Honours Year Project Report

Supervised Categorization of JavaScript^{TM} using Program Analysis Features

By

Lu Wei

Department of Computer Science

School of Computing

National University of Singapore

2004/2005

Honours Year Project Report

Supervised Categorization of JavaScript™ using Program Analysis Features

By

Lu Wei

Department of Computer Science

School of Computing

National University of Singapore

2004/2005

Project No: H079060
Advisor: Dr. Min-Yen Kan
Deliverables:
- Report: 1 Volume
- Source Code: 1 CD
- Data Corpus: 1 CD

Abstract

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

Subject Descriptors:
H.3.3 Information Search and Retrieval
F.3.2 Semantics of Programming Languages
D.2.8 Metrics

Keywords:
Information Retrieval, Machine Learning, JavaScript, ECMAScript, Program Comprehension, Source Clone, Program Pattern, Software Metrics.

Implementation Software and Hardware:
IBM PC, Linux, Java2 SDK 1.5.0, perl 5.004

Acknowledgement

First of all, I would like to express my deep gratitude to my advisor Dr. Min-Yen Kan, for his suggestions and patience on advising me in this project. He is a good advisor who is always willing to listen to his students' own ideas and encourage their independence in doing research. He is very easy to approach and has increased my interest in the subject of Information Retrieval and Natural Language Processing.

I am also very thankful to Dr. Bimlesh Wadhwa for kindly sharing her knowledge in software metrics and providing suggestions on this project.

I would like to thank other fellow members in the WING[1] group, especially members from the evaluation sub-group for their help on my surveys and evaluations.

I would also like to thank Mr. Tan Yee Fan for reviewing and providing valuable comments on my report.

Last but not least, I would like to thank my parents and my friends for their love, encouragement and patience throughout my studies.

[1] http://wing.comp.nus.edu.sg

List of Figures

3.1 Example of JavaScript functional unit 7

4.1 The big picture of my system 10
4.2 Example of JavaScript tokenization and type-binding 12
4.3 Algorithm NSWSplit(splitting of non-standard word for variable token normalization) 13
4.4 Example of JavaScript parse tree 15
4.5 Example of syntax features 15
4.6 Algorithm for syntax feature retrieval 16
4.7 Example of object function 19
4.8 Tree edit distance(TED) algorithm does not work better than string edit distance(SED) algorithm for my task 20
4.9 Examples of static object communication feature retrieved 24
4.10 Similar codes performing different tasks 25
4.11 Sample JavaScript unit (l), along with features extracted by static and dynamic analysis (r). 26

6.1 Problematic instance in annotation 34

A.1 JavaScript and its enclosing HTML page A-2

B.1 Screen shot for the JavaScript Helpfulness survey (http://wing.comp.nus.edu.sg/ luwei)B-2

D.1 The tree-edit distance(TED) algorithm used in my experiment D-1

List of Tables

3.1 JavaScript functional categories, sorted by frequency. Number of instances indicated in parenthesis in the description field. As well as usefulness(Rate) collected from users. Agreement(standard deviation, Agree) over users. Conclusion of its usefulness(Useful),“NTRL” stands for “neutral” representing a neutral functional category(more details to be found in appendix B). 8

4.1 Variable token normailzation-splitting phase 12
4.2 Variable token normalization - expanding phase 13
4.3 String Token Normalization 14
4.4 Edit cost scheme for lexical-token based edit distance(LED) algorithm 21
4.5 Units that reference their HTML context 23
4.6 Partial table describing DOM object’s references traced in object communication analysis 24
4.7 How the features look like 26

5.1 Unique JavaScript unit v.s. unique server distribution, in WT10G corpus 27
5.2 Evaluation on lexical analysis, Error reduction (ER) is measured against the text categorization baseline. 28
5.3 Evaluation on syntax analysis 29
5.4 Evaluation on code metrics analysis 30
5.5 Evaluation on context sensitive analysis 31
5.6 All components 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 32

Table of Contents

Title i

Abstract ii

Acknowledgement iii

List of Figures iv

List of Tables v

1 Introduction 1
- 1.1 Defining the Problem 1
- 1.2 Our Solution 2
- 1.3 Report Organization 2

2 Background 3
- 2.1 Text Categorization 3
- 2.2 Program Analysis, Program Comprehension, and Software Metrics 4

3 Categorization 6

4 Methods 9
- 4.1 Context Free Analysis 10
  - 4.1.1 Lexical Analysis 10
  - 4.1.2 Syntax Analysis 14
  - 4.1.3 Code Metrics Analysis 16
- 4.2 Context Sensitive Analysis 22
  - 4.2.1 JavaScript Unit Property Analysis 22
  - 4.2.2 Object Communication Analysis 23
- 4.3 A Tiny Example 26

5 Evaluation 27
- 5.1 Evaluation on Each Component 28
  - 5.1.1 Baseline 28
  - 5.1.2 Evaluation on Context Free Analysis 28
  - 5.1.3 Evaluation on Context Sensitive Analysis 31
- 5.2 Evaluation on All Components 31

6 Conclusion **33**
6.1 Limitations 33
6.2 Contributions 34
6.3 Future Work 35

References **36**

A JavaScript Functional Unit **A-1**
A.1 Interactive Unit A-1
A.2 Non-interactive Unit A-1

B Survey on JavaScript Usefulness **B-1**

C Token Table **C-1**

D The AST-based Edit Distance Algorithm Used **D-1**

E A Paper on this Work to AAAI Conference **E-1**

Chapter 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(*e.g.* Google[1]). 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.

1.1 Defining the Problem

To address the above discussed problem, I 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, I chose to focus on JavaScript as 1) its code is inlined within an

[1] http://www.google.com

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(Kushmerick, 1999), 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.

1.2 Our Solution

To perform this task, I introduce a machine learning framework that draws on features from text categorization as well as program analysis techniques including lexical analysis, syntax analysis, code metrics, and program comprehension[1]. I start by developing a baseline system that employs traditional text categorization techniques. I then show how the incorporation of features that leverage knowledge of the JavaScript language together with program analysis, can improve categorization accuracy. I conduct evaluation of my methods on the widely-used WT10G corpus, used in TREC[2] research, to validate my claims and show that the performance of my system eliminates over 50% of errors over the baseline.

1.3 Report Organization

In chapter 2, I examine the background in text categorization and discuss how features of the JavaScript language and techniques in program analysis can assist in categorization. In chapter 3, I present my proposed categorization scheme based on the WT10G corpus. In chapter 4, I present my methods that distills features for categorization from the principles of program analysis. In chapter 5, I describe my experimental setup and analysis. Finally, I conclude my work in chapter 6.

[1] Although these methods are different in terms of formal definition, I collectively refer to them as "program analysis" techniques in this thesis for readability

[2] Text REtrieval Conference:http://trec.nist.gov/

Chapter 2

Background

A survey of previous work shows that the problem of automated computer software categorization is relatively new. I 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. I feel that these two fields have a natural overlap which can be exploited.

2.1 Text Categorization

Text categorization assigns a document d one or more labels l from a predefined set of categories C, $l \epsilon C$. In many categorization scenarios, a hard categorization is enforced (*i.e.*, only one category can be assigned to a document), which allows standard information retrieval evaluation metrics of precision, recall and F_1 to be employed. A basic TC approach views a text as a collection of words, where each word represents partial evidence of the text's category. A vector space model (VSM) of dimension $|V|$ (the size of the vocabulary used in the corpus) can be used in which words are weighted according to a scheme, such as term frequency $\times$ inverse document frequency (TF$\times$IDF). TC research is dominated by research in tuning these

parameters in the general architecture: classic papers in TC have examined various weighting schemes (Salton & Buckley, 1988), learning paradigms (Yang & Liu, 1999), and methods that can deal with or reduce the dimensionality of the dataset (Koller & Sahami, 1997; Deerwester, Dumais, Landauer, Furnas, & Harshman, 1990). As such, the focus of related work in TC has been preoccupied with vocabulary approaches and not with the structure of the text itself. Usually text processing is limited to pre-processing. These tasks usually include the removal of stopwords, extraneous punctuation (*e.g.*, HTML tags), and word stemming.

Unlike natural language texts, program source code exhibits no ambiguity 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. Recent work in categorizing web pages (Wong & 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.

2.2 Program Analysis, Program Comprehension, and Software Metrics

Program analysis is the study of analysis of programs to determine their results and effectiveness. Traditional approaches to program analysis includes *Data Flow Analysis*, *Constraint Based Analysis*, *Abstract Interpretation*, *Type and Effect Systems* and so on(Nielson, Nielson, & Hankin, 1999). Recently, program analysis has developed into many subfields and represents the foundation of many other software analysis tasks such as reverse engineering, program comprehension and maintenance, in which formal methods are favored over approximation. Thomas Panas(Panas, 2003) gives a description of the program analysis process used in a reverse engineering and program comprehension scenario. The field of program comprehension develops models to explain how human software developers learn and comprehend existing code (Mathias, II, Hendrix, & Barowski, 1999). These models show that developers use both top-down and bottom-up models in their learning process(von Mayrhauser & Vans, 1994). Top-down models

imply that developers may use a model of program execution to understand a program. Formal analysis via dynamic analysis (Blazy & Facon, 1998; Gschwind, Oberleitner, & Pinzger, 2003) may yield useful evidence for categorization.

Program comprehension also employs software metrics, which are units of statistical measurements of certain software properties, such as complexity and efficiency performance. Variants of metrics are *e.g.* size metrics, complexity metrics(including information flow metrics), understandability metrics, object-oriented metrics and so on[1]. Of particular interest to my problem scenario are code reuse metrics(Kontogiannis, 1997), as JavaScript instances are often copied and modified from standard examples. In my experiments, I assess the predictive strength of these metrics on program categories.

I believe that a standard text categorization approach to this problem can be improved by adopting features distilled from program analysis. Prior work shows that certain IR techniques such as latent semantic analysis(LSA), can aid program comprehension and maintenance(Maarek, Berry, & Kaiser, 1991; Maletic & Marcus, 2000; I.Maletic & Marcus, 2001). However, to my best knowledge, the converse, how program analysis can assist text categorization, as an issue did not receive much attention. The key contribution of my work provides evidence that certain program analysis techniques do assist in text categorization.

[1]Metrics Basics:http://www.aivosto.com/project/help/pm-basics.html

Chapter 3

Categorization

JavaScript is mostly confined to web pages and performs a limited number of tasks. I 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 more 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 characteristics of the web page are controlled and manipulated in JavaScript.

I examined an existing JavaScript categorization scheme 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 my 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, I 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. I believe that their classification would not reflect actual JavaScript embedded on web sites.

To deal with these shortcomings, I decided to adapt the *js-examples.com* scheme based on

JavaScript instances in actual web documents. I use the WT10G corpus, commonly used in web IR experiments, as the basis for the work discussed in this paper.

In the WT10G corpus, I 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 objects. 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.

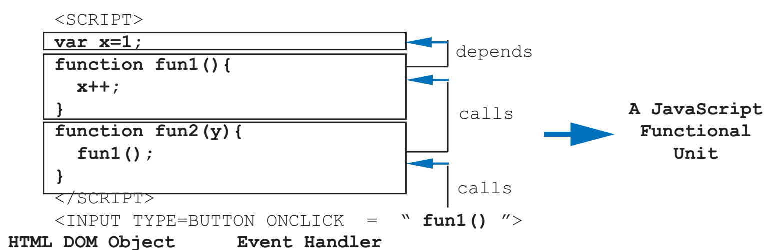

Figure 3.1: Example of JavaScript functional unit

As such, I 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 and involved statements. Any possible HTML object involved in the triggering of the *unit* is also included. Further more, a *functional unit* can either be *interactive* or *non-interactive*, depending on whether the *unit* is triggered by any event handler or not. A *unit* is an abstract representation of the JavaScript instance used in my classification task, which is constructed dynamically at run-time by my system. Figure 3.1 is a simple example of a JavaScript *functional unit*(more specifically, it is an *interactive unit*). For a detailed description of JavaScript *functional unit*, please refer to appendix A.

I base my categorization of JavaScript on these automatically extracted *units*. Based on my corpus study, I propose a modified classification of JavaScript into 33 discrete categories, shown

in table 3.1. 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. This table also includes statistics showing the "usefulness" level for each category. These data are collected from a survey over 25 web surfers. For a detailed discussion about the survey, please refer to appendix B.

Category	Description(# of unique instances in corpus)	Rate	Agree	Useful
Dynamic-Text Banner	Displays a banner which does not change with time (264)	-1.25	2.35	NO
Initialization	Initialize/modifies variables for later use (121)	-	-	-
Form Processing	Passing values between fields, or computing values from form fields (119)	+2.68	1.62	YES
Calculator	Displays and manipulates a calculator (88)	+3.06	1.77	YES
Image Pre-load	Pre-load images for future use (87)	-	-	-
Pop-up	Pops up a new window (79)	-3.38	2.03	NO
Changing Image	Change the source of an image (79)	+2.44	2.19	YES
HTML	Generate HTML components, such as forms (68)	+1.88	2.28	YES
Web Application	Web applications such as games & e-commerce (62)	-0.38	2.53	NTRL
Background Color	Change or initialize the background color (50)	-0.75	2.65	NTRL
Form Validation	Validate a forms data fields (50)	+2.81	2.64	YES
Page	Load a new page to the browser window (49)	+2.13	1.75	YES
Plain Text	Print some text to the page (46)	+1.88	2.28	YES
Multimedia	Load multimedia (43)	0	3.62	NTRL
Static-Text Banner	Displays a banner that changes content with time (42)	-1.25	2.35	NO
Static Time Information	Display static system time (41)	-0.44	2.68	NTRL
Loading Image	Load and display images (39)	+2.44	2.19	YES
Form Restore	Restore form fields to default values (37)	+2.94	2.02	YES
Server Information	Display page information from the server (36)	+2.62	1.67	YES
Dynamic Clock	Display a clock that changes with system time (35)	+0.25	2.49	NTRL
Navigation	Site navigator (33)	+2.63	1.89	YES
Browser Information	Check and display browser information (32)	-0.44	2.68	NTRL
Cookie	Store or retrieve data on server about client (26)	+0.88	2.60	NTRL
Trivial	Perform a simple one-liner task (26)	-	-	-
Interaction	User interaction with the page (24)	+0.19	2.99	NTRL
Warning Message	Display or pop up a warning message (16)	-0.29	2.34	NTRL
Timer	Display a timer which is running (10)	-0.37	2.53	NTRL
Greeting	Display a greeting to the user (10)	-0.10	3.14	NTRL
CSS	Change the Cascading Style Sheet of the page (7)	+3.38	1.75	YES
Client-Time based Counter	Display interval between current time and another time relevant to page (7)	+0.13	3.10	NTRL
Visiting Browser History	Visit a page from the browser's history (6)	+2.13	1.75	YES
Calendar	Display a calendar (5)	+0.56	3.22	NTRL
Others	Multiple functionality or too few instances (133)	-	-	-

Table 3.1: JavaScript functional categories, sorted by frequency. Number of instances indicated in parenthesis in the description field. As well as usefulness(Rate) collected from users. Agreement(standard deviation, Agree) over users. Conclusion of its usefulness(Useful), "NTRL" stands for "neutral" representing a neutral functional category(more details to be found in appendix B).

Chapter 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 the test program's category.

A simple approach to categorization would use white space and punctuations as delimiters, and treat the resulting tokens as separate dimensions for categorization. This results in a feature vector of n-dimensions, where n is the total number of unique tokens that occur in all training *unit* instances.

I attempt to improve on this baseline model in several approaches. These approaches can be grouped into two major classes: **context free analysis** and **context sensitive analysis**.

In context free analysis, I first investigate how tokenization can be improved by exploiting the properties of the language in terms of *lexical analysis* and *syntax analysis*. Next, I investigate the use of code metrics as features to help the categorization task. As for context sensitive analysis, I attempt to distill advanced features with the JavaScript *unit*'s enclosing HTML page serving as a "context". Specifically, I present two novel approaches in this part, I firstly investigate whether features related to a JavaScript *unit*'s property appearing on the HTML page can be retrieved, and then present my proposed *object communication analysis* in two phases, namely, static analysis and dynamic analysis.

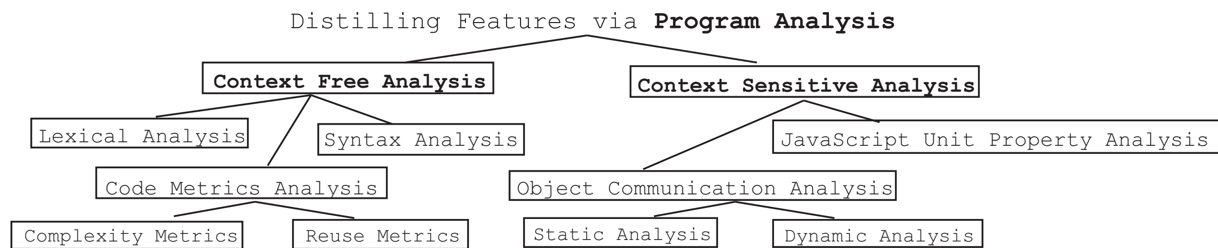

Figure 4.1: The big picture of my system

Each of the above discussed technique yields a different set of features, either a bag of tokens, or numeric values. A combination of these features will be passed to machine learner for classification. Figure 4.1 shows the big picture of my entire work.

4.1 Context Free Analysis

A context free analysis refers to an analysis process regardless of the contextual information associated to the particular program at hand. In JavaScript the contextual information includes built-in function calls and objects, as well as the enclosing web page of the *unit*. In this section, I present my context free analysis in three approaches, namely, *lexical analysis*, *syntax analysis*, and *code metrics analysis*.

4.1.1 Lexical Analysis

Traditional Tokenization Scheme for Free Text

In conventional free-text categorization tasks, word streams are separated by delimiters like white space and punctuations when constructing features. Although it works reasonably well for free-text, this traditional tokenization scheme exhibits many problems when being applied to texts like source codes. Let's take the following JavaScript code fragment from WT10G corpus for example:

```
var out+=msg; seed--; var cmd=''scrollit_r2l(''+seed+'')'';
var info = ''>>>>Welcome To Eric's Tourism Information Page <<<<<<''
```

When traditional tokenization scheme is applied, the statements "`out+=msg; seed--`" will be simply tokenized as "`out`", "`+`", "`=`", "`msg`", "`;`", "`seed`", "`-`", "`-`", and "`;`". In such a manner, the single atomic operators "`+=`" will be wrongly tokenized as separate tokens "`+`", "`=`", rather than a whole, and similarly for the atomic operator "`--`". Therefore, information conveyed by the operator "`+=`" "`--`" is lost, while information conveyed by operators like "`+`", "`=`" and "`-`" will be wrongly added as feature representing the instance. Similarly, in second line of the above example, although symbols "`>>>>`" in the statement appears inside a string, the tokenizer makes no distinction between them and the mathematical operator "`>`"(greater than).

As one can observe, the simple tokenization scheme, while works well for free-text categorization, does not serve as a good scheme for source code categorization. I next investigate how a more reasonable tokenization scheme can be introduced by a careful study of the nature of JavaScript language.

Compiler-based Tokenization and Type-binding

The task of the lexical analyzer(also known as lexer) is to look for patterns in a source program in order to convert them to a sequence of symbols, so-called tokens. A syntax-directed lexical analysis is especially instructive as it helps to type the program's tokens. Based on this idea, I therefore employ a compiler-based tokenization scheme. I tokenize word sequence into reasonable word units based on the parsing result of JavaScript language. I distinguish the tokens of each unit as to whether they are numeric constants, string constants, operators, variable and method names, regular expressions, language-specific reserved keywords, or part of comments. Just like in natural language processing tasks people have Part-Of-Speech(POS) tags representing types for each token, here I conclude JavaScript lexical tokens into the following types: *Comment Token*, *Variable Token*, *Number Token*, *String Token*, *Regular Expression Token*, *Keyword Token*, and *Symbol Token*. Figure 4.2 gives examples of types tagged to respective tokens. For more details about JavaScript token types, please refer to appendix C. Tokens of these types are tagged as such and their aggregate type counts are used as features

for categorization.

```
var x = 1;            var[KEY] x[VAR] =[SYM] 1[NUM] ;[SYM]
x *= 1.23e4;          x[VAR] *=[SYM] 1.23e4[NUM] ;[SYM]
alert("x="+x);        alert[VAR] ([SYM] "x="[STR] +[SYM] x[VAR] )[SYM] ;[SYM]
x++//cmt              x[VAR] ++[SYM] //cmt[CMT]
var re = /mac/i;      var[KEY] re[VAR] =[SYM] /mac/i[REG] ;[SYM]
--------------------------------------------------------------------------------
KEY:keyword; VAR:variable; SYM:symbol; NUM:number; STR:string; CMT:comment; REG:regexp
```

Figure 4.2: Example of JavaScript tokenization and type-binding

Variable Token Normalization

Variable and method names are special as they often convey the semantics of the program. For example, a JavaScript function with a name `setCookie` is highly likely to relate to the task "setting/storing cookie". However, for convenience and programming efficiency, programmers frequently use abbreviations or short forms for these names. For example, in the JavaScript statement `var currMon = mydate.getMonth()`, `currMon`, `mydate` and `getMonth` represent English words "current month", "my date" and "get month" respectively.

To a machine learner, the tokens `currMon` and `curMonth` are distinct and unrelated. To connect these forms together, I need to normalize these non-standard words (**NSW**) to resolve this feature mismatch problem (Sproat, Black, Chen, Kumar, Ostendorf, & Richards, 2001; Rowe & Laitinen, 1995).

Example	*Transition Patter*	*Split Result*
onMouseOver	single lowercase ⇔ single uppercase	on Mouse Over
IPAddress	consecutive uppercase ⇒ lowercase IP Address	IP Address
thisweek	not an English word, no transition and no less than 6 in length	apply $NSWSplit()$

Table 4.1: Variable token normailzation-splitting phase

I normalize such non-standard words by identifying likely splitting points and then expanding them to full word forms. In other words, this normalization process consists of two phases, namely, **NSW Splitting** and **NSW Expanding**. Splitting is achieved by identifying case changes and punctuation use. Words without case change which can not be found in dic-

Original	*Expanded*	*Original*	*Expanded*	*Original*	*Expanded*
msg	message	tab	table	cur/curr	current
pos	position	fig	figure	bg	background
img	image	pwd	password	str	string

Table 4.2: Variable token normalization - expanding phase

```
Algorithm NSWSplit : Splitting of a multi-token Non-Standard Word(NSW).
Input: the NSW token, Output: the resultant split words subtokens[]
Require: token has no case transition, is not English word and token.length() >= 6
 1: n <- token.length()            //n is the length of token
 2: prob <- new double[n][n]       //prob[i][j] is the prob stat for subtoken[i][j]
 3: subtokens <- new String[n][n]  //subtoken[i][j]=token.substring(i,j+1)
 4: for i = 0 to n do
 5:  for j = i to n do
        //read statistics from "Web Term Document Frequency Form"
 6:     prob <- getProb(subtoken[i][j])    //the prob. it appear on the page
 7:     prob[i][j] <- ( - prob * log(prob) )
 8:  end for
 9: end for
10: for parts = 1 to n do
11:  for i = 0 to n - parts + 1 do
12:     j = i + parts - 1
13:     for m = 1 to j - 1 do
14:          currProb <- prob[i][m] + prob[m + 1][j]
15:          if currProb > prob[i][j] then
16:               subtokens[i][j] <- combine(subtokens[i][m], subtokens[m + 1][j])
17:               prob[i][j] <- currProb
18:          end if
19:     end for
20:  end for
21: end for
22: return subtokens[0][n - 1]
```

Figure 4.3: Algorithm NSWSplit(splitting of non-standard word for variable token normalization)

tionary and is 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). The algorithm(NSWSplit) for splitting of such kind of NSW is illustrated in figure 4.3. My algorithm is based on the statistics collected from the "Web Term Document Frequency Form"[1]. I find the optimal split for the NSW based on the frequency statistics of a certain word appearing on Web pages provided by this site. Table 4.1 gives examples illustrating my overall NSW splitting scheme.

After NSW splitting, an expansion phase is carried out, in which commonly abbreviated shortenings are mapped to the word equivalents (*e.g.*, "`curr`" and "`cur`" → "`current`") using

[1]Web Term Document Frequency Form: http://elib.cs.berkeley.edu/docfreq/

a small (28 entries) hand-compiled dictionary. Table 4.2 gives examples for NSW expanding.

String Token Normalization

As JavaScript draws from Java and Web constructs, many string tokens conveys crucial information. For example, a string token like “`#FFFFFF`” and “`#F0F0F0`” indicating a hex value, always appear in the context of “changing background color” task. To a machine learner, however, they are considered unrelated. In order to tackle this feature mismatch problem, I further distinguish string tokens as URLs, file extensions(images and multimedia), HTML tags and color values. Table 4.3 gives examples on String Token Normalization.

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

Table 4.3: String Token Normalization

Token Stemming

Finally, all variable tokens and string tokens are stemmed using the Porter's Stemming Algorithm(Porter, 1997).

4.1.2 Syntax Analysis

Syntax plays a crucial role in program analysis and comprehension. It is a key feature that makes program source code different from free text. I believe syntax as an important feature could possibly assist my categorization task, and I am also interested in finding out whether syntax feature alone can perform categorization task well.

I parse every syntactically-correct JavaScript source code into a tree structure, that is, abstract syntax tree(AST). An AST is a data structure representing a program source code which has been parsed. AST is often used as a compiler or interpreter's internal representation of a program while it is being optimised and from which code generation is performed. The

range of all possible such structures is described by the abstract syntax. Figure 4.4 gives an example of JavaScript function and its corresponding AST.

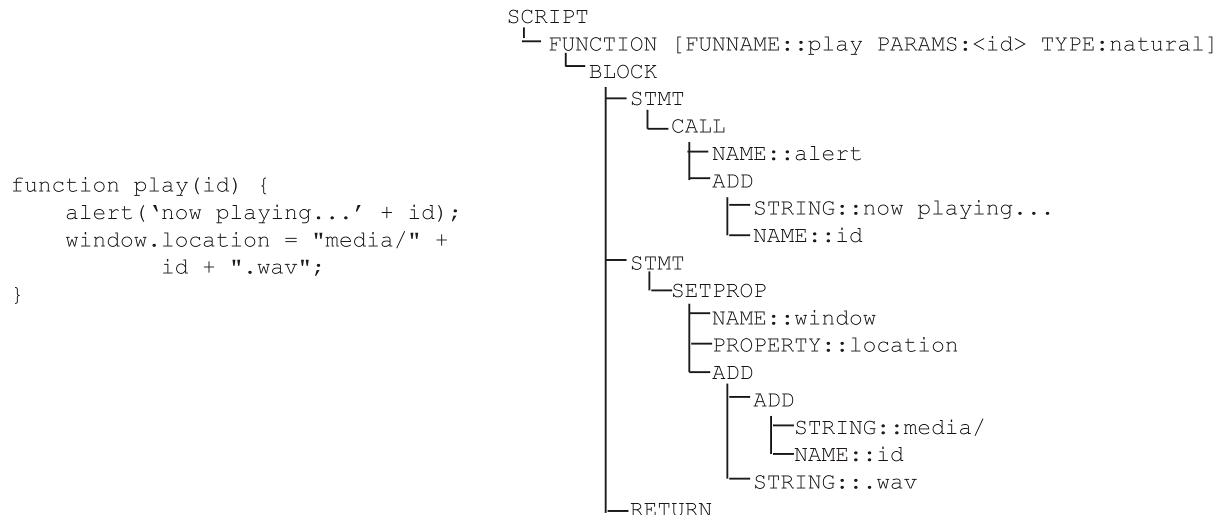

Figure 4.4: Example of JavaScript parse tree

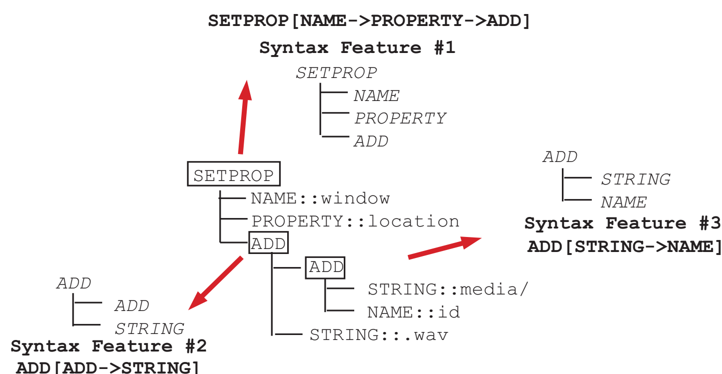

Figure 4.5: Example of syntax features

Syntax information as features are extracted from ASTs of the corresponding JavaScript code fragment and stored as a nested data structure. Assume we have an AST with N nodes in total. I make each of these node as root and keep all the direct and indirect children of that node, obtaining N subtrees rooted by N unique nodes. Next, for each of the N subtrees, based on the tree hierarchy, each nodes in the subtree will be stamped a level, root will be in *level 0*, root's direct children in *level 1*... Then I traverse the parse tree with pre-order scheme,

```
Algorithm SynFeatureExtract Retrieval of Syntax Features from an AST
Input: parse tree T, extraction depth D. Ouput: List of SyntaxFeature features
 1: features <- new SyntaxFeatureList()
 2: for each node in T
 3:  feature <- SynFeatureGenerate(node, D)
 4:  features.add(feature)
 5: end for
 6: return features

Algorithm SynFeatureGenerate To generate a Syntax Feature from an AST Node and its children
Input: AST Node node, extraction depth D, Output: Syntax Feature synFeature
 1: if D == 0 then
 2:  return new SyntaxFeature(node)
 3: end if
 4: childrenFeatures <- new SyntaxFeatureList() //initialize an empty syntax feature list
 5: for Node cursor <- node.firstChild to node.lastChild do
 6:  cursorFeature <- SynFeatureGenerate(cursor, D - 1)
 7:  childrenFeatures.append(cursorFeature)
 8: end for
 9: synFeature <- new SyntaxFeature(node, childrenFeatures)
10: return synFeature
_____________________________________________________________________
               where:
               SyntaxFeatureList := SyntaxFeature
                                 := SyntaxFeature + SyntaxFeatureList
               SyntaxFeature := node
                             := node + SyntaxFeatureList
```

Figure 4.6: Algorithm for syntax feature retrieval

i.e. $node_k \rightarrow node'_k s\ left\ subtree \rightarrow node'_k s\ right\ subtree$. In this manner, the visiting order constructs a *linear sequence* of nodes. This *linear sequence* will be the structure feature, or syntax feature that I can extract. It is important to note that during the pre-order traversing, I have a maximum depth constraint, *i.e.* I have a pre-determined maximum depth value D (I set it to 1-3 in my later experiments), such that nodes with a level deeper than D will be ignored (considered not visible) during traversal. The rationale for setting the maximum depth constraint is to make syntax features retrieved less specific and more general, so that more features can be matched over instances. Figure 4.6 gives an algorithm for syntactical feature retrieval. Figure 4.5 illustrates the algorithm by an example.

4.1.3 Code Metrics Analysis

There are varieties of code metrics. They are mainly designed or proposed for the purpose of software measurement and project management. In this section, I present my work on code metrics specifically in two types, namely, complexity metrics and reuse metrics respectively in

next two sub-sections.

Complexity 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 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 for the WT10G corpus). As such, I start with simple, classic metrics to assess their impact on categorization.

1. Counting metrics

Logical lines of code (LLOC) is one of the simplest metric in measuring software, it counts the number of logical lines in the code. In my system's implementation, I obtain this metric by counting the number of lines in a pretty-printed source code.

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.

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

2. Structure metrics

Total cyclomatic complexity (TCC) is a variance of the widely used control flow complexity metric, cyclomatic complexity(CC). 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. TCC is defined as $TCC = Sum(CC) - Count(CC) + 1$, 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 of the code lines within the procedures of your project. It is defined as $DECDENS = CC/LLOC$, where CC is the cyclomatic complexity, $LLOC$ refers to the logical lines of code.

Depth of conditional nesting (DCOND) is a complexity metric related to cyclomatic complexity. 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) is an information flow metric. 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. This metric is traditionally defined for class and interfaces, constructors and methods. In my system's implementation, I 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 classical software metrics employed in software measurement tasks and are generally applicable for different general-purpose programming languages. In addition to the above metrics, I believe some non-language specific metrics is worth investigation in my task. Based on my observation of JavaScript instances in my corpus, I further developed some additional metrics below. These metrics count language-specific information that I found were prevalent in the corpus and may be indicative of certain program functionality.

4. JavaScript language-specific metrics

Similar/Repeating Statements (SS) counts the number of statements with highly similar/repeating structure. The similarity measure of structure is examined by an abstract syntax tree. This is helpful in detecting scripts performing 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 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 function is a unique feature of the JavaScript language. One can define and instantiate an object via a function. Figure 4.7 gives an example of function object. This is helpful in detecting scripts involving complicated tasks like game or e-commerce.

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

Figure 4.7: Example of object function

Reuse Metrics(Edit Distance Measures)

Aside from complexity metrics, I 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 minimal edit distance between two inputs using strings, tokens or trees as elements for computation.

I can employ a standard string edit distance algorithm to calculate similarity between two script instances. I use the conclusion obtained from 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)(Baxter, Yahin, Moura, Sant'Anna, & Bier,

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. As AST is a better data representation of a JavaScript source code, one can therefore employ a tree-to-tree edit distance to measure the difference of two JavaScript codes. The tree-to-tree matching problem has a fairly long history(Tai, 1979; Lu, 1979; Wang, Zhang, Jeong, & Shasha, 1994; Selkow, 1977) and currently there exists some variances of approximate tree-to-tree matching algorithms. Most of them employ the dynamic programming scheme. Yang presents a tree-to-tree matching algorithm(Yang, 1991) to identify "syntactic differences between two programs". His method makes use of a recursive dynamic programming algorithm, with various relaxation schemes during matching phase. For example, he postulates that the symbols at the roots of `for` loops and of `while` loops are considered "comparable" and can be matched. He shows that his method is effective in identifying "syntactic differences" between programs and performs better than the simple string matching algorithm in the task.

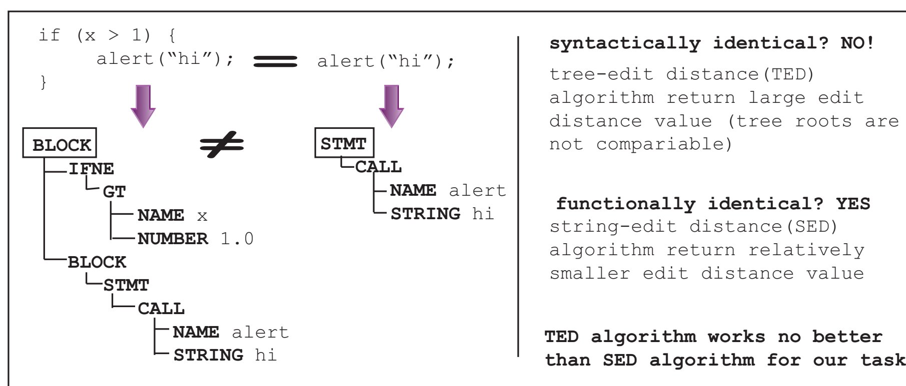

Figure 4.8: Tree edit distance(TED) algorithm does not work better than string edit distance(SED) algorithm for my task

However, it is important to note that "syntactically difference" does not infer "functionally difference", and vice versa. Actually lots of functionally identical codes are different in syntactical form. Take the two source codes in figure 4.8 for example, they both have a common statement `alert(''hi'')`, but it appears under a `if`-conditional block for the code to the left. This is considered a "syntactic" difference. Because of this difference, their respective syntactic

parse tree presented below are highly different in structure. In a tree edit distance(TED) algorithm, since the two root nodes are different, the two trees will be edited in a deletion-insertion manner, with a fairly large edit cost. This large cost is reasonable for syntactic-similarity identification as they are indeed syntactically different. However, on the other hand, a simple string edit distance(SED) algorithm, as we can observe, yields a relatively small edit distance in this case. From this example we can observe that the TED algorithm, although good in finding syntactical similarity over programs, may not be a better candidate than SED algorithms for finding functionality similarity.

Although the standard TED algorithm is not believed to be a better candidate than SED in assisting my task, I believe structure(syntax) information can augment simple SED algorithm's performance. A previous work shows a lexical-token based matching algorithm(Kamiya, Kusumoto, & Inoue, 2002) is effective in source clone detection task. In that work, It employs a complicated transformation scheme before the token-token matching process, and the algorithm is designed specifically for languages like C and Java. As this is not a dedicated work on clone detection tasks, I introduce a simple lexical tokens-based edit distance(LED) algorithm in this work, where lexical tokens are introduced earlier in section 4.1.1 on lexical analysis. Similar to SED algorithm, in which each character is treated as unit for matching, I treat lexical tokens as atomic unit in my matching algorithm. Two codes are represented in the lexical token form, and comparison is done over tokens. A deletion or insertion is assigned edit cost 1. As each token has two properties, *i.e.* its type and its value, I introduce a scheme for their edit cost during matching as described in table 4.4.

Editing from token t_1 to token t_2	*Edit cost*
1. TYPE(t_1) $\neq$ TYPE(t_2)	2.0
2. TYPE(t_1)==TYPE(t_2) && VALUE(t_1)==VALUE(t_2)	0
3. TYPE(t_1)==TYPE(t_2) && VALUE(t_1)$\neq$ VALUE(t_2)	
3-1. TYPE(t_1)==TYPE(t_2)==Variable Token || Symbol Token || Regexp Token || Number Token	0.5
3-2. TYPE(t_1)==TYPE(t_2)==Keyword Token	1.0
3-3. TYPE(t_1)==TYPE(t_2)==String Token	SED between VALUE(t_1) and VALUE(t_1)

Table 4.4: Edit cost scheme for lexical-token based edit distance(LED) algorithm

In order to assess each algorithm's effectiveness in assisting my task, I have implemented

all three different edit distance algorithms. As for the TED algorithm, I used a variance of the standard algorithm. Similar to Yang's algorithm, I introduced some relaxation scheme(e.g. `for` loop and `while` loops are considered identical), the algorithm is given in appendix D. I also introduce a checker "editable" checking whether two tree nodes are editable from one another. Please refer to appendix D for the detailed algorithm. A final **dis-similarity** measure is defined as follows for all the three algorithms:

$$\mathbf{DISIM(c_1, c_2) = MED(c_1, c_2)/max(n_1, n_2)},$$

where c_1, c_2 refers to two JavaScript codes, in the form of pretty-printed source code, AST tree, and token stream respectively for the three algorithms, and n_1 refers to the size(length) of the source code, total number of tree nodes, and total number of tokens for c_1 respectively, and similarly for n_2. $MED(c_1, c_2)$ is the corresponding minimum edit distance measure between code c_1 and c_2.

For each instance in my testing-set, I compute the dis-similarity measure with all instances in training-set. The training instance with a sm6allest dis-similarity value will be picked. Its category will be retrieved and stored as a separate "reuse metric" feature and passed to machine learner.

4.2 Context Sensitive Analysis

So far I have considered JavaScript *unit*s as independent of their enclosing web pages. In practice, JavaScript *unit* has an intimate relation with its enclosing page and is often meaningful only in context. Therefore I believe many important hidden features from the *unit*'s enclosing HTML page are worth investigation.

4.2.1 JavaScript Unit Property Analysis

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, I further extract the responsible DOM object and event handler. I 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*. These information are collected by my system and stored as a separate set of features.

`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 4.5: Units that reference their HTML context

4.2.2 Object Communication Analysis

JavaScript often interact with objects on HTML page in many ways. These objects are represented by the document object model (DOM[1]) and the browser object model (BOM)(Goodman, 2001)[2].

In fact, a *unit* which does not interact with any 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. Table 4.5 illustrates two examples where the script references DOM objects. In this section, I discuss how I recover such important object-communication features which get lost during previous analysis. I present my approach in two subsections, namely, static analysis and dynamic analysis.

Static Object Communication Analysis

I classify JavaScript *units*’ references to DOM objects into several categories: *gets*, *sets*, *calls*, and *creates*. These are illustrated in the JavaScript *unit* in figure 4.9: on line 0 and 2 `DoIt()` gets a reference to a form object named “`frm`”, on line 6 `frm.txt` refers to a `input` object `txt`

[1]http://www.w3.org/DOM/

[2]Although the two models are distinct from each other, I collectively refer to the two models as DOM for readability

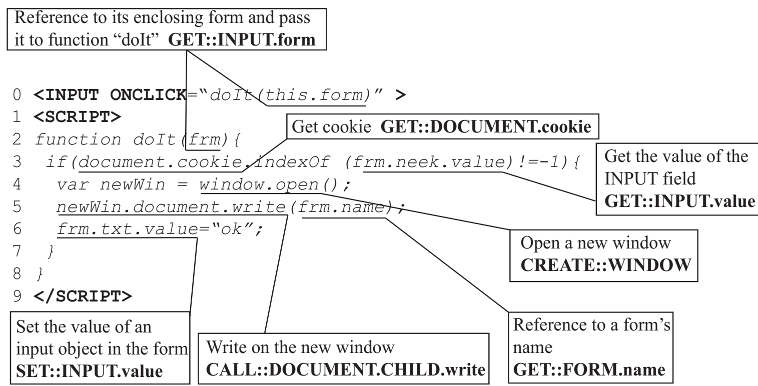

Figure 4.9: Examples of static object communication feature retrieved

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

Table 4.6: Partial table describing DOM object's references traced in object communication analysis

enclosed in the form, and the `input` object is assigned to the value "`ok`" by that statement, on line 5 the object `document` calls its `write` method, and on line 4 a `window` object is created. The count of each of these DOM object references is added as an integer feature for categorization.

DOM object settings and values may flow from one procedure to another. I recover the data type of objects by tracing the flow as variables are instantiated and assigned. Note that this process is done statically. I never know which execution path will be selected during real execution in a browser, therefore every possible execution path is traced. This is done with the assistance of the abstract syntax tree described in section 4.1.3. All the JavaScript *unit*'s interaction with DOM objects are then traced statically and recorded (*e.g.* GET::INPUT.value) as static object communication features for categorization. Figure 4.6 shows some example DOM objects together with their respective methods and attributes that could be traced by my system during static analysis.

Dynamic Object Communication Analysis

Static analysis is not able to recover certain information that occurs at run time. For example, figure 4.10 gives three JavaScript code fragments, performing three different tasks.

```
var toDay=new Date();
var str=toDay.getMinutes()
     * toDay.getSeconds();
document.write(str);
```

(A: random number)

```
var toDay=new Date();
var str=toDay.getMinutes()
     + ":" + toDay.getSeconds();
document.write(str);
```

(B: system time)

```
var toDay=new Date();
var str=toDay.getMinutes()
     * toDay.getSeconds();
```

(C: initialization)

Figure 4.10: Similar codes performing different tasks

Although all the three code fragments reference to system time and have similar variable names, function calls and syntax structure, they perform quite different tasks: code A generates and prints a random number on the page, while code B displays current system time, and code C is simply an initialization task. Simple lexical token-based analysis will yield highly identical set of tokens for them. Static object communication analysis, on the other hand, can not perform better than lexical analysis in dealing with this case, as it fails to recover information conveyed by the variables "`str`".

Such information can only get revealed during an execution. Dynamic analysis (*i.e.*, execution of the program) 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.

I illustrate how dynamic analysis can yield additional features for categorization in figure 4.11. This sample JavaScript *unit*, taken from the WT10G corpus, creates a dynamic text banner that scrolls in window's status bar. The function "`banner`" employs the function `window.setTimeout()` to make itself get executed over and over again after every 100 milliseconds, which makes the banner text in the window change over time. Without dynamic analysis, I cannot recover what value `msg.substring(0,index)` refers to. More importantly, dynamic analysis allows us to extract the value of the `‘‘banner(’’+index+‘‘)’’` variable. In this example, dynamic analysis also recovers the fact that the variable's value is changing, hence a

new feature, CHANGES::WINDOW.status, is added to the feature set.

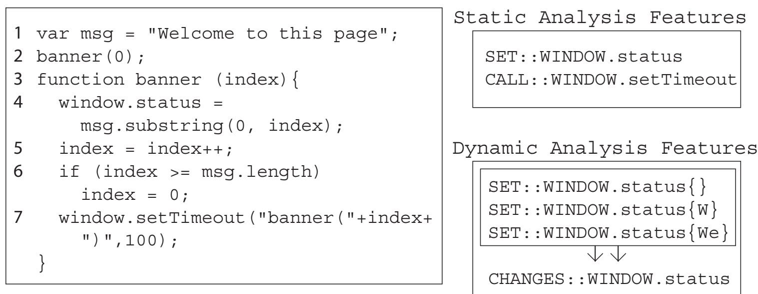

Figure 4.11: Sample JavaScript unit (l), along with features extracted by static and dynamic analysis (r).

4.3 A Tiny Example

Considering the following code:

```
myDate = new Date();
document.write(today.getMinutes()+``:''+today.getSeconds());
```

Analysis technique	Features retrieved and their values
Lexical analysis	today[VAR] 1,new[KEY] 1,Date[VAR] 1,([SYM] 1,)[SYM] 1,...
Syntax analysis	STMT[SETNAME] 1,SETNAME[BINDNAME→NEW] 1...
Code metrics analysis	*a set of numeric values*
JavaScript unit property analysis	*boolean features, depends on its enclosing HTML page*
Static object communication analysis	CREATE::DATE 1,CALL::DOCUMENT.write 1,CALL::DATE.getMinutes 1,...
Dynamic object communication analysis	CREATE::DATE 1,CALL::DOCUMENT.write 1,...,WRITE::TIME 1

Table 4.7: How the features look like

I illustrate in table 4.7 for each analysis techniques how features will get retrieved. Information obtained from lexical analysis, syntax analysis, static object communication analysis and dynamic communication analysis will be stored in a token form(*e.g.* CREATE::DATE is as an atomic token form). The count of such tokens is then added to the feature set as an integer feature, while features retrieved in code metrics analysis will also be a set of numeric features. These features will then be passed to a machine learner for classification.

Chapter 5

Evaluation

I 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 JavaScripts. 10,546 JavaScript *functional unit*s were extracted from them. String identical duplicates were then removed. This resulted in a corpus of 4,348 *unit*s, which are unique in textual form. Next, I annotated all the 4,348 *unit*s, dividing them into functional groups. Structurally-identical units were then removed within each group. This resulted in a final corpus of 1,637 *unit*s, which are unique in textual form and structure. The high ratio of the number of script instances to unique scripts validates my claim that many scripts are simply clones. Table 5.1 gives the distribution of webpages for the corpus.

Number of unique JavaScript units	*Number of unique servers*	*Number of units sub-total*
1	418	418
2-5	215	1023
6-10	30	1243
11-20	11	1380
21-30	8	1578
59	1	1637

Table 5.1: Unique JavaScript unit v.s. unique server distribution, in WT10G corpus

I 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 my feature sets contain over 10,000 features. More specifically, I used John C. Platt's SMO algorithm provided by the Weka machine learning toolkit (Witten & Frank, 2000), which is a sequential minimal optimization algorithm for training a support vector classifier using scaled polynomial kernels (Platt, 1999). I 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.

5.1 Evaluation on Each Component

5.1.1 Baseline

I build a baseline using a simple text categorization approach. I treat each JavaScript *unit* as a single document, with its corresponding JavaScript source code as document content. I then treat all the documents as text documents and tokenize them into bag of tokens, where white space and punctuations are delimiters(note that although punctuations are used as delimiters, they are not removed but preserved as tokens as well). The resultant bag of tokens form a feature vector representing the document, *i.e.* the JavaScript *unit*. This approach gives an 10-fold cross validation accuracy score 87.47%. This high baseline score indicates that the categorization task is relatively easy to perform well using simple methods, while on the other hand, to improve its performance could also be a challenging task to do.

5.1.2 Evaluation on Context Free Analysis

Evaluation on Lexical Analysis

Lexicial Features Used	*Accuracy(SVM)*	*ER*
B_t. Text categorization baseline	87.47%	–
L_c. Compiler-based Tokenization and Type-binding	88.57%	8%
L_n. L_c+variable/string token normalization	89.61%	17%
L. $max(L_c,L_n)$All lexical analysis	**89.61%**	17%

Table 5.2: Evaluation on lexical analysis, Error reduction (ER) is measured against the text categorization baseline.

I hypothesized that token features and token(variable and string) normalization would enhance performance. The results in table 5.2 show that simple typing of tokens as keywords, strings, symbols and so on is effective at removing 8% of the categorization errors. When normalization over variable tokens and string tokens is introduced, the performance is further improved, removing 17% of errors[1]. This validates my earlier hypothesis that program language features do positively impact program categorization.

Evaluation on Syntax Analysis

Syntax Features Used	*Accuracy(SVM)*
B_m. Most frequent class baseline	16.12%
B_t. Text categorization baseline	87.47%
S_1. D=1	65.36%
S_2. D=2	73.85%
S_3. D=3	74.34%
S. S_1+S_2+S_3 All Syntax Features	**75.02%**

Table 5.3: Evaluation on syntax analysis

I would like to assess how well structure(syntax) features alone can perform in a program categorization task. I experimented on three sets of syntax features based on the algorithm described in figure 4.6 by selecting features with different structure level. Results are shown in table 5.3. It is not surprising that using structure features alone can not beat the text categorization baseline. This is because lots of crucial information conveyed by tokens like variable names, strings are covered by a structure syntax feature. This result does provide evidence that syntax information play an important role in program categorization task and it alone can perform a certain level of categorization task.

Evaluation on Code Metrics Analysis

I break down my composite metric feature set into its components to assess their predictive strength. As complexity metrics are all numerical values, I assessed their effectiveness in catego-

[1]Note that Porter's stemming algorithm is applied in both L_c and L_n, as well as the baseline

Metric	*Accuracy(SVM)*	*Accuracy(J48 Tree)*
B_t. Text categorization baseline	87.47%	-
M_c. LLOC	16.25%	22.17%
M_n. NOA	16.13%	20.59%
M_p. NOP	16.31%	21.81%
M_t. TCC	16.19%	21.56%
M_e. DECDENS	17.34%	23.76%
M_d. DCOND	16.13%	19.98%
M_l. DLOOP	16.13%	16.74%
M_i. IFIN	16.25%	23.21%
M_{ST}. (M_c+M_n+...+M_i) All standard complexity metrics	20.46%	44.53%
M_s. SS	17.34%	17.39%
M_b. BO	23.40%	23.40%
M_o. OF	15.88%	15.64%
M_{JS}. (M_s+M_b+M_o) All JavaScript specific complexity metrics	23.34%	24.98%
M_{COMP}. M_{ST}+M_{JS} All complexity metrics	25.60%	47.95%
M_r. Pretty-printed String-based edit distance	73.85%	73.79%
M_p. AST-based edit distance(w/o "editable" constraint)	58.03%	57.79%
M_a. AST-based edit distance(w/ "editable")	72.69%	72.51%
M_t. Token-based edit distance	74.89%	74.77%
M_{REUSE}. max(M_r,M_p,M_a,M_t) All reuse metrics	74.89%	74.77%
M. M_{COMP}+M_{REUSE} All Metrics	**77.76%**	**77.09%**

Table 5.4: Evaluation on code metrics analysis

rization from two different machine learners. In addition to SVM classifier, a J48-tree classifiers is also employed here, as decision tree algorithms is believed to handle numeric features well. Experiment results are shown in table 5.4. A finding of my work is that applying published software metrics "as-is" may not boost categorization performance too much, rather these metrics need to be adapted to the classes and language at hand. My experiment 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 the simple text categorization baseline. We can also observe that token-based edit distance gives the best performance while AST-based edit distance works worse than simple string-based edit distance algorithm. This validate my claims in section 4.1.3. Code metrics alone is not sufficient to perform categorization task, but only when collectively used

with lexical analysis is performance increased(Please see section 5.2).

5.1.3 Evaluation on Context Sensitive Analysis

Type of Analysis	*Accuracy(SVM)*
B_t. Text categorization baseline	87.47%
P_u. JavaScript Unit Property Analysis	25.47%
P_s. Static Object Communication Analysis	79.78%
P_d. Dynamic Object Communication Analysis	71.22%
P_o. P_s+P_d All Object Communication Analysis	85.64%
P. P_u+P_o All Context Sensitive Analysis	**87.29**%

Table 5.5: Evaluation on context sensitive analysis

As for context sensitive analysis, 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). Dynamic analysis performs worse than static analysis because certain *unit*'s codes can only get executed conditionally, which makes dynamic analysis fail to retrieve relevant features. The combined feature set also does not beat the simple lexical approach, but serves to augment its performance.

5.2 Evaluation on All Components

So far I have assessed individual the effectiveness for each individual feature sets in categorization task. I have measured the performance difference using different sets of machine learning features. Next, I would like to investigate how these feature sets can be incorporated together so that the overall performance can be augmented.

Table 5.6 shows the overall evaluation on each component and combination of them. Here, I can see the majority class categorizer performs poorly, as this dataset consists of many classes without a dominating class. However, the simple text categorization baseline accurate on 87% of the test instances. When informed lexical tokenization is done and combined with features from software metrics, context sensitive analysis, I am able to improve categorization accuracy

Features used	*Accuracy(SVM)*	*ER*
B_m. Most frequent class baseline	16.12%	–
B_t. Text categorization baseline	87.47%	–
L. All lexical analysis	**89.61%**(**)	17%
S. All syntax analysis	74.34%	–
M. All code metrics analysis	77.76%	–
P. All context sensitive analysis	87.29%	–
L+S	89.00%	–
L+M	**90.04%**(*)	21%
L+P	**92.36%**(**)	39%
L+M+P	**93.95%**(**)	52%

Table 5.6: All components 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

to around 94%. While syntax features performs worst amongst all the components, it fails to augment the performance when used together with lexical features. I believe this is because instances from different categories share certain common syntax information(*e.g.* a `while` loop). If such feature is selected by the machine learner, a noise will will be added . In viewing of this, I exclude this set of features from my further experiments. Perhaps unsurprisingly, using only software metrics and program comprehension features fail to contribute good classifiers. However, when coupled with a strong lexical feature component, I show improvement.

At first glance, the performance improvement seems marginal, only contributing a few percentage points. I am encouraged as 1) my dataset is large (1.6K instances), 2) ten-fold cross-validation is used and 3) the classification is difficult, consisting of 32 classes. 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 I also calculate the error reduction that is achieved by my methods over the text categorization baseline. By this metric, more than half of the classification errors are corrected by the introduction of my techniques.

Chapter 6

Conclusion

6.1 Limitations

My results are promising, but I would like to call attention to some of the limitations of my work that need to be addressed:

Annotator Agreement It is always the case that different annotator will come up with a different categorization scheme for the same corpus. Also, unlike many other annotation tasks, annotator for JavaScript must have a good knowledge of JavaScript language to make sure annotation task can be well performed. In order to achieve a higher annotation consistency, my corpus is annotated carefully by myself. I annotated the a corpus consisting of 4,348 instances as described in chapter 5. The entire process takes up to 40 hours, including a manual annotation phase and a manual refinement phase(during annotation, I shall not simply assign a category by merely examining the JavaScript source code associated to a JavaScript *unit*, rather I need to examine the enclosing HTML page of the *unit*, so that an accurate category can be inferred and assigned. For example, the annotator needs to examine in what context a *unit* appears and how it interacts with the user and other HTML DOM objects. However, although the annotation is very careful, during the annotation process, he notes that some instances are problematic in assigning categories.

For example, as shown in figure 6.1, it is somehow hard to assign a precise category for the

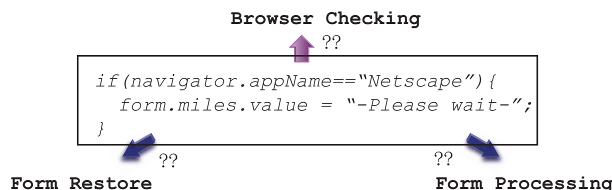

Figure 6.1: Problematic instance in annotation

given JavaScript instance as it could either be a "form restore", or a "form processing" task. Because it checks the browser's application name, one can even argue it should be annotated as "browser checking"! I feel this a source of some errors and plan to work on further annotation and finding inter-annotator agreement if possible. A reasonable upper bound of performance may be less than 100%, meaning that my performance gains may be more significant than discussed in my evaluation section.

Dynamic Analysis Incompleteness Many tasks are executed conditionally depending on varies conditions, *e.g.*, whether a flag is set to true before execution, whether the browser is detected to be compatible, and so on. In my dynamic analysis, I assume scripts are only executed under as Microsoft Internet Explorer 4.0, which causes certain features or data fail to get extracted. As browser checking is ubiquitous in JavaScripts, in future work, I may relax this constraint and follow all execution pathways that are conditional on the browser.

Choice of Classifier A state-of-the-art machine learner - SVM classifier is used throughout evaluations in this work. As one may have noticed from figure 5.4, when handling numeric features, a SVM classifier performs no better than a decision tree classifier. As I have various numeric features in my feature space, I believe the choice of classifier could also be a source of my classification error.

6.2 Contributions

I present an novel approach to the problem of program categorization. In specific I target JavaScript categorization, as its use is largely confined to a small set of purposes and is closely

tied to its enclosing web page. The major contribution of this work includes:

Created a functionality-based categorization on JavaScript A key contribution of my work is to create a functional categorization of JavaScript instances, based on a corpus study of over 18,000 web pages with scripts. Prior work on categorization JavaScript are developer-based and none is based on user-oriented functionality. In this work, I worked out a categorization scheme which is purely functionality based and is well defined.

Showed how program analysis can assist program categorization Automatic program categorization as a challenging task did not receive much attention before. Rather than treating the problem merely as a straightforward text categorization problem, I incorporate and adapted metrics and features that originate in program analysis. My corpus study confirms that many scripts are indeed copies or simple modifications. While the baseline does well, performance can be greatly improved by utilizing program analysis. A careful lexical analysis eliminates 17% of errors. Further works employing program analysis techniques result in a 52% overall categorization error reduction. While my results are preliminary, I believe they provide evidence that program categorization can benefit from adapting work from program analysis.

Developed a complete software application on JavaScript categorization I have developed a complete software application on JavaScript categorization. The system includes modules performing program analysis on JavaScript and feature extraction and automatic categorization. It is release in SoCForge(*http://ecmascript.socforge.net*) as an open source package under GNU license.

6.3 Future Work

I plan to extend this work to other scripting languages and decompiled plug-ins appearing on Web pages. I aim to characterize the information conveyed by these devices to assist end users to filter irrelevant material and to help summarize such information for users to make more informed Web browsing and searching choices.

References

Baxter, I. D., Yahin, A., Moura, L. M. D., Sant'Anna, M., & Bier, L. (1998). Clone detection using abstract syntax trees. *ICSM* (pp. 368–377), 1998.

Blazy, S., & Facon, P. (1998). Partial evaluation for program comprehension. *ACM Computing Surveys*, *30*(3), September, 1998.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W., & Harshman, R. A. (1990). Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, *41*(6), 1990, 391–407.

Goodman, D. (2001). *Javascript bible gold edition*. 909 Third Avenue, New York, NY 10022: Hungry Minds, Inc.

Gschwind, T., Oberleitner, J., & Pinzger, M. (2003). Using run-time data for program comprehension. *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension* (p. 245), Washington, DC, USA, 2003: IEEE Computer Society.

I.Maletic, J., & Marcus, A. (2001). Supporting program comprehension using semantic and structural information. *Proceedings of the 23rd International Conference on Software Engineering* (pp. 103–112), Toronto, Ontario, Canada, 2001.

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.

Koller, D., & Sahami, M. (1997). Hierarchically classifying documents using very few words. In D. H. Fisher (Ed.), *Proceedings of ICML-97, 14th International Conference on Machine Learning* (pp. 170–178), Nashville, US, 1997: Morgan Kaufmann Publishers, San Francisco, US.

Kontogiannis, K. (1997). Evaluation experiments on the detection of programming patterns using software metrics. *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)* (pp. 44–54), Washington, DC, USA, 1997: IEEE Computer Society.

Kushmerick, N. (1999). Learning to remove internet advertisements. *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents* (pp. 175–181), 1999: ACM Press.

Lu, S.-Y. (1979). A tree-to-tree distance and its application to cluster analysis. *1*(2), 1979.

Maarek, Y. S., Berry, D. M., & Kaiser, G. E. (1991). An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, *17*(8), August, 1991, 800–813.

Maletic, J. I., & Marcus, A. (2000). Using latent semantic analysis to identify similarities in source code to support program understanding. *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)* (p. 46), 2000.

Mathias, K. S., II, J. H. C., Hendrix, T. D., & Barowski, L. A. (1999). The role of software measures and metrics in studies of program comprehension. *ACM Southeast Regional Conference*, 1999.

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of program analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Panas, T. (2003). *Towards a generic architecture for reverse egnineering*. PhD thesis.

Platt, J. C. (1999). Fast training of support vector machines using sequential minimal optimization. , 1999, 185–208.

Porter, M. F. (1997). An algorithm for suffix stripping. , 1997, 313–316.

Rowe, N., & Laitinen, K. (1995). Semiautomatic disabbreviation of technical text. *Information Processing and Management*, *31*(6), 1995, 851–857.

Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, *24*(5), 1988, 513–523.

Selkow, S. M. (1977). The tree-to-tree editing problem. *Inf. Process. Lett.*, *6*(6), 1977, 184–186.

Sproat, R., Black, A., Chen, S., Kumar, S., Ostendorf, M., & Richards, C. (2001). Normalization of non-standard words.

Tai, K.-C. (1979). The tree-to-tree correction problem. *J. ACM*, *26*(3), 1979, 422–433.

von Mayrhauser, A., & Vans, A. M. (1994). Dynamic code cognition behaviors for large scale code. *Proceedings of the 3rd Workshop on Program Comprehension* (pp. 74–81), Washington, D.C., USA, November 14, 1994.

Wang, J. T. L., Zhang, K., Jeong, K., & Shasha, D. (1994). A system for approximate tree matching. *IEEE Transactions on Knowledge and Data Engineering*, *6*(4), 1994, 559–571.

Witten, I. H., & Frank, E. (2000). *Data mining: Practical machine learning tools with java implementations*. San Francisco: Morgan Kaufmann.

Wong, W.-C., & Fu, A. W.-C. (2000). Finding structures of web documents. *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, Dallas, USA, May 14, 2000.

Yang, W. (1991). Identifying syntactic differences between two programs. *Software - Practice and Experience*, *21*(7), 1991, 739–755.

Yang, Y., & Liu, X. (1999). A re-examination of text categorization methods. In M. A. Hearst, F. Gey, & R. Tong (Eds.), *Proceedings of SIGIR-99, 22nd ACM International Conference on Research and Development in Information Retrieval* (pp. 42–49), Berkeley, US, 1999: ACM Press, New York, US.

Appendix A

JavaScript Functional Unit

There are two types of JavaScript functional units as I have defined in chapter 3, namely interactive *unit* and non-interactive *unit*. I give a discussion on how they are formally defined and extracted by my system via an 81-line example HTML page illustrated in figure A.1. This example is derived from an pages in the WT10G corpus.

A.1 Interactive Unit

In the example page, there are four fragments of JavaScript embedded by the HTML tag "`<SCRIPT>`". On line 35, the event handler "*ONLOAD*" of the "*BODY*" object is associated with the JavaScript code "`scrollit_r21(80)`". When invoked, it triggers the JavaScript function "`scrollit_r21`" defined on line 04 - line 32. This information is collected statically by my system and a single JavaScript *functional unit* as an abstract representation is extracted by my system at its run-time.

Similarly, on line 57, the event handler "*ONCLICK*" is associated with the code "`calculate_total(this.form)`", which in turn triggers the function "`calculate_total`" defined online 49 - line 52. This function calls another function "`verify`" in its function body(line 50). Further more, function "`verify`" writes to the global variables "`miles`" and "`mpgcity`" on line 41 and line 43 respectively. All the above information is collected by my system and are stored as a single interactive *functional unit*.

A.2 Non-interactive Unit

Different from these interactive *unit*s, a non-interactive *unit* is those *unit*s which is not associated to any event handler. Line 78 shows an example of such type of *unit*. That line of code will get executed immediately when the web page is loaded. Unlike interactive *unit*s, this type of *unit* does not require any client-side interaction with the page and is therefore termed as non-interactive *unit*.

```
01 <HTML>
02 <HEAD><TITLE>Eric's Homepage</TITLE>
03 <SCRIPT>
04 function scrollit_r2l(seed){
05   var msg = "                 >>>>Welcome To Eric's Tourism Information Page <<<<<<";
06   var out = " "; var c=1;
07   if(seed>100){
08     seed--;  var cmd = "scrollit_r2l("+seed+")";
09     timerTwo = window.setTimeout(cmd, 80);
10   }
11   else {
12      if(seed <= 100 && seed > 0){
13        for(c = 0; c< seed; c++)
14         out+=" ";
15        out+=msg; seed--; var cmd="scrollit_r2l("+seed+")";
16        timerTwo = window.setTimeout(cmd,80);
17      }
18      else {
19        if(seed<=0){
20          if(-seed<msg.length){
21            out+=msg.substring(-seed, msg.length);
22            seed--; var cmd = "scrollit_r2l(" + seed + ")";
23            window.status = out;
24            timerTwo = window.setTimeout(cmd, 80);
25          } else {
26            window.status = " ";
27            timerTwo = window.setTimeout("scrollit_r2l(80)", 75);
28          }
29        }
30     }
31   }
32 }
33 </SCRIPT>
34 </HEAD>
35 <BODY ONLOAD="scrollit_r2l(80)">
36 <IMG SRC="images/logoon.gif" NAME="logon" ONMOUSEOVER="rollOverIt(this.name)">
37 <SCRIPT>
38 var miles,mpgcity;
39 function verify(obj){
40  if(obj.name =="Miles")
41    miles = obj.value;
42  else
43    mpgcity = obj.value;
44  if(mpgcity<0)
45   alert("Invalid mpgcity!");
46  if(miles<100)
47   alert("Invalid miles!");
48 }
49 function calculate_total(form){
50   verify(form.Miles);
51   form.Result.value = (1.36E15-form.Miles.value)*3+Math.floor(form.Mpgcity.value);
52 }
53 </SCRIPT>
54 <FORM NAME="isn">
55 <INPUT TYPE="TEXT" NAME="Miles" VALUE=1 ONCHANGE="verify(this)">
56 <INPUT TYPE="TEXT" NAME="Mpgcity" VALUE=2 ONCHANGE="verify(this)">
57 <INPUT TYPE="BUTTON" NAME="compute" VALUE="Compute" ONCLICK="calculate_total(this.form)">
58 <SELECT NAME="cities">
59  <OPTION VALUE="/newyork.html"> New York</OPTION>
60  <OPTION VALUE="/lonton.html">  Lonton    </OPTION>
61 </SELECT>
62 <INPUT TYPE="BUTTON" VALUE="Go" ONCLICK="loadpage(this.form)">
63 </FORM>
64 <SCRIPT LANGUAGE="JAVASCRIPT">
65 var logooff = new Image;
66 var projoff = new Image;
67 logooff.src = "images/logooff.gif";
68 projoff.src = "images/projoff.gif";
69 function loadpage(form){
70   window.location=form.cities[form.cities.selectedIndex].value;
71 }
72 function rollOverIt(img){
73   document.images[img].src= "images/"+img+"off.gif";
74 }
75 </SCRIPT>
76 <IMG SRC="images/projon.gif" NAME="proj" ONMOUSEOVER="rollOverIt(this.name)">
77 <SCRIPT LANGUAGE="JAVASCRIPT">
78 document.writeln("<b>Page Last updated: "+document.lastModified+"</b>");
79 </SCRIPT>
80 </BODY>
81 </HTML>
```


Figure A.1: JavaScript and its enclosing HTML page

Appendix B

Survey on JavaScript Usefulness

While my purpose for doing JavaScript categorization is to find functionality associated to JavaScript *unit*s and filter irrelevant or annoying JavaScript features from useful ones. Therefore it is important to get feedback from users about which of the above discussed JavaScript functionality is useful for a user while which ones are not.

For this purpose, I have conducted a “JavaScript Usefulness” survey over a group of 25 people[1]. I give short descriptions for each of my summarized categories and ask users for its “usefulness”, based on their own personal opinions. I provide 11 options for them to choose from, from “-5” to “+5”, where “-5” represent “least useful”, or “most annoying” functionality, “+5” represents the “most useful” functionality, and “0” in the middle represents a “neutral” functionality. Based on the survey result, I conclude which functionality are generally considered “useful”(YES) while which are “annoying”(NO) amongst users.The result of the survey is presented in the right half of table 3.1.

It is noted that the survey does not cover all the categories. Some categories like “Initialization”, “Pre-load Image” are not covered by the survey because of the following reasons: 1) they are hidden functionality which are not visible to a user and therefore it does not make sense to ask for a usefulness; 2) they have close relationship to other functionality, and some other functionality may not be able to be achieved without them.

We can observe from the result that pop-up and banners are generally not favored by people, while tasks like form processing, load page are considered useful for most people. From this survey, we can also observe that certain functionality achieved by JavaScript like “timer”, “cookie” are considered “inconclusive” in their usefulness, meaning users does not care much about the presents or absence of these types of functionality on a web page, i.e. they are neither “useful” and must be kept intact, nor “annoying” and must get filtered out.

Figure B.1 gives a screen shot of the survey.

[1] They are members from the WING group and evaluation subgroup. WING homepage: http://wing.comp.nus.edu.sg

Note:
We are collecting data from different individuals.
If you have finished the rating before, please do **NOT** enter this system again unless you are invited to do so. Thanks!

Javascript Helpfulness Rating

JavaScript is a client-side programming language meant for use on the Web. When you download a page with JavaScript in it, that script is run on your computer inside your Web browser.

As a client-side scripting language that runs in your Web browser, most of what JavaScript can do revolves around monitoring or altering the content of Web pages and interacting with the user through the Web browser interface.

Some Javascripts employed by certain web pages do help promote the HTML page and they should not be filtered. However, as you may have experienced, many Javascripts actually do things which annony us.

We are investigating what kind of functionalities achieved by Javascript is more helpful to the end-users and which are not.
You are invited to carefully rate several types of functionalities achived by Javascript.
It will take you approximately 10 minutes or so to finish all.

Please read the description patiently first, thanks!
Pop-up

Pop-up can be either a new browser window or an alert box that is opened in relation to an event within another window. It can either be opened automatically when the main browser window is launched, or be triggered by certain events, for example, clicking of a button.

1.Please rate "Pop-up"
annoying neutral
-5 -4 -3 -2 -1 0 +1 +2 +
Please provide comments about your rating for this functionality

2.Please rate "Client-side Time
annoying neutral
-5 -4 -3 -2 -1 0 +1 +2 +
Please provide comments about your rating for this

Figure B.1 Screen shot for the JavaScript Helpfulness survey (http://wing.comp.nus.edu.sg/~luwei)

Appendix C

Token Table

Comment Token All the comments or documentation appearing in the JavaScript source code. For example, in "var x=1 //assignment", here "assignment" is code comment and therefore stored as *Comment Token*.

String Token All the Strings appearing in the JavaScript source code. For example, in "document.write('<BR>')" and "setTimeOut(cmd+'(5)')", here "'<BR>'" and '(5)' are stored as *String Token*s.

Symbol Token All the punctuation symbols in the JavaScript source code. For example, in "var x += new Array(6);", "+=", "(", ")" and ";" are stored as *Symbol Token*.

Number Token All the numbers in the JavaScript source code. Note that non-decimal numbers and exponential numbers are converted to decimals and stored. For example, in "var x = 1.6E2" and "var y = 0x35", decimal numbers "160" and "53" are stored respectively, as *Number Token*s.

Regular Expression Token All the regular expressions in the JavaScript source code. For example, in "var re = /mac/i;", the regular expression token "/mac/i" will be stored as a whole, as *Regular Expression Token*.

Keyword Token All the keywords and reserved words of JavaScript in source code. For example, in "if(x!=true) break;", "if", "true" and "break" will be stored as *Keyword Token*s.

Variable Token All the identifiers and variables names, including object name, function name, parameters and method names. For example, in "var alpha=beta+ cos(theta)", "alpha", "beta", "cos" and "theta" are *Variable Token*s.

Appendix D

The AST-based Edit Distance Algorithm Used

```
Algorithm getMES Get Minimum Edit Operation Sequence between two JavaScript AST:
Input: AAST subTreeA, AAST subTreeB, EditSequence seq, int level Output: EditSequence
 1: M <- subTreeA.degree
 2: N <- subTreeB.degree
 3: dist <= EditSequence[M + 1][N + 1]
 4: if editable(subTreeA,subTreeB) then   //check whether two trees can be edited directly
      //if editable, compute the effort required for the direct edit
 5:  op <- editEffort(subTreeA, subTreeB)
 6:  dist[0][0] <- seq.append(op)
 7: else
 8:  op1 <- new Deletion(subTreeA, level)
 9:  op2 <- new Insertion(subTreeB, level)
10:  dist[0][0] <- seq.append(op1), seq.append(op2)
11: end if
12: for k = 1 to M do
13:  childA <- subTreeA.getChild[k]
14:  op <- new Deletion(childA, level + 1)
15:  dist[k][0] <- dist[k - 1][0].append(op)
16: end for
17: for k = 1 to N do
18:  childB <- subTreeB.getChild[k]
19:  op <- new Insertion(childB, level + 1)
20:  dist[0][k] <- dist[0][k - 1].append(op)
21: end for
22: for i = 1 to M do
23:  for j = 1 to N do
24:       seq[1] <- getMES(subTreeA, subTreeB, dist[i - 1][j - 1]), level + 1)
25:       op <- new Insertion(subTreeB, level + 1)
26:       seq[2] <- dist[i][j - 1].append(op)
27:       op <- new Deletion(subTreeA, level + 1)
28:       seq[3] <- dist[i - 1][j].append(op)
          //pick the EditSequence with minimum cost
29:       k <- argmini(computeCost(seqi)) (where i=1,2,3)
30:       dist[i][j] <- seq[k]
31:  end for
32: end for
33: return dist[M][N]
```

Figure D.1: The tree-edit distance(TED) algorithm used in my experiment

Appendix E

A Paper on this Work to AAAI Conference

This work has been submitted to the AAAI National Conference[1] for publication. The author of this thesis is the first author of the paper. AAAI paper has a limitation of 6 pages in length, and therefore the structure of the work described in this thesis is slightly different from the paper.

[1] http://www.aaai.org/Conferences/National/2005/aaai05.html

Supervised Categorization of JavaScript using Program Analysis Features

Wei Lu Min-Yen Kan

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, 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 of 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 exhibits no ambiguity 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. Recent work in categorizing web pages (Wong & 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 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 (Mathias *et al.* 1999). These models show that developers use both top-down and bottom-up models in

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

their learning process (von Mayrhauser & Vans 1994). Top-down models imply that developers may use a model of program execution to understand a program. Formal analysis via mock code execution (Blazy & Facon 1998) 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 (Kontogiannis 1997), as JavaScript instances 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 (Maletic & Marcus 2000). 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.

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 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.com 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 interact 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.

Category	Description (# units in corpus)
Dynamic-Text Banner	Displays a banner which does not change with time (264)
Initialization	Initialize/modifies variables for later use (121)
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 (79)
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 that changes content 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 (33)
Browser Information	Check browser information (32)
Cookie	Store or retrieve data on server about client (26)
Trivial	Perform a simple one-liner task (26)
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 functionalities or too few instances (133)

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

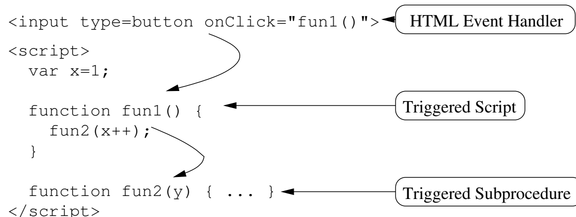

Figure 1: A JavaScript unit, the basic element used for our classification.

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.

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 mock 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.

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

Table 2: Examples of Name token normalization.

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 (Rowe & 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). A following expansion phase is carried out, in which commonly abbreviated shortenings are mapped to the word equivalents (*e.g.*, "curr" and "cur" $\rightarrow$ "current") using a small (28 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 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(G) 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. 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. 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. The similarity measure of structure is examined by a parse-tree(AST). This is helpful in detecting scripts performing similar tasks in a conditional manner. For example, displaying different greetings based on the time (e.g., morning, afternoon evening).

Built-in Object References (BOR) counts the number of built-in objects (*e.g.*, `window`, `date`) 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 minimal edit distance between two inputs using strings, tokens or trees as elements for computation.

We can employ a standard string edit distance 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.

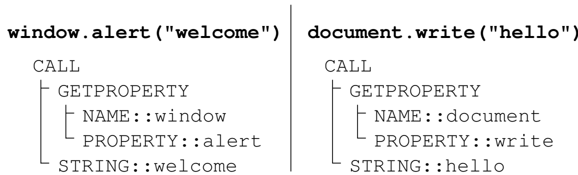

Figure 2: JavaScript and their parse trees.

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 the the unit and stores data types of its variables. Figure 2 shows an example in which the "window" and "document" occur in the same syntactic position in the JavaScript unit, but actually refer to different objects. In this case, the AST-based edit distance would correctly account for this and return a high distance value. We have implemented a module to create an AST for JavaScript units. We use the resulting AST module to define a tree-based edit distance measure between two JavaScript units. Asides from the tree-based distance measure, a lexical-token based edit distance algorithm is also implemented.

`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.

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 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 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.

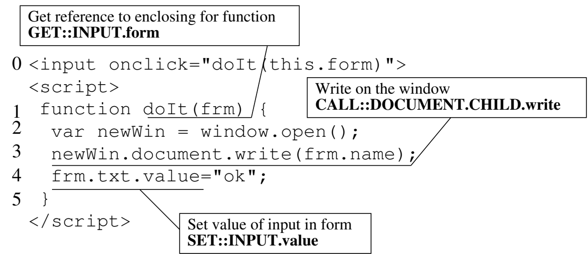

Figure 3: Types of DOM object references.

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 `frm.txt.value` object 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.

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

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

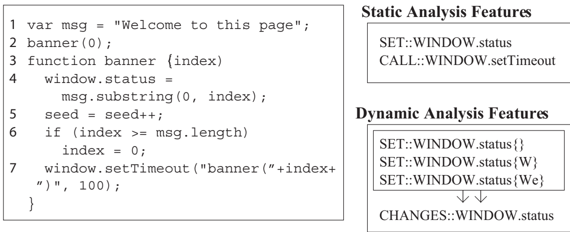

Figure 4: Sample JavaScript unit (l), along with features extracted by static and dynamic analysis (r).

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" likely to be trigger a unit whose class is to *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.

4.3.2 Dynamic analysis Static analysis is not able to recover certain information that occurs at run time. Dynamic analysis (*i.e.*, mock execution) 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.

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 `banner(index)` variable. In this example, dynamic analysis also recovers the fact that the variable's value is changing, hence a new feature, CHANGES::WINDOW.status, is added to the feature set.

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 our feature sets contain over 10,000 features. 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.

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.

At first glance, the performance improvement seems marginal, only contributing a few percentage points. We are encouraged as 1) our dataset is large (1.6K instances), 2)

ten-fold cross-validation is used and 3) the classification is difficult, consisting of 32 classes. 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, 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. Finally, 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.

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 agreement. 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 an 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.

Rather than treat 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 can be greatly improved by utilizing program analysis. By careful lexical analysis, 10% of errors are eliminated. Further improvements using static analysis and mock execution results in a 52% overall reduction of categorization error error. While our results preliminary, we believe they provide evidence that program categorization can benefit from adapting work from program analysis.

We plan extend this work to other scripting languages and decompiled plug-ins appearing on Web pages. We aim to characterize the information conveyed by these devices to assist end users to filter irrelevant material and to help summarize such information for users to make more informed Web browsing and searching choices.

References

Baxter, I. D.; Yahin, A.; Moura, L. M. D.; Sant'Anna, M.; and Bier, L. 1998. Clone detection using abstract syntax trees. In *ICSM*, 368–377.

Blazy, S., and Facon, P. 1998. Partial evaluation for program comprehension. *ACM Computing Surveys* 30(3).

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.

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 Intelligence (ICTAI'00)*, 46.

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*.

Rowe, N., and Laitinen, K. 1995. Semiautomatic disabbreviation of technical text. *Information Processing and Management* 31(6):851–857.

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.

Wong, W.-C., and Fu, A. W.-C. 2000. Finding structures of web documents. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*.