Digging for diamonds: Identifying valuable end-user code in repositories

Jarrod Jackson[1], Kathryn Stolee[2], Christopher Scaffidi[1]
[1] *Oregon State University,* [2] *University of Nebraska-Lincoln*
jacksoja@engr.orst.edu, kstolee@cse.unl.edu, cscaffid@engr.orst.edu

Abstract

To a large extent, repositories of end-user code are "write-only": much of the code that people publish never sees substantial reuse. Yet buried within these repositories are valuable pieces of code, though finding them is not always easy. In prior work, we developed a model that can predict, when a web macro is created, whether that script will be reused by anybody. In the current paper, we analyze data from two other end-user repositories to investigate the model's generalizability to other kinds of code. We find that the model performs well for a wide range of different purposes and configurations, including predicting future reuse events based on data about past events, indicating that the model could serve as an effective basis for future repository enhancements.

1. Introduction

By definition, end-user programmers primarily create code for their own use, yet this does not preclude also sharing code so other people can reuse it. Code reuse takes many forms, ranging from simply running an existing program all the way to copying and perhaps customizing code to create a new program.

Empirical studies give a mixed picture of the extent to which end-user programmers actually reuse each other's code in practice. On one hand, some studies mention stories of how reuse helped people to be more productive, successful, or creative [7][9]. Yet despite these specific cases, other studies suggest that code reuse among end users remains low. For example, web developers rarely reuse each other's code, even though they frequently reuse images and other non-code content [10]; scientists rarely reuse each other's code, even though they have some of the strongest programming skills of all end users [12].

Studies show that reuse is low even in repositories specifically aimed at helping users to share and reuse code. For example, in the Scratch repository of animations, only 15% of programs have been created by reusing code [9]. But Scratch is one of the most successful end-user repositories. In the CoScripter public repository of web macros (scripts that automate web browsing actions), only 4% of scripts were ever copied into new scripts [11]. In contrast, except in specific industries such as aerospace, nearly 50% of professional programmers' code is *typically* composed from existing code [3], with even higher levels possible if a body of highly-reusable code is systematically cultivated [5].

In contrast, repositories of end-user code are not so cultivated but rather contain a mixture of valuable code with large amounts of much less valuable code. As the manager of one repository laments, "many of these files are poorly written (uncommented spaghetti code) or poorly motivated (homework problems of no general interest)" [4]. His comments reveal three *traits* that apparently impede reuse: the absence of comments, the presence of code complexity, and topical obscurity or insularity. Not only do these and other traits appear to directly interfere with reuse, but once a repository is cluttered with "a proliferation of worthless code" [4], it indirectly becomes difficult to find *good* code buried in the repository. The image of "digging for diamonds" metaphorically describes that challenge: helping users to identify valuable code in a repository, so that they can more *consistently* benefit from reuse.

In prior work, we introduced an approach aimed at identifying reusable end-user code in repositories [11]. Specifically, we presented a machine learning model that used 35 traits of CoScripter web macro scripts to accurately predict whether scripts would be reused. These predictions might form the basis for cultivating more valuable repositories. For example, if a user tries to upload code that has traits associated with low reusability, the repository could suggest (and perhaps offer to automate) ways to tweak the code to increase reusability. Conversely, repositories could identify code that has traits of high reusability, so the code could be promoted in search engine results, and so a system administrator could highlight good code on the home page or a "Pick of the Week" blog [4].

Since designing, implementing, deploying and evaluating such tools would be significantly costly, it is worth investigating several questions whose answers might suggest ways to refine or extend the predictive model, as well as ways to shape the resulting tools:

Q1: How well does this approach generalize to other kinds of code? The prior work showed that the machine learning model could use web macro traits to accurately

predict reuse. How accurately can similar traits be used to predict reuse of *other* kinds of code?

Q2: How well does the approach generalize to predicting different levels of reuse? Prior work tested how well the model could predict whether code would be reused at all. But even if code is reused once, it still might not be especially valuable. Can the model also accurately identify code that will be used *many* times, revealing the true "diamonds" that are most valuable?

Q3: How well does the approach generalize to other indicators of value? Wholesale reuse is only one proxy or indication of value. Another is scavenging—reuse of pieces of code, rather than code in its entirety. Another is ratings or reviews of code. Can the model identify code that is valuable according to indications other than wholesale reuse?

Q4: How durable are the machine-learned models? Using these models to build tools assumes that it is possible to use past events to predict future events. How accurate is this assumption? Put another way, how much does the model accuracy degrade over time?

Q5: How much incremental benefit do various traits provide? The traits used by the model do not come for free: a program must be written to compute each trait that is fed into the machine learning model. Could traits be omitted without sacrificing model accuracy?

To answer these questions, we test the model using data from the Yahoo Pipes and UserScripts repositories, which contain end-user programs that transform RSS feeds and web pages (respectively). We find that the model can indeed accurately identify valuable end-user code in these repositories, using a flexible range of different measures of value and levels of reuse. The machine-learned models only degrade slightly in accuracy over time, indicating that tools could use past events to predict the future. We also identify a set of traits that provide most of the model's accuracy and require relatively little effort to compute.

Our paper is organized as follows. Section 2 summarizes related work aimed at supporting end-user code reuse. Section 3 describes the model and the new datasets for testing it. Section 4 assesses the approach's generalizability (Q1-Q3), Section 5 assesses models' durability (Q4), and Section 6 assesses traits' incremental value (Q5). Section 7 presents key conclusions and implications for future work.

2. Related work

Some of the newest and fastest-growing end-user code repositories contain thousands of web scripts. These are pieces of interpreted code that combine or transform data provided by web sites. Examples include the CoScripter web macro system, which lets users create scripts that automate browsing actions such as clicking on links and filling in forms [7]. Another example is Yahoo Pipes, which are scripts that retrieve lists of records via the RSS protocol from web sites, then combine and filter those records [16]. A third example is the UserScripts repository of "GreaseMonkey" web scripts, whose purpose is to alter and enhance the HTML of web sites (e.g., by removing advertisements) [2].

Search engines are the main approach for digging through the thousands of scripts that end users have uploaded to these and other repositories [4][9][14][15]. Code can also be tagged using a folksonomy [13], then browsed by category. These approaches identify code based on its content (reflected in keywords, tags or categories) rather than its value, so search results contain a mixture of scripts with varying levels of value. Our model complements these approaches in being aimed at identifying code based on indicators of value.

To help users focus on valuable code that they might want to reuse, search engines typically sort results by numbers of downloads (e.g., [4][7]). Yet the numbers of downloads is not available until after the code has been downloaded, making this approach no help in finding "diamonds in the rough." In contrast, our model uses only information available at the time of the code's creation, in order to identify diamonds not yet discovered.

Some repositories let users sort scripts based on ratings (e.g., [2][4][7]). However, almost no code is ever rated by these repositories' users; for example, right now, only 7% of UserScripts code has been rated. Thus, as with download counters, ratings provide no help with finding undiscovered gems. Moreover, as with most ratings by end users of online content [6], ratings of end-user code are skewed high; for example, right now, only 22% of rated UserScripts code is rated neutral or lower. This strong bias suggests that ratings provide an incomplete view of code reusability. Since almost all ratings are positive, we evaluate in this paper whether the model is useful for predicting which code will accrue ratings and provoke discussion; in addition, we complement this indicator of value with others (e.g., download counts).

Aside from user communities, a repository's other human element is the system administrator, who diligently tries to cultivate a body of highly valuable code. This typically includes reviewing code and promoting the best on the home page or a blog (e.g., [4][7][9]). However, this cultivation is limited, since system administrators do not have time to review every one of the hundreds of scripts uploaded every week. Ideally, the repository should be able to identify the most valuable scripts and bring them to the administrator's attention for consideration as worthy of promotion. Conversely, the repository might also identify the scripts that are least likely to be valuable, so that the administrator can consider removing them. In this paper, we evaluate whether our model is suitable for these purposes.

3. Approach

Motivated by the limitations in existing repositories, we aim to provide an automated machine learning approach for categorizing end-user code based on value, not just content. Our approach is intended to generalize over many kinds of end-user code, to classify new pieces of code even if they have never yet been downloaded or rated, and to identify the most valuable code just as well as it can identify the least valuable code.

Our model for identifying valuable code is a supervised machine learning model, meaning that it has two phases: training and usage. The training step infers a function that maps from objects to a category; in our case, the objects are scripts, while the output categories are binary measures of script value. For example, given the goal of identifying scripts that will be downloaded at least 1000 times, the training process will infer a function for identifying scripts that likely will meet this threshold. Later, the inferred function can be applied to other scripts to make predictions about their value.

In prior work, the model worked well for predicting whether CoScripter web macros would ever be reused [11]. Our goal in the present work is to assess how well this model generalizes to two other kinds of end-user code (Q1 from Section 1), how well it is able to distinguish between high- and low-value scripts at a range of different thresholds controlling what constitutes "high" or "low" value (Q2), whether it is able to accurately identify valuable code based on different indications of what qualifies as "valuable" (Q3), how much accuracy is lost when the machine learning model is trained on old scripts but used on newer scripts (Q4), and how much incremental accuracy is obtained as more information is available about scripts' traits (Q5). Answering these questions requires characterizing the traits of scripts (Section 3.1), identifying multiple thresholds for several indications of value (Section 3.2), and finally testing the model (Sections 4-6).

3.1 Script traits

Each script is represented as a collection of traits. During training, these are used to automatically generate constraints that are generally satisfied by valuable code but rarely by less valuable code (Figure 1). For example, *number_of_comments* ≥ 3 might be one inferred constraint, while *number_of_variables* ≤ 2 might be another. (The original algorithm included a step that discarded some traits from consideration [11], but we found that this did not affect model accuracy on average, so we now omit this step.) After "loading" the model with constraints during training, it can later be used to identify other code as high- or low-value (Figure 2). Specifically, a script is classified as "valuable" if it matches at least a certain number of constraints β.

TrainModel

Inputs: Training scripts $R' \subseteq$ Repository R

Script traits $C = \{c_i : R \to [0, \infty)\}$

Binary measure of value $m : R \to \{0,1\}$

Outputs: Constraint set $Q = \{ q_i : R \to \{\text{false}, \text{true}\} \}$

Let the set of high-value scripts $R_m = \{s \in R' : m(s) = 1\}$

Let the set of low-value scripts $\overline{R}_m = \{s \in R' : m(s) = 0\}$

Let $p(S, c, \tau) = \left|\{s \in S : c(s) \geq \tau\}\right| \Big/ \left|S\right|$

Initialize Q to an empty set of constraints

For each constraint $c_i \in C$,

Let $\mu_i = \sum_{s \in R'} c_i(s) \Big/ \left|R'\right|$

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)

$\tau_i = \text{argmax}\ p(R_m, a_i, \tau) - p(\overline{R}_m, a_i, \tau)$

Add this constraint to Q: $a_i(s) \geq \tau_i$

Return Q

Figure 1. Selecting predictors during training

EvalScript

Inputs: One script $s \in$ Repository R

Minimal predictor matches $\beta \in (0, \left|Q\right|]$

Constraint set $Q = \{ q_i : R \to \{\text{false}, \text{true}\} \}$

Outputs: Prediction of value $\in \{0,1\}$

Let nmatches = # of satisfied constraints in Q

If nmatches ≥ β then return 1 else return 0

Figure 2. Labeling a script as high- or low-value

Moving from CoScripter to other repositories has required moderate changes to the set of traits used to test the model. Many of the original 35 traits initially developed for CoScripter macros could also be computed in identical or analogous ways for Yahoo Pipes and UserScripts (Table 1). Other traits were very specific to CoScripter—for example, CoScripter supports a "mixed initiative" instruction, which pauses execution while the user performs an action manually, but Yahoo Pipes and UserScripts do not support a similar instruction. Also, as discussed below, we acquired data from these two repositories by programmatically reading their websites rather than through internal server logs (which IBM provided for CoScripter), and this limited our ability to compute some traits that were based on history or code authorship. On the other hand, the new repositories

Table 1. Traits computed for UserScripts and Yahoo Pipes scripts. Asterisks indicate traits that are new (not computed or having an analogue in the earlier CoScripter work). N/A indicates not applicable or not able to be computed with the available data. Section 6 describes the numeric information value of each trait.

	Name	Meaning	UserScripts	Yahoo Pipes
Code-based	comments	int: # of comment lines	0.27	N/A
	code_lines	int: total # of non-comment lines in script	0.28	0.39
	total_lines	int: total # of lines (code_lines + comments)	0.25	N/A
	distinct_lines	int: total # of distinct non-comment lines in script	0.25	N/A
	literals	int: # of literal strings hardcoded into script	0.29	0.30
	internal_vars	int: # of temporary variables declared in the code *	0.27	0.31
	internal_functs	int: # of functions or callable methods defined in the code *	0.05	0.10
	params	int: # of input values read by script from user	N/A	0.46
	input_sources	int: # of input sources read by script from servers *	N/A	0.17
	internal_flow	int: # of dataflow connections internally (Pipes wires) *	N/A	0.37
	loops	int: # of loops in script *	0.24	0.34
Annotation-based	test_title	bool: true if title contains the word "test"	0.02	0.00
	punct_title	bool: true if title contains punctuation other than periods	0.11	0.21
	desc_len	int: length of description that supplements the title *	0.23	0.26
	nonroman	pct: % of non-whitespace chars that aren't roman	0.22	0.12
	tags	int: # of tags *	N/A	0.28
URL-based	ip_urls	int: # of URLs in script that use numeric IP addresses	0.24	0.05
	inet_urls	int: # of URLs that reference intranet websites	0.24	0.03
	us_urls	int: # of US URLs in script	0.07	0.32
	nonus_urls	int: # of non-US URLs in script	0.08	0.01
	no_urls	bool: true if nonus_urls and us_urls are each 0	0.01	0.26
	distinct_hosts	int: # of distinct hostnames in script's URLs	0.05	0.26
	urldom_sim	real: similarity of URLs in this script to other scripts' URLs	0.24	0.35
History-based	author_id	int: identifies when user joined (a measure of experience)	0.00	0.21
	forum_posts	int: # of posts by the script author on the site's forum	0.04	N/A
	prev_created	int: # of scripts created by this script's author	0.16	0.23
	is_cloned	bool: true if script was created by cloning another script *	N/A	0.25

presented features such as loops that CoScripter lacked, leading to 8 new traits. The other 19 of the 27 traits in Table 1 were computed as for CoScripter.

3.2 Measures of value

The model attempts to predict a binary measure of script value. For example, we can use a binary measure that is based on whether a script is reused at least a certain number of times. The model uses binary rather than absolute measures of value because information cascades badly cloud the meaning of absolute measures of reuse. Specifically, once an information cascade puts a script high in search results, the amount of reuse rapidly accelerates. Thus, there is little meaningful difference between 1000 uses (for example) and 5000 uses. Consequently, rather than evaluating if the model can use the

traits above to predict absolute levels of value, we instead evaluate how well it can predict if scripts will exceed several binary thresholds of reuse.

To establish thresholds for converting absolute measures of value into binary measures, we downloaded the 200 newest UserScripts created on or before May 1, 2009, and 282 Yahoo Pipes created in Mar 2009. (Since Yahoo Pipes did not have a way to search by date range, we took the 1000 scripts returned by the "Browse Pipes" page, issued additional searches to find nearly 7000 additional scripts that used the 10 most popular modules, and finally filtered the results to obtain the 282 scripts created in March 2009.) These dates were selected because they were approximately 6 months prior to the download date, giving scripts a fair amount of time for users to try them out. The repositories provided the data needed to compute several indicators of script value:

- *UserScripts installs*: number of times that a script was downloaded since its creation
- *UserScripts lines*: maximum number of lines that the script had in common with a later-created script, as an indication of partial reuse; for example, if a script had 5 lines in common with one later script, and 7 in common with another script, then *lines* = 7
- *UserScripts reviews*: total number of ratings, textual reviews and discussions provoked by the script, as a measure of interestingness to users
- *Yahoo Pipes clones*: number of times the script was copied into a new pipe

To convert these into binary measures of value, we histogrammed absolute measures and chose thresholds at the 75th, 50th, and 25th percentiles where possible (Figure 3). For example, 25% of UserScripts had ≥ 300 installs (what we consider "diamonds"), so we set one threshold at that level. Conversely, only 25% of UserScripts failed to have ≥ 20 installs (providing a threshold distinguishing between scripts of *some* versus *very little* value). Due to the low numbers of UserScripts reviews and Yahoo Pipes clones, we could compute only two distinct thresholds for those two indicators of value. Overall, these 4 indicators yield 10 measures of value.

4. Assessing generalizability

To assess how well the model could handle the broad range of binary measures, we used the ten-fold validation typical of machine learning research. That is, we trained the model on 90% of data, tested it on the other 10%, then averaged results after repeating 10 times so each script could act as a test. We measure accuracy with False Positive (FP) and True Positive (TP) rates:

TP= # high-value scripts labeled as high-value / # high-value scripts
FP= # low-value scripts labeled as high-value / # low-value scripts

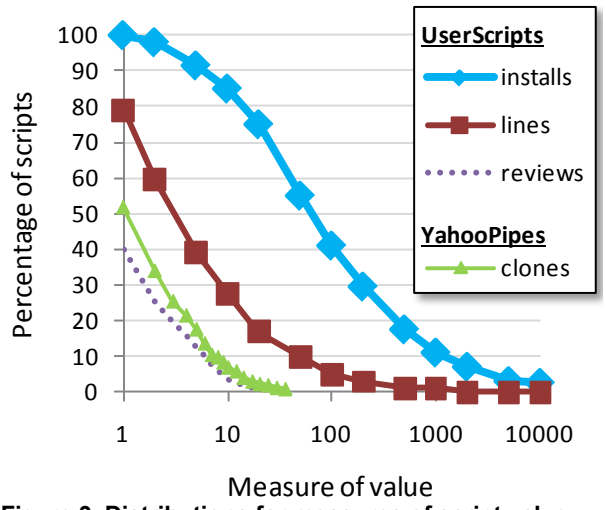

Figure 3. Distributions for measures of script value

TP is the same as the recall measure used in information retrieval. FP is similar in purpose to information retrieval's precision measure, but FP is often preferred over precision, since FP is more robust than precision to small changes in the experimental data [8].

Raising β makes the model more selective, reducing FP as well as TP. Conversely, lowering β identifies more scripts of interest and raises TP, but at the cost of also raising FP. Ideally, TP will rise faster than FP.

Testing the model on the 10 measures of value revealed that TP rose to between approximately 0.7 and 0.9 by the time that FP reached 0.4 (Figure 4). This is the same range of accuracy previously attained on CoScripter data [11], indicating that the model generalizes to various kinds of end-user code (Q1 from Section 1). Accuracy was little affected by the specific binary thresholds chosen, indicating that the model is just as good at finding the most valuable scripts as it is at finding the least valuable (Q2). Finally, the graphs in Figure 4 show little variation in accuracy even though they are based on different indications of value (installs, lines copied, reviews, and clones), indicating that the model generalizes fairly well to multiple indications of value (Q3).

5. Assessing model durability

Using the model to enhance repositories will require training on past data in order to make predictions about the value of new scripts that are created later. Training on one data set and testing on another ("data shifting") typically reduces a machine learning model's accuracy since, for some domains, the relationship between the output variable and the input variables can slowly change over time. For example, in our case, as a repository's user population becomes more experienced, certain traits (e.g., code comments) might be less necessary for users to be able to successfully reuse code.

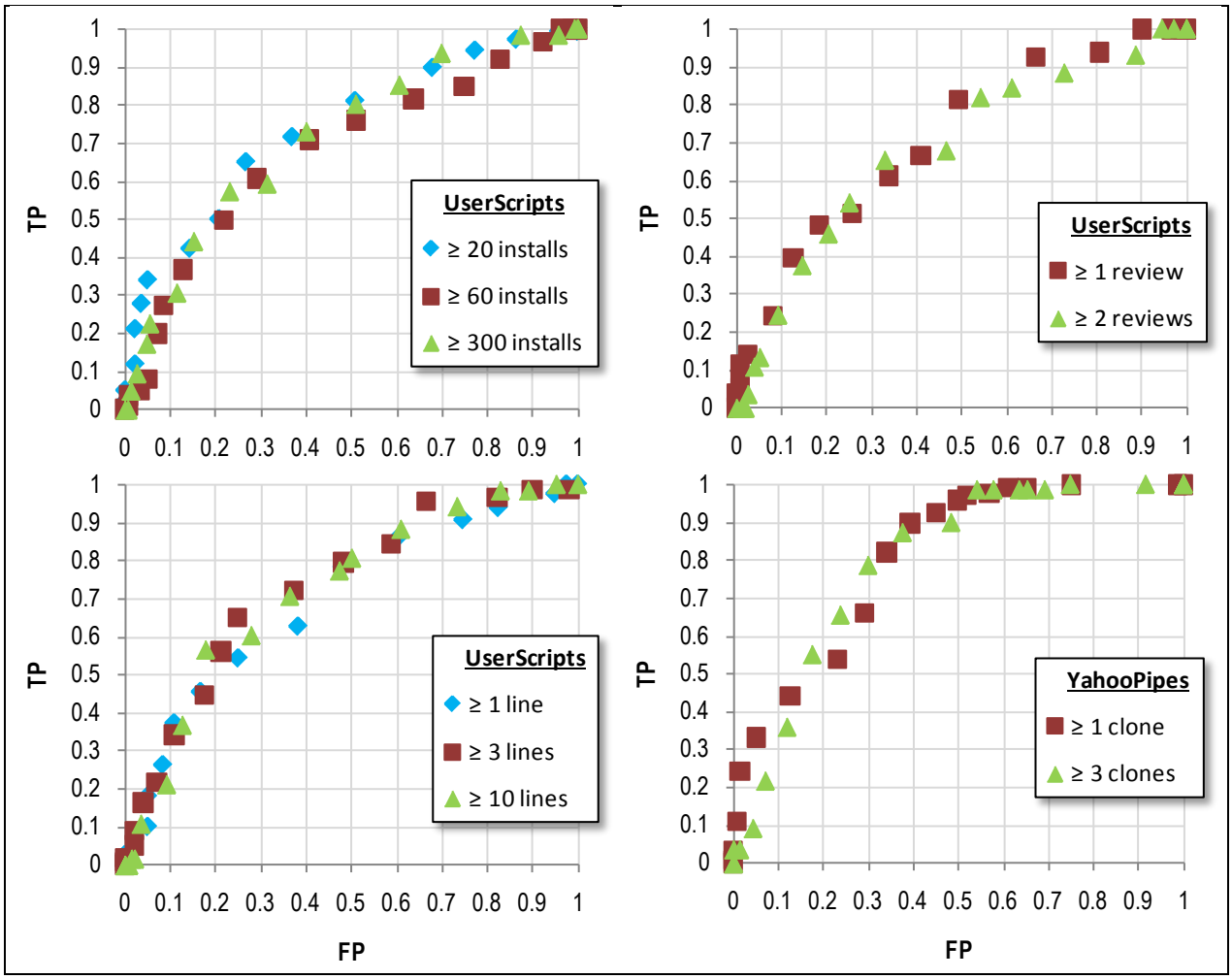

Figure 4. Model accuracy for 10 different measures of script value.

To test how well the model performs in a data shifted scenario, we repeated the experiment in Section 4 using the highest thresholds. However, this time, we trained the model on the Spring 2009 data already described and then tested it on 200 new UserScripts and 200 new Yahoo Pipes created in Fall 2009.

The model performed almost as well in the data shifted scenario as it did when training and testing on the same data set (Figure 5). There was no noticeable loss of accuracy for UserScripts ("US" in Figure 5), but the TP rate for Yahoo Pipes ("YP") did decrease by as much as 0.2. (Similar results were obtained for other measures, not shown in order to conserve space.)

The implication is that the relationship between code traits and value can indeed shift a bit over 6 months in an end-user repository (Q4 from Section 1), though the resulting drop in accuracy does not appear to be precipitous. One way to address this drop would be to retrain the model more frequently than every 6 months. For example, perhaps it could be retrained offline each week to make predictions for the next week.

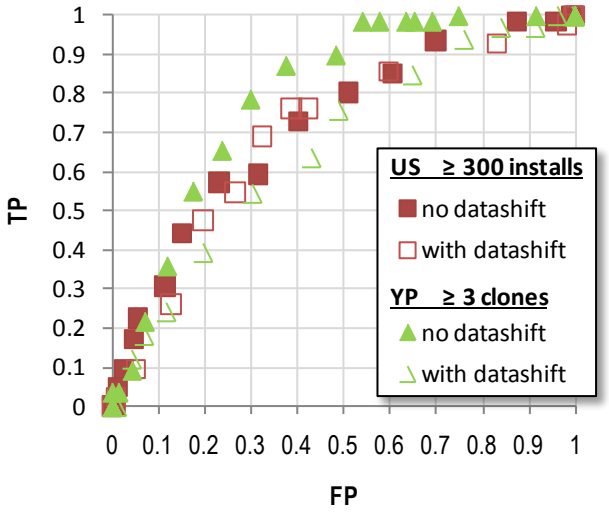

Figure 5. Little loss of accuracy when training on one dataset and testing on another (data shifting)

6. Incremental benefit of computing traits

Before the machine learning model can be trained, each trait must be computed for each training script. This involves programming that requires time and effort. Before making this investment, repository designers might want to know how much incremental benefit could result from computing each trait.

Repository designers might also face decisions whose choices could prevent computing certain traits. For example, a designer might consider letting users upload binary (compiled) programs, rather than source code or scripts. Before making this choice, it would be useful to know how much the absence of code-based traits would reduce the model's accuracy. As another example, the designer might consider a peer-to-peer architecture rather than a centralized server, but this could prevent logging the data needed for computing history-based traits. How much would this choice impact accuracy?

To answer such questions (each a form of Q5 from Section 1), we repeated the "search for diamonds" experiment of Section 4 using only subsets of traits (Figure 6). For instance, the "only C" configuration tested the model using only code-based traits, while the "all but C" configuration included all traits except those based on code. To support comparability across configurations, we report the TP value when β is set so FP is 0.4.

In addition to the configurations whose results are presented by Figure 6, we tested the model using each trait by itself. When using a single trait, the only valid setting for β is 1, making it impossible to tune β so FP=0.4. Consequently, for individual traits, we report the information value $\max p(R_m, a_i, \tau) - p(\overline{R}_m, a_i, \tau)$ (computed as shown in Figure 1). This value is precisely equivalent to the difference TP-FP that would result from making predictions using just that one trait by itself. Table 1 shows this numerical information value for each trait. (Although there are alternate possible measures of each trait's individual information value, this simple TP-FP measure proved most useful in prior work [11].)

Of the four categories of traits, we found that the code-based traits (followed by annotation-based traits) were most crucial for model accuracy. Comparing the "all" and "all but C" bars of Figure 6 shows that omitting code-based traits reduced TP by approximately 0.12. In contrast, omitting history, URL, or annotation traits dropped TP by 0.07 or less. Turning to the "only C" bars, the code-based traits alone did not perform as well for UserScripts as they did for Yahoo Pipes. Yet for Yahoo Pipes, code-based traits alone provided nearly the same accuracy as using all traits combined. History-based and URL-based traits were least useful.

Reviewing the results in Table 1, most code-based traits had a TP-FP of at least 0.25. This individual accuracy was nearly as high as the TP-FP =0.8-0.4=0.4 that the group as a whole attained for Yahoo Pipes ("only C" bar, Figure 6). This suggests that the code-based traits provided redundant information, and omitting a few might not reduce the "only C" group's total accuracy by very much. In addition, repository designers might consider omitting some history- or URL-based traits, since virtually none of these performed well for both repositories (Table 1), and since omitting each of them did not harm accuracy much overall (Figure 6).

Our results resemble those found when testing traits on CoScripter repository logs [11]. In that work, code-based traits such as counts of comments, code length, and parameters worked well. History-based traits fared better for CoScripter than for UserScripts and Yahoo Pipes, but many more history-based traits were possible for CoScripter because we had access to the repository's internal logs. Yet computing these history-based traits from internal logs required a great deal of programming to data-clean the logs. In contrast, the other three categories of traits can be computed with much less effort.

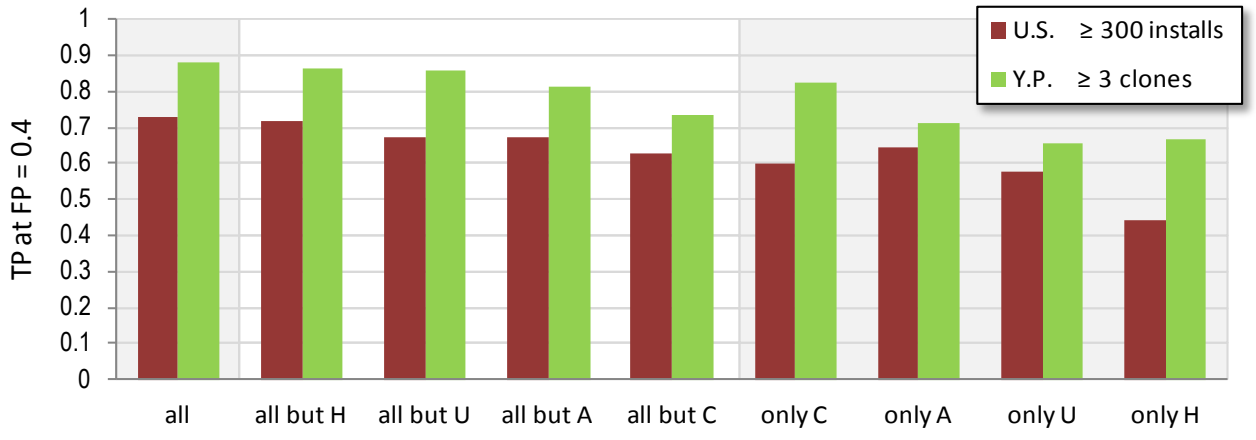

Figure 6. Model accuracy using subsets of traits (History, URL, Annotations, Code), revealing traits' relative benefit

Overall, we conclude that the benefit-to-effort ratio of history-based traits is lowest while that of code-based traits is highest.

7. Conclusions and future work

We have developed an automated machine learning approach for categorizing end-user code based on value. Using the model's accuracy on CoScripter macros in a prior experiment as a baseline, our latest experiment revealed similar accuracy when using the model for Yahoo Pipes and UserScripts, with a range of different value indicators and value thresholds. These results indicate that the model generalizes to other kinds of end-user code, besides web macros (Q1 from Section 1), that it can predict different levels of value (Q2), and that it can predict value based on different value indicators (Q3). Accuracy was only slightly harmed by predicting future reuse events based on past rather than contemporary events (Q4). Overall, code-based traits provided the best incremental benefit for relatively little effort (Q5).

Overall, these results indicate that the model can serve as an effective basis for future repository enhancements that we will develop in future work.

First, we will extend search engines to incorporate predictions of code value. For example, when a search engine needs to break ties between scripts that equally match a user's keyword query, these predictions could be used to give prominence to scripts that are most likely to provide high value.

Second, we note that users can control the code-based traits, which predicted value so well. For example, users could replace hardcoded literals with parameters and embed more comments. But since end users are often not trained in programming techniques, they might not recognize opportunities to make these small code edits. Thus, we will experiment with developing automated design critics that give users suggestions about how to improve code's value. These critics might even offer to automate certain changes. We expect that users will be motivated to take such advice if doing so is easy and if they find that doing so frequently facilitates later code reuse.

Third, we will provide system administrators with enhanced repository-management tools. For example, we could help them review high-value scripts to find some that could be promoted on the home page, as well as to find low-value code that could be purged.

Finally, having made progress on identifying valuable end-user code in repositories, we will now also start to focus on helping users to benefit from the *less* valuable code in repositories. These repositories contain thousands of scripts that do not run correctly, that are over-specialized for the needs of a particular person, or that are "uncommented spaghetti code" [4]. Professional programmers encounter similar problems all of the time, but they can rely on training and experience to diagnose, repair and extend code so they can incorporate it into new programs. By studying how professionals perform these tasks, and then embedding that expertise into automated assistants, we hope to help end-user programmers more fully benefit from online repositories.

8. References

[1] S. Bikhchandani, D. Hirshleifer, and I. Welch. A Theory of fads, fashion, custom, and cultural change as informational cascades, *J. Political Economy,* 100(5), 1992, 992-1026.

[2] T. Brooks. Watch this: Greasemonkey the web, *Information Research*, 10(4), 2005, http://InformationR.net/ir/10-4/TB0507.html

[3] W. Frakes and C. Fox. Sixteen questions about software reuse, *Comm. ACM*, 38(6), 1995, 75-87.

[4] N. Gulley. Improving the quality of contributed software and the MATLAB File Exchange, *2nd Workshop on End User Software Engineering*, 2006, 8-9.

[5] Y. Kim and E. Stohr. Software reuse: Survey and research directions, *J. Management Information Sys.*, 14(4), 1998, 113-147.

[6] M. Kramer. Self-selection bias in reputation systems, *Intl. Federation Information Processing*, 2007, 255-268.

[7] G. Leshed, et al. CoScripter: Automating & sharing how-to knowledge in the enterprise, *SIGCHI Conf. on Human Factors in Computing Sys.*, 2008, 1719-1728.

[8] 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(9), 2007, 637-640.

[9] A. Monroy-Hernández and M. Resnick. Empowering kids to create and share programmable media, *Interactions*, 15(2), 2008, 50-53.

[10] M. Rosson, J. Ballin, and H. Nash. Everyday programming: Challenges and opportunities for informal web development, *Symp. Visual Languages and Human-Centric Computing*, 2004, 123-130.

[11] C. Scaffidi, et al. Predicting reuse of end-user web macro scripts, *Symp. Visual Languages and Human-Centric Computing*, 2009, 93-100.

[12] J. Segal. *Professional end user developers and software development knowledge*, Tech. Rpt. 2004/25, Dept. of Computing, The Open University, Milton Keynes, 2004.

[13] G. Smith. *Folksonomy: social classification*, 2004, http://atomiq.org/archives/2004/08/folksonomy_social_classification.html

[14] G. Stahl, T. Sumner, and A. Repenning. Internet repositories for collaborative learning: Supporting both students and teachers, *Proc. Computer Support for Collaborative Learning*, 1995, 321-328.

[15] R. Walpole, M. Burnett. Supporting reuse of evolving visual code, *Symp. Visual Languages*, 1997, 68-75.

[16] Yahoo Pipes Overview, http://pipes.yahoo.com/pipes/docs?doc=overview