Appears in Asian Information Retrieval Symposium (AIRS) 2005. Copyright (C) Springer-Verlag. http://www.springer.de/comp/lncs/index.html

Supervised Categorization of JavaScript™ using Program Analysis Features

Wei Lu and Min-Yen Kan

Department of Computer Science,
School of Computing,
National University of Singapore,
Singapore, 117543
{luwei, kanmy}@comp.nus.edu.sg

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.

1 Introduction

Current generation web pages are no longer simple static texts. As the web has progressed to encompass more interactivity, form processing, uploading, scripting, applets and plug-ins have allowed the web page to become a dynamic application. While a boon to the human user, the dynamic aspects of web page scripting and applets impede the machine parsing and understanding of web pages. Pages with JavaScript, Macromedia Flash and other plug-ins are largely ignored by web crawlers and indexers. When functionality is embedded in such web page extensions, key metadata about the page is often lost. This trend is growing as more web content is provided using content management systems which use embedded scripting to create a more interactive experience for the human user. 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 page interactivity. Although understanding embedded applets and plug-ins are also important, 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).

An automatic categorization of JavaScript assists an indexer to more accurately model web pages' functionality and requirements. Pop-up 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.

To perform this task, we introduce a machine learning framework that draws on features from text categorization, program comprehension and code metrics. We start by developing a baseline system that employs traditional text categorization techniques. We then show how the incorporation of features that leverage knowledge of the JavaScript language together with program analysis, can improve categorization accuracy. We conduct evaluation of our methods on the widely-used WT10G corpus [4], used in TREC research, to validate our claims and show that the performance of our system eliminates over 50% of errors over the baseline.

In next section, we examine the background in text 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 experimental setup and analysis and conclude by discussing 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, which has been largely ignored by text categorization research.

Ugurel *et al.*'s [11] work is 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 [14] 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 has developed into many subfields, in which formal methods are favored over approximation. The subfield of program comprehension develops models to explain how human software developers learn and comprehend existing code [9]. These models show that developers use both top-down and bottom-up models in their learning process [12]. Top-down models imply that developers may use a model of program execution to understand a program. Formal analysis via code execution [2] may yield useful evidence for categorization.

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 [3, 5, 7], as JavaScript instances are often copied and 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 [8]. The key contribution of our work shows that the converse is also true: program analysis assists in text 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 mostly confined to web pages and performs a 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. This property makes the text categorization approach well-defined, in contrast to categorization of programs in other programming languages. 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 a document object model (DOM), affect categorization performance.

Ugurel *et al.* [11] proposed 11 topic categories for source code topic classification task, including *circuits*, *database*, and *development*. They have 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.

We examined an existing JavaScript categorization from a well-known tutorial site, *www.js-examples.com*. 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 intended for the developer, rather than the consumer. Examples that have similar effects are often categorized differently as the implementation uses different techniques. In contrast, we intend to categorize JavaScript functionality with respect to the end user. Second, the classification is

Category	Description (# units in corpus)
Dynamic-Text Banner	Displays a banner that changes content with time (264)
Initialization	Initialize/modifies variables for later use (123)
Form Processing	Passing values between fields, or computing values from form fields (119)
Calculator	Displays and manipulates a calculator (88)
Image Pre-load	Pre-load images for future use (87)
Pop-up	Pops up a new window (80)
Changing Image	Change the source of an image (79)
HTML	Generate HTML components, such as forms (68)
Web Application	Web applications such as games & e-commerce (62)
Background Color	Change or initialize the background color (50)
Form Validation	Validate a forms data fields (50)
Page	Load a new page to the browser window (49)
Plain Text	Print some text to the page (46)
Multimedia	Load multimedia (43)
Static-Text Banner	Displays a banner which does not change with time (42)
Static Time Information	Display static system time (41)
Loading Image	Load and display images (39)
Form Restore	Restore form fields to default values (37)
Server Information	Display page information from the server (36)
Dynamic Clock	Display a clock that changes with system time (35)
Navigation	Site navigator (32)
Browser Information	Check browser information (32)
Cookie	Store or retrieve data on server about client (26)
Trivial	Perform a simple one-liner task (24)
Interaction	User interaction with the page (24)
Warning Message	A static warning message (16)
Timer	Display a timer which is running (10)
Greeting	Display a greeting to the user (10)
CSS	Change the Cascading Style Sheet of the page (7)
Client-Time based Counter	Display interval between current time and another time relevant to page (7)
Visiting Browser History	Visit a page from the browser's history (6)
Calendar	Display a calendar (5)
Others	Multiple functionality or too few instances (133)

Table 1. JavaScript functional categories, sorted by frequency. Number of instances indicated in parenthesis in the description field.

used for example scripts, which are usually truncated and for illustrative purposes. We believe that their classification would not reflect actual JavaScript embedded on web sites.

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 use the WT10G corpus, commonly used in web IR experiments, as the basis for our work. In the WT10G corpus, we see that JavaScript that natively occurs in actual web pages are different and more difficult to handle. 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.

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 of (potentially) called subprocedures. Any

HTML involved in the triggering of the unit is also included. Figure 1 shows an example.

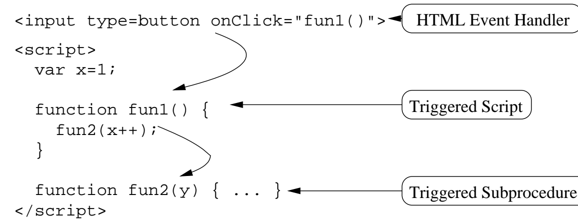

Fig. 1. A JavaScript unit, the basic element used for our classification.

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

4 Methods

Given such a categorization, a standard text categorization approach would tokenize pre-classified input 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 would be inferred as its category.

A simple approach to categorization uses a compiler's own tokenization, treating the resulting tokens as separate dimensions for categorization. An n dimensional feature vector results, where n is the total number of unique tokens that occur in all training unit instances.

We improve on this text categorization baseline in three ways. We first show how tokenization can be improved by exploiting the properties of the 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, which in turn influence an unit's classification.

4.1 Using language features for improved tokenization

A syntactic analysis of a programming language is instructive as it helps to type the program's tokens. After basic compiler-based tokenization, we distinguish the tokens

of each unit as to whether they are numeric constants, string constants, operators, variable and method names, or language-specific reserved keywords, or part of comments. As JavaScript draws from Java and web constructs, we further distinguish regular expression operators, URLs, file extensions images and multimedia, HTML tags and color values. Tokens of these types are tagged as such and their aggregate type counts are used as features for categorization.

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

To a machine learner, the tokens `currMon` and `curMonth` are unrelated. To connect these forms together, we need to normalize these non-standard words (NSW) to resolve this feature mismatch problem [10]. 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). A following expansion phase is carried out, in which commonly abbreviated shortenings are mapped to the word equivalents (*e.g.* "`curr`" and "`cur`" → "`current`") using a small (around 20 entries) hand-compiled dictionary .

Example	Transition Pattern	Result (with expansion)
`curMsg`	single lowercase $\rightleftharpoons$ single uppercase	`current message`
`IPAddress`	consecutive uppercase → lowercase	`ip address`
`thisweek`	no transition and length ≥ 6	`this week`

Table 2. Examples of Name token normalization.

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 [3, 5] 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). As such, we start with simple, classic complexity metrics to assess their impact on categorization. Examples of them are:

Cyclomatic Complexity (CC) Cyclomatic complexity is a widely used control flow complexity metric. 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.

Number of Attributes (NOA) is a data flow metric that counts the number of fields declared in the class or interface. In JavaScript, it counts the number of declared variables and newly created objects in the source code.

Informational fan-in (IFIN) is an information flow metric, 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. This metric is traditionally defined for class and interfaces, constructors and methods.

We also developed several metrics based on our observation of JavaScript instances in our corpus. These metrics count language structures that we found were prevalent in the corpus and may be indicative of certain program functionality.

Similar Statements (SS) counts the number of statements with similar structure.

Built-in Object References (BOR) counts the number of built-in objects (*e.g.* `date`, `window`) referenced by the unit.

In these metrics, similarity is determined by using a simple tree edit distance model based on the syntax of the language, discussed next.

4.2.1 Code Reuse using Edit Distance Aside from complexity metrics, 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.

`if (x > 1) {`
`alert ("hi");`
`}`
	`alert ("hi")`	**SED**
		there is common subsequence

BLOCK ├ **IFNE** │ └ **GT** │ ├ **NAME** x │ └ **NUMBER** 1.0 └ **BLOCK** └ **STMT** └ **CALL** ├ **NAME** alert └ **STRING** hi	**STMT** └ **CALL** ├ **NAME** alert └ **STRING** hi	**TED** different roots results in large edit distance
if**[KEY]** (**[SYM]** x**[VAR]** >**[SYM]** 1**[NUM]**)**[SYM]** {**[SYM]** alert**[VAR]** (**[SYM]** "hi"**[STR]**)**[SYM]** ;**[SYM]** }**[SYM]**	alert**[VAR]** (**[SYM]** "hi"**[STR]**)**[SYM]**	**LED** more reasonable and accurate than SED

Fig. 2. String based edit distance (SED), tree based edit distance (TED) and lexical-token based edit distance (LED)

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

In program analysis, abstract syntax trees (ASTs) [1] are often used to model source code and correct for these discrepancies. An AST is a parse tree representation of the source code that model 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 [15]. However, as is shown in Figure 2, the given two code fragments are of the same functionality, but have different syntactic structures. Hence, syntactic difference does not 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. Aside from the tree-based edit distance measure, we can also measure similarity from a lexical-token approach (lexical-token based edit distance, LED) [6], in which source codes are parsed into a stream of lexical-tokens, and these tokens become the elements for computation. 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

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

Table 3. Units that reference their HTML context.

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 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)[1]. 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

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

JavaScript are DOM objects whose data type can only be inferred by examining the enclosing HTML document. Table 3 illustrates two examples where the script references DOM objects.

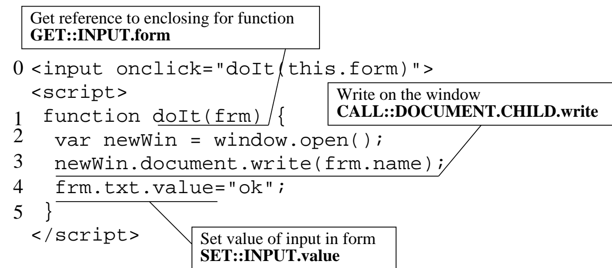

Fig. 3. Types of DOM object references.

We classify references to DOM objects into three categories: *get*s, *set*s, and *call*s. These are illustrated in the JavaScript unit in Figure 3: 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`", 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.

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

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 4.2.1. A variable and its data type form a single unified token (*e.g.* newWin↦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.

4.3.2 Dynamic analysis Static analysis is not able to recover certain information that occurs at run time. Dynamic analysis (*i.e.*, execution of code) can extract helpful features along the single, 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.

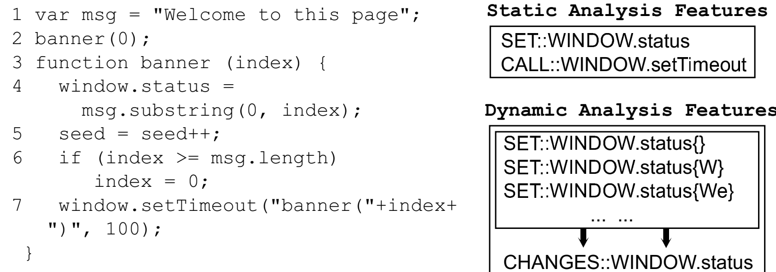

Fig. 4. Sample JavaScript unit (l), along with features extracted by static and dynamic analysis (r).

We illustrate how dynamic analysis can yield additional features for categorization in Figure 4. This sample JavaScript unit, taken from the WT10G corpus, 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 makes the banner text in the window change over time. Without dynamic analysis, we cannot recover what value `msg.substring(0,index)` refers to. More importantly, dynamic analysis allows us to extract the value of the expression `''banner(''+index+'')''`. In this example, dynamic analysis also recovers the fact that the variable's value is changing, hence a new feature is added to the feature set (*i.e.* CHANGES::WINDOW.status).

5 Evaluation

We tested the above methods on the WT10G corpus, containing approximately 1.7 M web pages from over 11K distinct servers. After pre-processing and cleaning of the WT10G corpus, over 18 K pages contained processable JavaScript scripts units. String identical duplicates and structurally-identical script units were then removed. This resulted in a final corpus of 1,637 units, which are unique in textual form and structure. The high ratio of the number of script instances to unique scripts validates our claim that many scripts are simply clones.

We perform supervised text categorization using a support vector machine approach (SVM). SVMs were chosen as the machine learning framework as they handle high-dimensional datasets efficiently. This is extremely important as feature vectors contain anywhere from 3,000 to 8,000 features, depending on which feature sets are used in the

model configuration. Specifically, we used the generic SVM algorithm (SMO) provided with WEKA [13]. We use a randomized, ten-fold cross validation of the final corpus of 1,637 script units, which excludes the *other* category. Instance accuracy is reported in the results. Due to space limitations, we report instance accuracy which has been used in previous work [11] and have omitted other IR metrics such as precision and recall.

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.

Features used	Accuracy	ER
Most frequent class baseline	16.12%	–
Text categorization baseline	87.47%	–
L. All lexical analysis	89.61%(**)	17%
L_c. Language token counting	88.57%	8%
L_n. Function/variable normalization	87.66%	1%
M. All software metrics	77.76%	–
M_s. Standard classic metrics	20.46%	–
M_j. w/ new metrics (M_s+SS+BOR)	25.60%	–
M_e. String-based edit distance	73.85%	–
M_a. AST-based edit distance	72.69%	–
M_t. Token-based edit distance	74.89%	–
P. All program comprehension	87.29%	–
P_s. Static analysis	79.78%	–
P_d. Dynamic analysis	71.22%	–
L+M	90.04%(*)	21%
L+P	92.36%(**)	39%
L+M+P	93.95%(**)	52%

Table 4. Component Evaluation Results. Error reduction (ER) is measured against the text categorization baseline. (*) indicates the improvement over the approach using previous feature set is statistically significant at 0.05 level under T-test, (**) indicates statistically significant at 0.01 level

Table 4 shows the component evaluation in which we selected certain combination of features as input to the SVM classifier. Here, we can see the majority class categorizer 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, 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 classifiers. However, when coupled with a strong lexical feature component, we show improvement.

The performance improvement may seem marginal, but in fact they are statistically significant, as demonstrated by the use of a one-tailed t-test. We believe significance is

achieved due to the large scale of the evaluation set's degrees of freedom present in the classification problem.

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 gains in performance do not make advances in these problems less 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.

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. Less effective is when variable and function names are normalization through splitting and expansion. When both techniques are used together, their synergy improves performance, 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 perform 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 serves to augment its performance.

A Note on Efficiency. The experiments in this paper were conducted on a single, modern desktop machine with two gigabytes of main memory. In general, feature creation is fast, for all 1.6K script instances in our corpus took approximately 3 minutes, and a 10-fold cross validation of the SMO classifier takes about 10 minutes. The exception to the feature creation is when edit distance-based code reuse metrics were computed. These features are computed in a brute-force, pairwise manner and took up to ten hours to generate. We are currently looking into faster approaches that may lower the complexity of the approach.

6 Shortcomings

Our results are promising, but we would like to call attention to some of the shortcomings of our work that we are currently addressing:

Annotator Agreement. Our corpus is annotated by one of the paper authors. While this provides for consistency, the annotator notes that some instances of problematic, even for a language whose applications are largely distinct. We feel this a source of some errors and are working on further annotation and finding inter-annotator agree-

ment. A reasonable upper bound of performance may be less than 100%, meaning that our performance gains may be more signficant than discussed in this paper.

Dynamic Analysis Incompleteness. Many tasks are executed conditionally depending on the browser's type. In our dynamic analysis, we assume scripts are only executed under as MSIE 4.0, which causes certain analyses to fail to extract data. As browser checking is ubiquitous in JavaScripts, we may relax this constraint and follow all execution pathways that are conditional on the browser.

7 Conclusion and Future Work

We present 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 our researchers to use our dataset as a standard reference collection, we have made our dataset, annotations and resulting system freely available[1].

Although previous work [11] 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 adapted metrics and features that originate in program analysis. Our corpus study confirms that many such scripts are indeed copies or simple modifications. While our baseline does well, performance is greatly improved by utilizing program analysis. By careful lexical analysis, 10% of errors are eliminated. Further improvements using static analysis and execution results in a 52% overall reduction of categorization error. We believe they provide evidence that program categorization can benefit from adapting work from program analysis.

We currently deploy our system as part of a smart JavaScript filtering system, that filters out specific JavaScript units that have functionality irrelevant to the web page (*e.g.* banner, pop-up). We plan to extend this work to other scripting languages and decompiled plug-ins appearing on web pages. The aim of such work is to assist end users to filter irrelevant material and to summarize such information for users to make more informed web browsing a wider variety of classification (including subject-based classification) on a wider range of computer languages in future work.

References

1. Baxter, I. D.; Yahin, A.; Moura, L. M. D.; SantAnna, M.; and Bier, L. 1998. Clone detection using abstract syntax trees. In ICSM, 368–377.
2. Blazy, S., and Facon, P. 1998. Partial evaluation for program comprehension. ACM Computing Surveys 30(3).

[1] `http://wing.comp.nus.edu.sg/~luwei/SMART/`

3. Cory Kapser and Michael W. Godfrey. Aiding Comprehension of Cloning Through Categorization. Proc. of 2004 International Workshop on Software Evolution (IWPSE-04), Kyoto, Japan, 2004.
4. D. Hawking. Web Research Collection. http://es.csiro.au/TRECWeb/, June 2004.
5. I. Krsul and E. H. Spafford. Authorship Analysis: Identifying the Author of a Program. Proc. 18th NIST-NCSC National Information Systems Security Conference, 514–524, 1995.
6. 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), 2002, 654–670.
7. 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), 44–54. Washington, DC, USA: IEEE Computer Society.
8. Maletic, J. I., and 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 In- telligence (ICTAI00), 46.
9. Mathias, K. S.; II, J. H. C.; Hendrix, T. D.; and Barowski, L. A. 1999. The role of software measures and metrics in studies of program comprehension. In ACM Southeast Regional Conference.
10. Rowe, N., and Laitinen, K. 1995. Semiautomatic disabbreviation of technical text. Information Processing and Management 31(6):851–857.
11. S.Ugurel, B.Krovetz,C.L.Giles,D.Pennock,E.Glover,H.Zha. What is the code? Automatic Classification of Source Code Archives. Eighth ACM International Conference on Knowledge and Data Discovery (KDD 2002), 623–638 (poster), 2002.
12. von Mayrhauser, A., and Vans, A.M. 1994. Dynamic code cognition behaviors for large scale code. In Proceedings of the 3rd Workshop on Program Comprehension, 74–81.
13. Witten, I. H., and Frank, E. 2000. Data Mining: Practical machine learning tools with Java implementations. San Francisco: Morgan Kaufmann.
14. Wong,W.-C., and Fu, A.W.-C. 2000. Finding structures of web documents. In ACM SIGMOD Workshop on Research Issues in DataMining and Knowledge Discovery (DMKD).
15. Yang, W. (1991). Identifying syntactic differences between two programs. Software - Practice and Experience, 21(7), 1991, 739–755.