kamiya, T., Kusumoto, S., Inoue, K., 2002. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.* 28 (7), 654–670.
- Kapser, C., Godfrey, M. W., 2004. Aiding comprehension of cloning through categorization. In: IWPSE. pp. 85–94.
- Kontogiannis, K., 1997. Evaluation experiments on the detection of programming

- patterns using software metrics. In: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97). IEEE Computer Society, Washington, DC, USA, pp. 44–54.
- Krovetz, R., Ugurel, S., Giles, C. L., 2003. Classification of source code archives. In: SIGIR '03: Proceedings of the 26th annual international ACM SIGIR conference on Research and development in information retrieval. ACM Press, New York, NY, USA, pp. 425–426.
- Krsul, I., Spafford, E. H., 1995. Authorship analysis: Identifying the author of a program. In: Proc. 18th NIST-NCSC National Information Systems Security Conference. pp. 514–524.
- Maletic, J. I., Marcus, A., 2000. Using latent semantic analysis to identify similarities in source code to support program understanding. In: Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00). p. 46.
- Mathias, K. S., James H. Cross, I., Hendrix, T. D., Barowski, L. A., 1999. The role of software measures and metrics in studies of program comprehension. In: ACM-SE 37: Proceedings of the 37th annual Southeast regional conference (CD-ROM). ACM Press, New York, NY, USA, p. 13.
- Rowe, N., Laitinen, K., 1995. Semiautomatic disabbreviation of technical text. *Information Processing and Management* 31 (6), 851–857.
- Ugurel, S., Krovetz, R., Giles, C. L., Pennock, D. M., Glover, E. J., Hongyuan, Z., 2002. What's the code? automatic classification of source code archives. In: Proceedings of ACM SIGKDD Conference on Knowledge and Data Discovery.
- von Mayrhauser, A., Vans, A. M., November 14 1994. Dynamic code cognition behaviors for large scale code. In: Proceedings of the 3rd Workshop on Program Comprehension. Washington, D.C., USA, pp. 74–81.
- Witten, I. H., Frank, E., 2000. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco.
- Wong, W.-C., Fu, A. W.-C., May 14 2000. Finding structures of web documents. In: ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD). Dallas, USA.
- Yang, W., 1991. Identifying syntactic differences between two programs. *Software - Practice and Experience* 21 (7), 739–755.