Predicting Reuse of End-User Web Macro Scripts

Chris Scaffidi[1], Chris Bogart[2], Margaret Burnett[2], Allen Cypher[3], Brad Myers[1], Mary Shaw[1]

[1] Carnegie Mellon University {cscaffid, bam, mary.shaw} @cs.cmu.edu

[2] Oregon State University {bogart, burnett} @eecs.oregonstate.edu

[3] IBM Research-Almaden acypher @us.ibm.com

Abstract

Repositories of code written by end-user programmers are beginning to emerge, but when a piece of code is new or nobody has yet reused it, then current repositories provide users with no information about whether that code might be appropriate for reuse. Addressing this problem requires predicting reusability based on information that exists when a script is created. To provide such a model for web macro scripts, we identified script traits that might plausibly predict reuse, then used IBM CoScripter repository logs to statistically test how well each corresponded to reuse. We then built a machine learning model that combines the useful traits and evaluated how well it can predict four different types of reuse that we saw in the repository logs. Our model was able to predict reuse from a surprisingly small set of traits. It is simple enough to be explained in only 6-11 rules, making it potentially viable for integration in repository search engines for end-user programmers.

1. Introduction

End-user programmers often solve problems by adapting, extending or otherwise reusing existing solutions. For example, an end-user programmer might copy an existing script that automates some computation, then modify it to automate a slightly different computation.

An essential step for this adaptive reuse is *identifying* useful code. To be useful, the code must be *relevant* to the programmer's task, and it must be *reusable* enough for the programmer to get some benefit from it. In particular, even if a piece of code is exactly relevant to the needs of the programmer, but it lacks the qualities needed to support reuse—perhaps because its functionality is not understandable or not reliable—then the code is not useful.

Repositories provide mechanisms to help users share code. Yet while today's repositories offer some support for identifying *relevant* code, they offer poor support for identifying *reusable* code.

One large repository of end-user code, IBM's CoScripter wiki [12], illustrates these limitations. With the CoScripter repository, users can publish and search for scripts that automate browser interactions with web sites. For example, one search feature locates scripts based on keyword queries, and another finds scripts that are relevant to a user-specified web site. But the repository provides users with little *reusability*-associated information, relying mainly on download counters and ratings that previous users gave to scripts. For example, a skilled user might publish an excellent script that automates a common, complicated task (such as searching several sites to find the cheapest airfare), with meticulously-commented code – and yet with today's repository features, this script would remain buried amid thousands of other scripts until it is "discovered", downloaded, and rated.

We would like to address this lack of reusability information in order to help users find hidden gems in the repository. For example, we would like the repository to sort search results based not only on keyword-based relevance but also on some reliable measure of probable reusability. Moreover, because a script might be reusable in some ways but not in others (e.g.: readable but buggy), we would like the repository to provide succinct explanations of its recommendations, so users can judge recommendations. Finally, we would like to accomplish this when the script is new and undiscovered, well before ratings and downloads occur.

As a first step, in this paper we explore how accurately we can predict whether a newly created web script will be reused. In essence we ask, "What kind of reuse predictions can we make at the time the code is created?"

Toward this end, Section 2 introduces web macro scripting and considers the applicability of existing models for evaluating reusability. Section 3 identifies script traits that each empirically correspond to reuse of scripts. Sections 4 and 5 present and evaluate a machine learning model that combines script traits to predict reuse. Section 6 summarizes our findings.

978-1-4244-4876-0/09/$25.00 ©2009 IEEE

2. Related work

Web macro scripting is a form of end-user programming that “mashes up” information from web sites. IBM’s CoScripter tool (formerly called Koala) records users’ actions in Firefox as re-playable scripts [12]. For example, a user might record a script that submits a form at www.aa.com to look up a flight’s status. The author can edit the script, possibly giving it a title, modifying recorded instructions, changing literal strings to variable references (read at runtime from a per-user configuration file called the Personal Database), and can insert “mixed-initiative” instructions that cause CoScripter to pause at runtime while the user makes decisions and performs actions manually.

All scripts are stored on a wiki so other users can run, edit, or copy-and-customize them. With scripts created by over 6000 users, the CoScripter repository is one of the largest repositories of end-user code on the web.

The CoScripter repository’s user interface provides some information about the reusability of each script: the number of users who have downloaded it, the average rating that it has received from users, the number of users who have “favorited” it, and reviews/comments that users typed about the script. Figure 1 illustrates how the repository displays “favoriting” information; other screens show other information.

These pieces of information are popularity measures requiring that someone previously tried out a script. Relying on popularity as an indicator of reusability presents a “catch-22”: somebody must try to reuse a script before it can be found to be reusable. This limits the usefulness of popularity as a “predictor” of reusability—in fact, only 68 of the 3754 public scripts currently in the repository have any ratings at all.

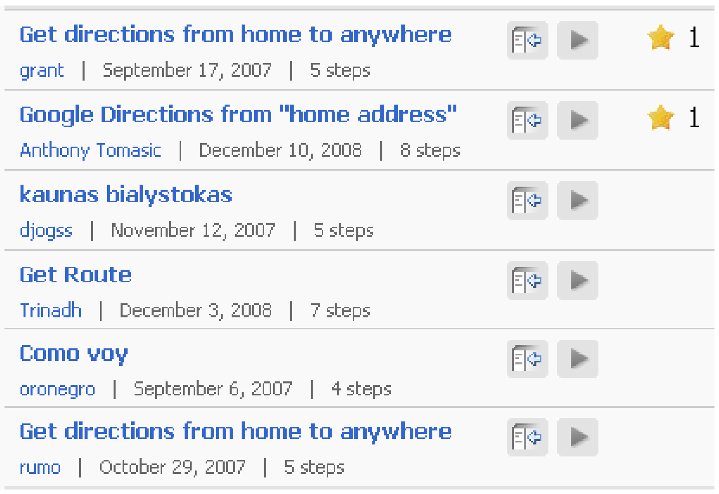
Other end-user programming repositories share the same circularity. For example, the Matlab File Exchange repository shows download counters, ratings, and reviews [7]. The Forms/3 spreadsheet repository helps users evaluate code by letting them interactively try out spreadsheets—again, users cannot see how reusable code is until after they have tried to reuse it [19].

Even some repositories for professional programmers have this circularity. For instance, repositories based on collaborative filtering issue recommendations of the form, “People like you found the following components to be helpful” [13], which requires that some people try code before it can be recommended to other people.

Some repository features for professional programmers do not require prior uses of code in order to evaluate reusability. However, for the most part, these features depend on information that is unavailable in the end-user programming context. For example, many repositories for professional programmers recommend components based on call graphs, inheritance hierarchies, method signatures, and similar information based on types (e.g.: [8]). Others use call graphs and code complexity metrics to predict whether code contains defects [3][15] or use topological structural dependencies to identify sections of code that are related to one another [16]. Still others rely on documentation in order to find or recommend reusable code [5]. Such approaches are appropriate in repositories for Java-like code but less so for CoScripter scripts, which do not contain loops, call each other, inherit from each other, have static types, or carry any significant documentation aside from comments and script titles.

There is one approach that might apply to end-user scripting: predicting that code will have high reusability if the author previously created code that was reused, indicating that the author has high expertise [18]. This approach’s information requirements are not very restrictive, as a component’s reusability can be predicted even if it has never been reused, as long as someone previously tried to reuse one of the programmer’s other components. In subsequent sections, we include information of this kind in our model of web script reuse.

Research shows that in addition to repository features, other factors promote code reuse by professional programmers. These factors include budgeting time for careful design of code for reuse, long-term backing from management, and systematic consideration of how to support reuse during each phase of development processes [10]. The CoScripter repository offers an opportunity to empirically study what factors affect reuse in the end-user programming context, which often is opportunistic (with little up-front design for reuse), independent (with minimal centralized management), and informal (with no clearly specified development process).


Get directions from home to anywhere ★ 1
grant | September 17, 2007 | 5 steps

Google Directions from "home address" ★ 1
Anthony Tomasic | December 10, 2008 | 8 steps

kaunas bialystokas
djogss | November 12, 2007 | 5 steps

Get Route
Trinadh | December 3, 2008 | 7 steps

Como voy
oronegro | September 6, 2007 | 4 steps

Get directions from home to anywhere
rumo | October 29, 2007 | 5 steps

Figure 1: Searching with the word “directions” yields six scripts, two of which were “favorited” by one user each

3. Script traits that correspond to reuse

To identify candidate script traits corresponding to reuse, we began with the four key steps involved in reuse: finding, understanding, modifying, and composing code [1]. In our context, scripts rarely call one another, so composability is unlikely to play a major role, and while high modifiability is desirable, scripts are probably more reusable if they need no modification.

Thus, predicting reuse at script creation-time requires information that reflects the findability, understandability, and need to modify scripts prior to reuse. Based on these criteria, we identified promising script traits and tested how well each empirically corresponded to reuse.

3.1 Candidate script traits

We identified 35 candidate traits in 8 categories (Table 1):

- *Mass appeal*: Scripts might be more likely to be found if they reflect the interests of many users, as reflected by website URLs and other tokens in scripts. Promotion of a script as a tutorial (on the homepage) might also improve findability.
- *Language*: Scripts might be more understandable if their data and target web sites are written in the community's primary language (English).
- *Annotations*: Code comments and proper script titles might increase understandability.
- *Flexibility*: Parameterization and use of mixed-initiative instructions might increase the flexibility of scripts, reducing the need for modification.
- *Length*: Longer code requires more effort for understanding and tends to have more defects [14] that might require fixing. (Conversely, longer scripts contain more functionality, which may increase reuse.)
- *Author information*: Early adopters, IBM employees, online forum participants, and users who already created reused scripts might tend to produce bug-free scripts that can be reused with minimal modification.
- *Advanced syntax*: The use of advanced keywords might suggest that script authors were experts who could produce bug-free, reusable scripts.
- *Preconditions*: In a prior study, reuse seemed lower for scripts with preconditions such as requiring Firefox to be at some URL prior to execution, requiring users to log into a site prior to execution, or requiring many sites to be online during execution [4]. Preconditions might limit flexibility and impede reuse.

3.2 Data sources and measures of reuse

We extracted six months of logs from the CoScripter repository, which is primarily used by non-IBM employees. (IBM employees mainly use a small company-internal repository instead.) In addition, we retrieved the initial source code from version control for each public script in that period (yielding 937 scripts). We considered four forms of reuse:

- *Self-exec*: Did the script author ever execute the script between 1 day and 3 months after creating it? (We omitted executions within 1 day, as such executions could relate to creating and testing a script, rather than reuse per se.) {17% of all scripts met this criterion}
- *Other-exec*: Did any other user execute the script within 3 months of its creation? {49%}
- *Other-edit*: Did any other user edit the script within 3 months of its creation? {5%}
- *Other-copy*: Did any other user copy the script to create a new script within 3 months of the original script's creation? {4%}

We chose binary measures instead of absolute counts for three reasons. First, by default, the wiki sorts scripts by the number of times that each was run. This seems to cause an information cascade [2]: oft-run scripts tend to be reused very much more in later weeks. Second, scripts can recursively call themselves (albeit without parameters). Third, some users also apparently set up non-CoScripter programs to run scripts periodically (e.g.: once per day). These three factors cloud the meaning of absolute counts. When computing reuse measures, we considered a script to be reused if a periodic program ran it. However, we examined the logs to find automated spiders that executed (or even copied) many scripts in a short time, and we filtered out those events.

Our 3 month post-creation observation interval was a compromise. Using a short interval risks incorrectly ruling a script as un-reused if it was only reused after the interval. Selecting a long interval reduces the number of scripts whose observation interval fell in the 6 months of logs. Selecting half of the log period as the observation interval resulted in few cases (20) of erroneously ruling a script as un-reused yet yielded over 900 scripts.

3.3 Testing script traits

For each trait, we divided scripts into two groups based on if each script had an above-average or below-average level of the trait (for Boolean-valued traits, treating "true" as 1 and "false" as 0). For each group's scripts, we computed the four reuse measures. Finally, for each combination of trait and reuse measure, we performed a one-tailed z-test of proportions. In cases where the correspondence between script trait and reuse measure was actually opposite what we expected, we also report whether a one-tailed z-test would have been significant in the opposite direction.

Table 1: Each row shows one script trait tested for correspondence to four kinds of reuse. Traits are sorted in the order discussed in Section 3.1. A + (−) hypothesis Hyp indicates that we expected higher (lower) levels of reuse to correspond to the trait (e.g.: for *keywrd_sim*, we expected if a script had many tokens in common with other scripts, then it is more likely to be reused). **Non-empty empirical results** indicate statistically significant differences in reuse, with + (−) indicating that higher (lower) reuse corresponds to higher levels of the trait. One + or − indicates one-tail significance at p<0.05, ++ or −− indicate p<0.01, and +++ or −−− indicate p<0.00036 (which is the cutoff corresponding to a Bonferroni correction of p<0.05). The shaded cells were particularly useful in our predictive model, as explained later in Section 5.2.

Trait				Empirical Results			
Catg	Name	Meaning	Hyp	Self Exec	Other Exec	Other Edit	Other Copy
Mass appeal	*keywrd_sim*	real: normalized measure of how many scripts contain the same tokens as this script	+	+			
	urldom_sim	real: normalized measure of how many other scripts contain the same URL domains as this script	+		+++	++	+
	ip_urls	int: # of URLs in script that use numeric IP addresses	−				
	inet_urls	int: # of hosts referenced by script that seem to be on intranets	−		−		
	tutorial	bool: true if script was created for tutorial list	+		+	+++	+++
Language	*us_urls*	int: # of US URLs in script	+	+	+		
	nonus_urls	int: # of non-English words in literals + # of URLs outside USA	−	−		−	−−
	no_urls	bool: true if *nonus_urls* and *us_urls* are each 0	−		−−−		
	nonroman	pct: % of non-whitespace chars in title or content that are not roman	−		−	−	
Annotations	*comments*	int: # of comment lines	+	+	++	+++	+++
	test_title	bool: true if script title contains the word "test"	−	−−			
	copy_title	bool: true if script title contains the phrase "Copy of"	−	−−	−		
	titled	bool: true if script has a title	+	+++	++		
	punct_title	bool: true if script title contains punctuation other than periods	−	−			
Flex.	*params*	int: # of parameters (configuration variables) read by script	+	++	+	+++	+++
	literals	int: # of literal strings hardcoded into script	+	+++			
	mixed_init	int: # of mixed-initiative “you manually do this” instructions	+		+	+	++
Length	*code_lines*	int: total # of non-comment lines in script	−	++			
	total_lines	int: total # of lines (*code_lines* + *comments*)	−	+			
	distinct_lines	int: total # of distinct non-comment lines in script	−	+++			++
Author	*author_id*	int: id of the user who authored the script (lower for early adopters)	−	+++	−−−	−−	−−−
	script_id	int: id of the script (tends to be lower for early adopters)	−		−−−	−−−	−−−
	ibm	bool: true if script's author was at an IBM IP address	+	++		+++	+++
	forum_posts	int: # of posts by the script author on the CoScripter forum	+	++		++	
	loauth_name	bool: true if script author’s name starts with punctuation or 'A'	+		−		
	prev_created	int: # of scripts by same author that were created prior to this script	+	+++	−−−		
	prev_selfexec	int: # of scripts by same author that were executed by author prior to this script's creation	+	+++	−−−	−−	
	prev_otherexec	int: # of scripts by same author that were executed by other users prior to this script's creation	+	−−−	+++	−	−
	prev_otheredit	int: # of scripts by same author that were edited by other users prior to this script's creation	+	−−−	+++	−	−
	prev_othercopy	int: # of scripts by same author that were copied by other users prior to this script's creation	+	−−−	+++	−	−
Adv Syn	*ordinals*	bool: true if script uses ordinals (eg: “third”) to reference form fields	+	+++			
	ctl_click	bool: true if script uses "control-click" or "control-select" keywords	+		+		+++
Pre-cond.	*assume_url*	bool: true if first line of script is not a "go to URL" instruction	−		−−−		
	assume_login	bool: true if script contains "log in", "logged in", "login", or "cookie"	−				
	distinct_hosts	int: # of distinct hostnames in script’s URLs	−		+++		

We report statistical significance at several levels, including a level based on a Bonferroni correction that compensates for the large number of tests (140). Some traits are not statistically independent, nor are the measures of reuse. Thus, the correction establishes a lower-bound on the statistical significance of results.

In terms of robustness, we noted that many traits are count integers that could be normalized by script length. We tested these traits twice, once with the traits in Table 1, and again with length-normalized traits. In virtually every case, results were identical.

3.4 Results and discussion

Most traits corresponded to reuse as hoped. Different reuse measures corresponded to different traits, suggesting that different kinds of reuse occur for different reasons. For example, scripts created by apparent experts were more likely to be run by other users, but less likely to be edited—perhaps such scripts worked properly and rarely needed tweaks. Thus, rather than combining script traits into one model that attempts to accurately predict all possible forms of reuse, we need a generalized model that can be instantiated for each measure of reuse.

Length traits corresponded to higher likelihood of reuse by the script author, though this was not a complete surprise. Empirically, the increased functionality in longer scripts seemed to outweigh any risk of increased defects. This reveals the importance of code's usefulness, in addition to findability, understandability, and modifiability (or lack of need for modification).

These statistical tests show that it is possible to find traits that correspond to reuse and are automatically computable when a script is created. While we do not claim to have found a definitive set of all traits that might correspond to reuse, we do have an appropriate set of traits for use in constructing a predictive model of reuse.

4. Prediction of reuse

In order to predict reuse, we developed a model that evaluates how well scripts satisfy arithmetic rules that we call "predictors". We present a machine learning algorithm that selects rules that predict the reuse measure used during training. For example, one predictor might be that the number of comments *comments* ≥ 3, another might be that the number of referenced intranet sites *inet_urls* ≤ 1, and a third might be *titled* ≥ 1. Ideally, predictors are true for every reused script and false for every un-reused script.

After the first algorithm selects a set of predictors, a second algorithm uses this set to predict if some other script will be reused. It counts how many predictors the script matches and predicts that the script will be reused if it matches at least a certain number of predictors. Continuing the example above, requiring at least 1 predictor match would predict that a script will be reused only if *comments* ≥ 3 or *inet_urls* ≤ 1 or *titled* ≥ 1.

After describing the algorithms for training and using this model, we evaluate it by comparing its quality to that of several machine learning models commonly used on other software engineering problems.

4.1 Training and using the predictive model

Our algorithm trains a predictive model (Figure 2). Its inputs are a training set of scripts R', real-valued traits C defined for each script, a measure of reuse m that tells if each script is reused, and a tunable parameter α described below. The output of our algorithm is a set of predictors Q.

In the first of three stages, the algorithm determines if each script trait c_i corresponds to higher or lower reuse. To do this, the algorithm places scripts into two groups (R_m and $\overline{R_m}$), depending on if each script was reused according to m. It compares the proportion of reused scripts with an above-average value of c_i to the proportion of un-reused scripts with an above-average value of c_i. If higher levels of c_i correspond to lower reuse, the algorithm multiplies the trait by -1, so higher levels of adjusted traits a_i correspond to higher levels of reuse.

Second, for each adjusted trait a_i, the algorithm finds the threshold τ_i that best distinguishes between reused and un-reused scripts. The selected value of τ_i maximizes the difference between the proportion of reused scripts that have an above-threshold level of adjusted trait versus the proportion of un-reused scripts that have an above-threshold level of adjusted trait.

Finally, the algorithm creates a predictor for each adjusted trait that is relatively effective at distinguishing between reused and un-reused scripts. As noted above, each predictor should ideally match every reused script but not match any un-reused scripts. Of course, achieving this ideal is not feasible in practice. Instead, a predictor is created if the proportion of reused scripts that have an above-τ_i amount of the adjusted trait is at least a certain amount α higher than the proportion of un-reused scripts that have an above-τ_i amount of the adjusted trait.

This minimal difference α between proportions in the reused and un-reused groups is a tunable parameter. Lowering α allows more adjusted traits to qualify as predictors, which increases the amount of information captured by the model but might let poor-quality predictors enter the model. This is a typical over-training risk in machine learning, so we must evaluate the empirical impact of changing α.

After training, predicting if a new script will be reused requires testing each predictor (Figure 3). If the script matches at least a certain number of predictors β, the model predicts that the script will be reused.

The minimal number of predictors β is tunable. Lowering β increases the number of scripts predicted to be reused, decreasing the chance that useful scripts slip by. However, lowering β risks erroneously predicting that un-reused scripts will be reused. As with α, this leads to a trade-off typical of machine learning, but in the case of β, the trade-off is directly between false negatives and false positives. As with α, we must evaluate the empirical impact of changing β.

```
TrainModel
  Inputs: Training scripts $R' \subseteq$ Repository R,
            where R is a set of scripts
          Script traits $C = \{c_i : R \rightarrow [0, \infty)\}$
          Measure of reuse $m : R \rightarrow \{0,1\}$
          Minimal proportion difference $\alpha \in (0,1)$
  Outputs: Predictors $Q = \{ q_i : R \rightarrow \{0,1\} \}$

Let the reused script set $R_m = \{s \in R' : m(s) = 1\}$
Let the un-reused script set $\overline{R}_m = \{s \in R' : m(s) = 0\}$
Let $p(S, c, \tau) = |\{s \in S : c(s) \geq \tau\}| / |S|$
Initialize Q to an empty set of predictors
For each $c_i \in C$,
  Let $\mu_i = \sum_{s \in R'} c_i(s) / |R'|$
  Let adjusted trait
    $a_i(s) = c_i(s)$ if $p(R_m, c_i, \mu_i) \geq p(\overline{R}_m, c_i, \mu_i)$
    or $a_i(s) = - c_i(s)$ otherwise
  Compute threshold (through exhaustive search)
    $\tau_i = \text{argmax } p(R_m, a_i, \tau) - p(\overline{R}_m, a_i, \tau)$
  If $p(R_m, a_i, \tau_i) - p(\overline{R}_m, a_i, \tau_i) \geq \alpha$
    then add the following predictor to Q...
    $q_i(s) = 1$ if $a_i(s) \geq \tau_i$ and 0 otherwise
Return Q
```

Figure 2. Selecting predictors during training

```
EvalScript
  Inputs: One script $s \in$ Repository R
          Minimal predictor matches $\beta \in (0, |Q|]$
          Predictors $Q = \{ q_i : R \rightarrow \{0,1\}\}$
  Outputs: Prediction of reuse $\in \{0,1\}$

Let nmatches = 0
For each $q_i \in Q$
  If $q_i(s) = 1$ then nmatches = nmatches + 1
If nmatches ≥ β then return 1 else return 0
```

Figure 3. Predicting if a script will be reused

5. Evaluation

Ten-fold cross-validation is the usual method used when training and evaluating a model on the same data set. We used this approach to compare the quality of our model with the quality of model variants, as well as the quality of other machine learning models, all of which attempt to predict reuse solely based on information available when a script is created.

5.1 Quality measures

Our approach of training a model on traits to predict reuse resembles the approach used in software engineering defect prediction, which trains a model on module traits to predict the presence of defects [3]. The primary quality measures used in that literature are False Positive (FP) and True Positive (TP) rates:

$$TP = \frac{\text{\# reused scripts correctly predicted as reused}}{\text{\# reused scripts}}$$

$$FP = \frac{\text{\# un-reused scripts erroneously predicted as reused}}{\text{\# un-reused scripts}}$$

TP is the same as the recall measure used in information retrieval, indicating the fraction of interesting items (reused scripts) correctly identified. FP indicates the fraction of uninteresting items (un-reused scripts) erroneously identified; it is similar in purpose to the precision measure (fraction of identified items that are correctly identified as interesting) used in information retrieval to quantify prediction specificity, but FP is often preferred over precision in software engineering, since FP is more robust than precision to small changes in the experimental data [14].

5.2 Evaluating model quality

For each reuse measure, we performed 10-fold cross-validation at varying levels of α and β (Figure 4).

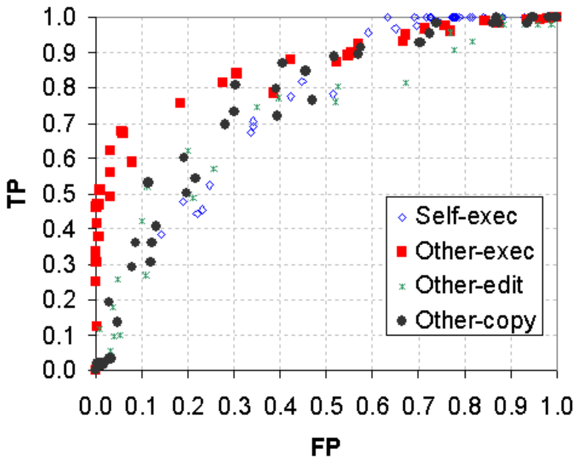


Figure 4. Model quality at various parameter values

The resulting level of prediction quality was comparable to that of other research that applies machine learning to software engineering problems. For example, TP ≤ FP + 0.3 for many defect prediction models

[15]. At this level of quality, such algorithms are adequate for focusing programmers' attention on particular pieces of code, and the programmers can then evaluate for themselves whether the code actually is worthy of further attention. As our algorithm achieved similar quality, we are optimistic that it will prove useful for focusing end-user programmers' attention on scripts that might be worthy of further attention for reuse.

To determine if our simple algorithm for combining predictors yielded quality as good as more complex algorithms, we considered alternative comparison models: logistic regression, Naïve Bayes, and J48 decision trees. We selected these because they have proven useful in prior research on defect prediction [15]. However, deeply-nested models like those generated from Naïve Bayes and J48 have been found in experiments to be unintuitive to users [9][17], so we also compared our algorithm to one that generates fairly explainable models. Specifically, we compared to PART, a "straightforward and elegant" algorithm [6] that produces relatively flat and easy-to-interpret rule sets, much like our own algorithm. We used the Weka machine learning toolkit implementation of the algorithms [20], which exposes few tunable parameters for models, so we recorded the quality measures using model designers' suggested (default) parameters (Table 2).

Table 2. Quality measures of comparison models

	Logistic Regress.		Naïve Bayes		J48		PART	
	FP	TP	FP	TP	FP	TP	FP	TP
Self-exec	.04	.17	.70	.97	.07	.30	.35	.11
Other-exec	.13	.76	.03	.56	.14	.78	.78	.21
Other-edit	.01	.11	.71	.84	.01	.13	.16	.02
Other-copy	.01	.19	.17	.55	.01	.19	.17	.02

Comparing these results to the TP scores of our model at corresponding values of FP (Figure 4), we found negligible differences in quality between our model and these alternative models (or, in a few cases, our algorithm produced somewhat better results).

Moreover, models generated by our algorithm were simpler than even those produced by the most explainable comparison algorithm, PART. Specifically, PART generated rule sets of between 18 and 55 rules. In contrast, our algorithm generally produced between 6 and 11 rules (as discussed below). Thus, our model not only achieves high quality, but it does so using a relatively small set of rules, which should enhance the explainability of recommendations to users.

To evaluate whether the quality of our model's predictions were purely due to the binary nature of our reuse measures, we evaluated how the number of reuses related to script traits. In particular, we set $\alpha = 0.07$ so that almost all scripts matched at least one predictor (TP ≈ 0.98) and plotted the average number of reuses as a function of predictor matches (nmatches in Figure 3). We found that the number of execution, edit, and copy actions by users other than the script's author generally rose sharply with the number of predictors matched (Figure 5), although there was an odd drop in execution by other users at 10 or more matches. Reuse by the script author showed no identifiable trend when compared to number of matches. These results generally affirm that the traits really do contain information about reusability and that our results are not simply due to the binary nature of our measures of reuse.

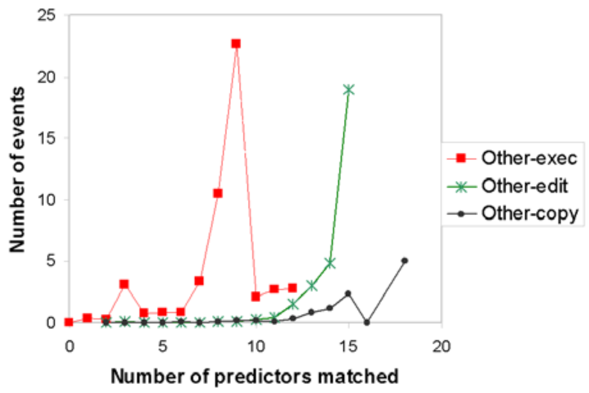


Figure 5. Absolute level of reuse rose sharply with the number of matches

To evaluate which traits were most useful for predictions, we shaded cells in Table 1 where script traits were active at $\alpha = 0.16$ (FP = 0.4). At this level, 17 script traits were active, including 6 to 11 for each measure of reuse. The most useful were in the Mass appeal, Length, and Author categories, with minor contributions from the Language, Annotation, and Flexibility categories. Other traits generated active predictors at lower levels of α. Most of these traits had shown strong statistical correspondence to reuse. This match is not perfect, however, as the z-score used in the statistical tests depends not only on differences in proportions but also the absolute sizes of proportions. This is common in statistics: differences might be statistically significant yet not meaningful. This disparity is most likely to occur when almost all scripts or almost no scripts have a particular trait (e.g.: *tutorial*, *forum_posts, ibm, ctl_click*, etc), which limits these traits' usefulness for making predictions.

6. Discussion and future work

In this paper, we have presented a model that can accurately predict whether a web script will be reused by the script's author or other end-user programmers.

Perhaps the biggest (pleasant) surprise produced by this work is that the model predicted reuse so well based on such a small amount of information. That is, it is not surprising that each script characteristic individually was related to reuse—rather, the surprise is that the synthesis of these characteristics together provides such accurate predictions of reuse.

Since our target audience is end-user programmers, it is particularly encouraging that we did not need to use an incomprehensible, labyrinthine machine learning model to predict reuse. Our model's predictions can be succinctly explained by listing just a handful of unnested rules and indicating which were satisfied or violated.

Possible applications of the model include a range of new features for script repositories, thereby helping to guide users to code that they can reuse. For example, we could sort search results based on a weighted combination of the relevance and reusability, with field testing to evaluate different approaches for weighting these two factors. However, just because a script is reused does not necessarily mean that the reuse was a success. Consequently, as we develop new repository features based on our model, we will need to evaluate how well guiding users to likely-to-be-reused code actually helps them to complete their programming goals. For example, we could run our model on IBM's private (internal) repository to identify likely-to-be-reused scripts, then interview workers who have already used those scripts, in order to understand the results of reusing those scripts.

In the longer term, we would like to help end-user programmers reuse other kinds of code beyond web scripts. This will require generalizing the model that we have developed and finding ways to apply it to spreadsheets, JavaScript or other kinds of end-user programming. We will also analyze whether the model can accurately predict whether code will be repeatedly or often reused. We look forward to these and other opportunities to help end users find code that is not only relevant to their programming problems, but also reusable enough to provide practical benefits.

7. Acknowledgements

This work was supported by the EUSES Consortium via NSF ITR-0325273, by NSF grants CCF-0438929 and CCF-0613823, and by an IBM International Faculty Award. Opinions, findings, and recommendations are the authors' and not necessarily those of the sponsors.

8. References

[1] T. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions, *IEEE Software (4)*, 2, March 1987, 41-49.

[2] S. Bikhchandani, D. Hirshleifer, and I. Welch. A Theory of Fads, Fashion, Custom, and Cultural Change as Informational Cascades, *J. Political Economy (100)*, No. 5, 1992, 992-1026

[3] G. Boetticher, et al. 4th Intl. Workshop on Predictor Models in Software Engineering. *Companion Proc. 30th Intl. Conf. Software Eng.*, 2008, 1061-1062.

[4] C. Bogart, et al. End-User Programming in the Wild: A Field Study of CoScripter Scripts, *2008 IEEE Symp. Visual Lang. and Human-Centric Computing*, 2008.

[5] D. Čubranić and G. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. *25th Intl. Conf. Software Eng.*, 2003, 408-418.

[6] E. Frank and I. Witten. Generating Accurate Rule Sets Without Global Optimization, *15th Intl. Conf. on Machine Learning*, 1998, 144-151.

[7] N. Gulley. Improving the Quality of Contributed Software and the MATLAB File Exchange, *2nd Workshop on End User Software Eng.*, 2006, 8-9.

[8] G. Gui and P. Scott. Coupling and Cohesion Measures for Evaluation of Component Reusability, *2006 Intl. Workshop on Mining Software Repositories*, 2006, 18-21.

[9] U. Johansson, L. Niklasson, and R. Konig. Accuracy vs. Comprehensibility in Data Mining Models, *7th IEEE Intl. Conf. on Information Fusion*, 2004, 295-300.

[10] R. Leach. *Software Reuse: Methods, Models, and Costs*, McGraw-Hill, 1997.

[11] G. Leshed, et al. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise, *26th SIGCHI Conf. on Human Factors in Computing Sys.*, 2008, 1719-1728

[12] G. Little, et al. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *25th SIGCHI Conf. Human Factors in Computing Sys.*, 2007, 943-946.

[13] F. McCarey and M. Cinneide. Rascal: A Recommender Agent for Agile Reuse. *Artif. Intell. Rev. (24)*, No. 3-4, 2005, 253-276.

[14] T. Menzies, et al. Problems with Precision: A Response to Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors', *Trans. Software Eng. (33)*, No. 9, 2007, 637-640.

[15] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction, *30th Intl. Conf. Software Eng.*, 2008, 181-190.

[16] M. Robillard. Automatic Generation of Suggestions for Program Investigation. *10th European Software Eng. Conf.*, 2005, 11-20.

[17] S. Stumpf, et al. Toward Harnessing User Feedback for Machine Learning, *12th Intl. Conf. on Intelligent User Interfaces*, 2007, 82-91.

[18] G. Suryanarayana, et al. Architectural Support for Trust Models in Decentralized Applications, *28th Intl. Conf. Software Eng.*, 2006, 52-61.

[19] R. Walpole and M. Burnett. Supporting Reuse of Evolving Visual Code, *1997 Symp. Visual Lang.*, 1997, 68-75.

[20] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2005.