Digging for diamonds: Identifying valuable web automation programs in repositories

Jarrod Jackson, Christopher Scaffidi
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR, USA
jacksoja@engr.orst.edu, cscaffid@engr.orst.edu

Kathryn T. Stolee
Department of Computer Science & Engineering
University of Nebraska
Lincoln, NE, USA
kstolee@cse.unl.edu

Abstract—**Web automation programs offer a means for users to enhance the usability of the web. These programs can be published on a wiki or other repository, thereby making them available for use by other users. However, in addition to programs of broad usefulness to the community at large, these repositories also contain many programs that are unreliable or highly specialized to the needs of very small sub-communities. These less valuable programs clutter the repository and make it difficult to find the valuable web automation programs. In this paper, we evaluate a machine learning model distinguish between high-value and low-value web automation programs. We find that the model performs well for a wide range of different languages, purposes and configurations, indicating that the model could serve as an effective basis for future repository enhancements.**

Keywords- Web automation; end-user programming; repositories

I. INTRODUCTION

The traditional view of the web is one where people manually click on links and fill in forms, but recent years have seen a proliferation of tools enabling people to create programs that *automate user interactions with web sites*. An example of a web automation program is the "mashup," which is a program that extracts, combines and visualizes data from multiple sites, thereby providing a synthesized view to support decision-making. For example, a user might construct a mashup that synthesizes news from ten websites to produce a unified feed of articles about Microsoft Corporation. An example of a tool to facilitate mashup creation is the Yahoo! Pipes programming environment, which allows users to create mashups that can retrieve, transform, and/or combine web data sources to produce a new feed [23]. For example, a Yahoo! Pipes mashup might plot news articles about Microsoft on a map, based on dates or city names mentioned in the stories.

By automating tedious, error-prone interactions with web sites, web automation programs offer a means for users to essentially enhance the usability of the web. Moreover, programs can be published on a wiki or other repository, thereby making them available for use by other users. For example, such a repository enables one user to run a Microsoft article mashup created by another user. Yet another user could copy the mashup and use it as a basis for a new program, such as a mashup that combines the original ten news sources with three additional sources to produce a timeline of articles about Samsung Corporation rather than Microsoft. Because they enable users to improve the web's usability in a shareable way, web automation programs are starting to be used in business [7], science [6], education [11], disaster response [12], public health [2], national defense [4], and digital government [24].

With the rapid rise of interest in these programs, however, a scalability problem has emerged. A typical repository might contain thousands of these programs, but as few as 10% of these might be of value to the community at large [17]. The other 90% include programs that are unreliable, as well as those that are so highly personalized to particular users and are so hard to read or modify that it is inconvenient for others to customize the programs to meet their own needs. The proliferation of less valuable programs makes it difficult to find the reliable programs that would be valuable to the community.

Today's repository search engines are of little use for solving this problem, since they generally only match and sort based on keywords, with little consideration for whether users might deem each program to be of any value. For example, if a user wanted an aggregated newsfeed of articles about Microsoft, searching for "Microsoft news" would return 114 hits on the Yahoo! Pipes repository right now; approximately 10% of these are working feeds that synthesize news sources to produce a unified feed of news articles about Microsoft Corporation. The other 90% either are broken, retrieve only a few articles, or only draw from a single website. Some of them seem at first to be relevant and functional, but further investigation reveals that they do not work as expected. These less useful programs clutter search results and make it difficult to find desired functionality.

The image of "digging for diamonds" metaphorically describes our goal: helping community members to identify valuable programs, so that they can more *consistently* benefit from a repository. As a step toward achieving this goal, we evaluate a machine learning model that can identify end-user programs that are likely to be valuable to the community. This model could ultimately be used to enhance repository search engines. For example, an enhanced search engine might sort results for "Microsoft news" based on a combination of keywords *and* our model's assessment of programs' value, rather than just based on keywords alone.

In prior work, we prototyped our model for use on one particular repository of web automation programs, with four specific measures of the programs' value [17]. We evaluated the prototype model for that repository and found its accuracy to be comparable to that of more complex machine learning models, including those used in other areas of software engineering.

The new contributions of the current paper are to show:

1. The model's accuracy is comparable for other kinds of web automation programs, as well. This implies that the model could be used not just for our original repository, but also for recommending web automation programs in general.

2. The model's accuracy is comparable for a range of different measures of programs' apparent "value." In particular, our model can be used to classify programs based on whether they are likely to be downloaded often, or based on whether they are likely to be copied often, or based on whether they are likely to provoke substantial interest by people (as reflected in quantities of online comments about the programs).

3. The model's accuracy is comparable for identifying programs that meet different thresholds of value. Prior work tested how well the model could predict whether programs would have *any* value. Even if a program has some value (e.g., is downloaded once), it still might not be especially valuable. Our current work shows that the model is also good at finding "diamonds" in the repository that have *high* apparent value but have not yet been "discovered."

4. The model's accuracy is only slightly reduced by training it on past data to predict future events. This means that a search engine could be configured based on data from the past few months and used to search through users' programs in the future. Without the ability to predict the future based on the past, such recommendations would not have been feasible.

5. Most of the model's accuracy derives from one particular portion of the model, which incorporates information extracted from source code. This finding will help to reduce the amount of effort required for using our model in practice. Specifically, if a company wanted to use our model to improve a repository's search engine, then the company could start by integrating just the *portion* of our model that has performed so well, thereby obtaining most of the overall model's accuracy while expending only a fraction of the effort that would be required to integrate the overall model. Later, if desired, the company could expend the effort required to integrate the rest of the model.

Our paper is organized as follows. In Section II, we summarize related work aimed at helping end users to find and reuse programs from online repositories. In Section III, we describe our model and our datasets for testing it. In Section IV, we present the results of our analysis. We conclude in Sections V and VI by discussing limitations, conclusions and implications.

II. Related Work

Some of the newest and fastest-growing end-user code repositories contain thousands of web automation programs. 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 [10]. Another example is Yahoo! Pipes, which are programs that retrieve lists of records via the RSS protocol from web sites, then combine and filter those records [23]. A third example is the UserScripts repository of "GreaseMonkey" scripts, whose purpose is to alter and enhance the HTML of web sites (e.g., by removing advertisements) [3]. For each of these systems, Figure 1 shows an example of a program that has not yet been "discovered" by users in the corresponding repository.

- go to "http://vgt.familygivingtree.org/fgt"
- click the "Sponsor this Child" button
- click the "chekout.gif" button
- enter your "Full Name" into the "Name on Card" textbox
- enter your "credit card number" into the "Card Number" textbox
- select your "credit card expiration month" from the "January" listbox
- select your "credit card expiration year" from the "2010" listbox

Figure 1a. Part of a CoScripter macro that no users have run, other than the author. CoScripter directly executes these English-like instructions. This macro signs the user up to sponsor a Christmas gift for a poor child (using variables such as "Full name" that are defined in each user's personal configuration file).

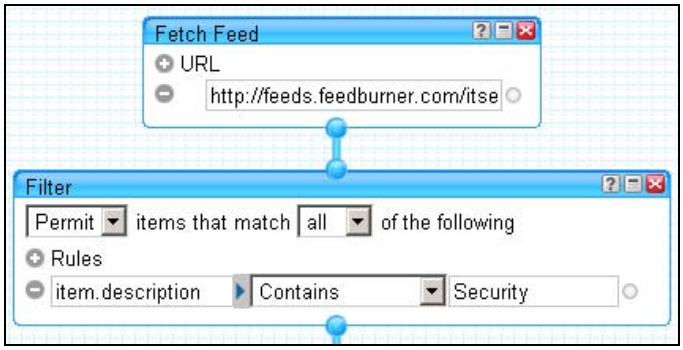

Figure 1b. Part of a Yahoo! Pipes program that no users have yet discovered and copied/cloned. The Pipes engine runs these visual instructions directly. This program retrieves articles about information technology security.

```
var link = $(element).find("a.tableViewCellTitleLink");
var t = {
  text: link.text(),
  url: link.attr("href")
};
var tweetLink = $('<a href="http://twitter.com/share?'+
if(isHelvetapaper()){
  $(element).find(".cornerControls").append(tweetLink.a
}
else {
```

Figure 1c. Part of a GreaseMonkey script that nobody has yet downloaded and installed from UserScripts.org. Such scripts are written in JavaScript. This script modifies the InstaPaper.com website by adding a link to send a "tweet" about an article through Twitter.

Search engines are the main approach for digging through the thousands of programs that end users have uploaded to these and other repositories [5][15][20][22]. Programs can also be tagged using a folksonomy [19], then browsed by category. These approaches identify programs based on content (reflected in keywords, categories or tags) rather than value, so search results contain a mixture of programs with varying levels of value.

To help members of a community focus on programs that they might consider valuable, search engines typically sort results by numbers of downloads (e.g., [5][10]) or based on ratings (e.g., [3][5][10]). Yet download counts and ratings are not available until after a program has been downloaded and/or rated, and few are ever rated by these repositories' users. For example, right now, only 7% of UserScripts programs have been rated. Consequently, the repository's popularity-based sorting features essentially place the vast majority of programs in a blanket coarse-grained category of "unpopular" (or perhaps "not yet popular"), thus providing virtually no help in distinguishing highly valuable programs from less valuable programs. Moreover, as with most ratings by end users of online content [9], ratings by end-user programs are skewed high. For example, right now, only 22% of rated UserScripts programs are rated neutral or lower. This strong bias suggests that even when ratings are available, they provide an incomplete view of programs' value.

In contrast, our model uses information available at the time of a program's creation, such as data computed through static analysis of programs, to identify "diamonds in the rough" that have perhaps not yet been rated but that are likely to be highly valued when they are discovered. Since almost all ratings are positive, we evaluate in this paper whether the model is useful for predicting which programs will accrue large numbers of ratings. In addition, we evaluate whether the model can predict other indicators of value (e.g., whether a piece of code will be downloaded often). These predictions, in turn, could be used to refine search results in the vast mass of cases where current approaches cannot be applied. This might enable users to more easily find the code that they need from search results.

Aside from the users, an online community's other human element is the system administrator, who diligently tries to cultivate a body of highly valuable programs. This typically includes reviewing programs and promoting the best on the home page or a blog (e.g., [5][10][15][23]). However, this cultivation is limited, since system administrators do not have time to review every one of the hundreds of programs uploaded every week. Ideally, the repository should be able to identify the programs that are probably most valuable and bring them to the administrator's attention for consideration as worthy of promotion. Conversely, the repository might also identify the programs 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 and other purposes aimed at helping system administrators.

III. Approach

Motivated by the limitations in existing repositories, our automated machine learning approach categorizes end-user web automation programs based on likelihood of high value, rather than content [17]. Evaluating the effectiveness of this approach requires several steps: characterizing the traits of programs to be used for predictions, identifying target thresholds for several indications of value, and finally testing the model. Here, we describe each step in detail.

A. Program Traits

Our model for identifying apparently valuable programs 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 programs, while the output categories are binary measures of program value. For example, given the goal of identifying programs that will be copied at least 1000 times, the training process will infer a function for identifying programs that likely will meet this threshold. Later, the inferred function can be applied to other programs to make predictions about their apparent value.

Because our approach uses machine learning to make predictions, we need to identify features of programs, which we call traits, that can be used in the machine learning algorithm. Moving from CoScripter to other repositories has required moderate changes to the set of traits used to drive 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 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.

B. Measures of Value

The model attempts to predict a binary measure of program value. Note that it is impossible to know from a repository log how *truly* valuable a program is. Instead, the problem with designing a repository is to use various *measures* of value, or *indicators* of value, as *proxies* for the underlying value. Each of these measures may offer a different perspective on what might constitute "value" to different users (whether value for wholesale reuse, value for discussion, value for learning, and so forth). Therefore, by testing our model with a *breadth* of different measures, we can ascertain how well it works for many different perspectives of what constitutes value.

Table 1. Traits computed for UserScripts and Yahoo! Pipes programs. 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 IV.C explores and explains 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 program	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 program	0.25	N/A
	literals	int: # of literal strings hardcoded into program	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 program from user	N/A	0.46
	input_sources	int: # of input sources read by program from servers *	N/A	0.17
	internal_flow	int: # of dataflow connections internally (Pipes wires) *	N/A	0.37
	loops	int: # of loops in program *	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 program 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 program	0.07	0.32
	nonus_urls	int: # of non-US URLs in program	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 program's URLs	0.05	0.26
	urldom_sim	real: similarity of URLs in this program to other programs' 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 program author on the site's forum	0.04	N/A
	prev_created	int: # of programs created by this program's author	0.16	0.23
	is_cloned	bool: true if program was created by cloning another program *	N/A	0.25

The model attempts to predict binary rather than absolute measures of value. For example, we can assess value using a binary measure based on whether a program is run/installed at least a certain number of times, or whether a program is reviewed at least a certain number of times. The reason which motivated this in our prior work was information cascades [1][17]: once user attention puts a program anywhere close to the top of 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 developing a model to predict a precise absolute level of apparent value, we focus on providing a model that can predict if programs exceed at least a certain threshold or, in particular, if programs are at least in a certain

quartile. For example, we ultimately want to use the model to predict whether a program is a "diamond" that will be in the top 25%; one use case for this prediction would be to identify promising programs that a system administrator should review to see if they deserve to be mentioned in a "Pick of the Week" blog. On the other hand, identifying programs in the lower quartile would make it possible to push those down in search results and perhaps even to recommend them for removal (or archiving) from the website.

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 programs returned by the "Browse Pipes" page, then issued additional searches to find nearly 7000 additional programs that used the 10 most popular modules. Finally, we filtered the results to obtain 282 programs created in March 2009.) These dates were selected because they were approximately 6 months prior to the download date, giving programs a fair amount of time for users to try them out.

The repositories provided us with the data needed to compute four indicators of program value:

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

To convert these into binary measures of value, we histogrammed absolute values and chose thresholds at the 75th, 50th, and 25th percentiles where possible (Figure 2). For example, 25% of UserScripts had ≥ 300 installs ("diamonds"), so we set one threshold at that level. Conversely, only 25% of UserScripts failed to have ≥ 20 installs (providing a threshold distinguishing between programs of *some* versus *very little* apparent 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 yielded 10 thresholds.

C. Training and Using the Model

During training, traits are used to automatically generate constraints that are generally satisfied by apparently valuable programs but rarely by the others. For example, number_of_comments ≥ 3 might be one inferred constraint, and another might be number_of_variables ≤ 2.

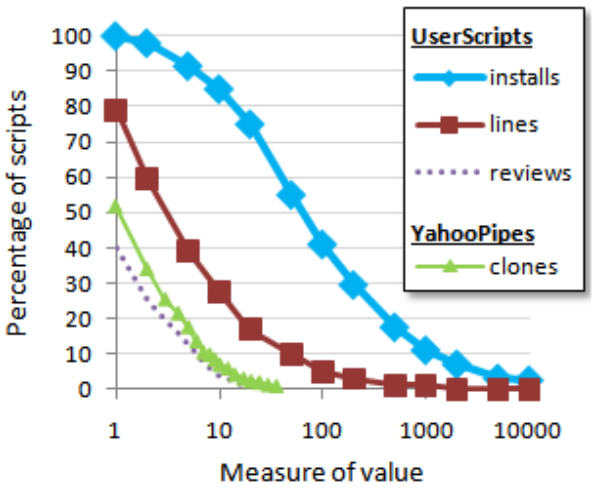

Figure 2. Distributions for measures of script value

The training algorithm proceeds in three stages (Figure 3). First, it determines if higher levels of the trait are positively or negatively related to higher levels of value; if the two are negatively related, then the trait is adjusted through multiplication by -1. Second, for each adjusted trait, the algorithm selects the constraint threshold that maximizes the *difference* between the proportion of high-value programs that meet the threshold, and the proportion of low-value programs that meet the threshold. Finally, the algorithm collects and returns the generated constraints.

TrainModel

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

Program traits $T = \{t_i : R \rightarrow [0, \infty)\}$

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

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

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

Let the set of low-value programs $\bar{R}_m = \{r \in R' : m(r) = 0\}$

Define proportion $p(S, t, \tau) = \left|\{s \in S : t(s) \geq \tau\}\right| / \left|S\right|$

Initialize Q to an empty set of constraints

For each $t_i \in T$,

Let $\mu_i = \sum_{r \in R'} t_i(r) / \left|R'\right|$

Let adjusted trait

$a_i(r) = t_i(r)$ if $p(R_m, t_i, \mu_i) \geq p(\bar{R}_m, t_i, \mu_i)$

or $a_i(r) = -c_i(r)$ otherwise

Compute threshold (through exhaustive search)

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

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

Return Q

Figure 3. Selecting predictors during training

(Our prototype of the training algorithm previously included a step to discard traits where this difference in proportions was below a certain level, but experiments on the prototype showed that this extra step did not affect model accuracy much on average [17], so we now omit this step.)

After "loading" the model with constraints in training, it can later be used to classify other programs as probably high- or low-value (Figure 4). Specifically, a program is classified as "valuable" if it matches at least β constraints.

We chose to create a new model, rather than simply use an existing algorithm such as a Bayesian classifier, because we wanted recommendations to be *automatically explainable*. Typical machine learning algorithms combine predictors into complex structures, such as a decision tree or a graph [14]. Explaining recommendations generated from complex models requires complex explanations [8], which can be unintuitive to users [21]. Thus, we sought to combine predictors in a simple way that might yield concise explanations, which led us to our model based on a simple count of the number of predictors matched by a program.

Prior work showed that our model was empirically just as accurate as more complex models on CoScripter data [17]. Moreover, the accuracy of our model 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 [13][16]. 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. Our current goal is to determine whether the model is again equally accurate for a broader range of web application programs and for a much broader range of configurations.

IV. Evaluation and Results

Ten-fold cross-validation is the usual method used when training and evaluating a machine-learned model on data. We used this approach to evaluate the accuracy of our model, configurations of our model with specific data, and executions of our model using just a single trait at a time.

EvalProgram
Inputs: One program $r \in$ Repository R
Minimal predictor matches $\beta \in (0, |Q|]$
Constraint set $Q = \{ q_i : R \rightarrow \{false, true\} \}$
Outputs: Prediction of value $\in \{0, 1\}$
Let nmatches = # of satisfied constraints in Q
If nmatches ≥ β then return 1 else return 0

Figure 4. Labeling a program as high- or low-value

A. Evaluating Model Accuracy

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 program could act as a test. We measure accuracy with False Positive (FP) and True Positive (TP) rates:

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

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 data [13]. Raising β makes the model more selective, reducing FP as well as TP. Conversely, lowering β identifies more programs 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 5). This is the same range of accuracy previously attained on CoScripter data [17], indicating that the model works well on various kinds of web automation programs. Accuracy was little affected by the specific binary thresholds chosen, indicating that the model is just as good at finding the most valuable programs as it is at finding the least valuable. Finally, the graphs in Figure 5 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 works nearly equally well on multiple indications of value.

B. Using the Past to Predict the Future

Using the model to enhance repositories will require training on past data in order to make predictions about the apparent value of new programs 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 user population becomes more experienced, certain traits (e.g., code comments) might be less necessary for users to be able to successfully reuse programs.

To test how well the model performs in a data shifted scenario, we repeated the experiment above 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 Fall 2009.

The model performed almost as well in the data shifted scenario as when training and testing on the same data set (Figure 6). There was no noticeable loss of accuracy for UserScripts ("US" in Figure 6), but the TP for Yahoo! Pipes ("YP") did go down by as much as 0.2. (Similar results were obtained for other measures, not shown to conserve space.)

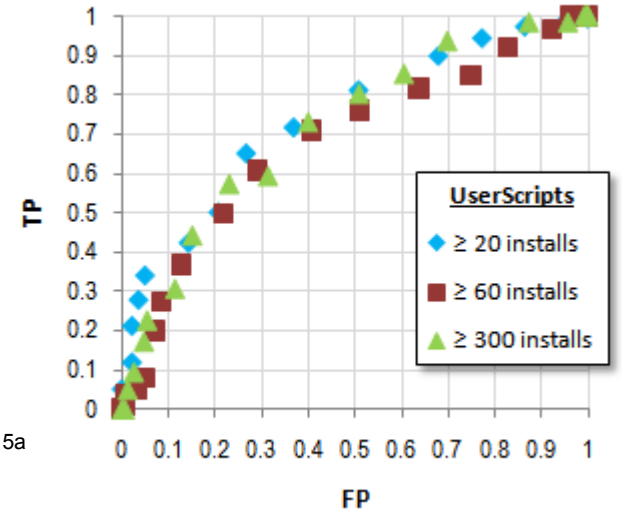

Fig 5a

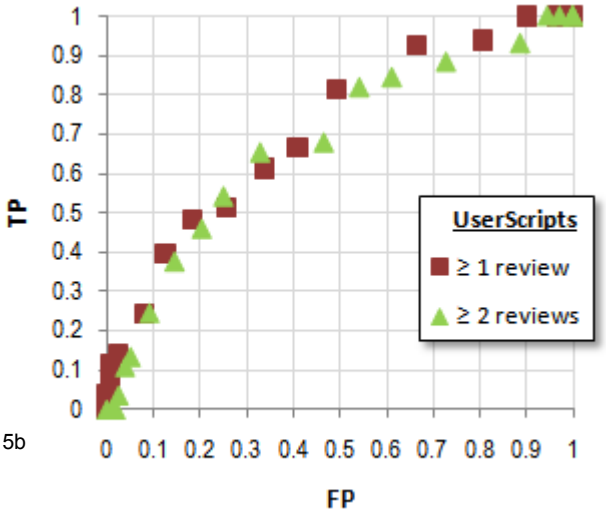

Fig 5b

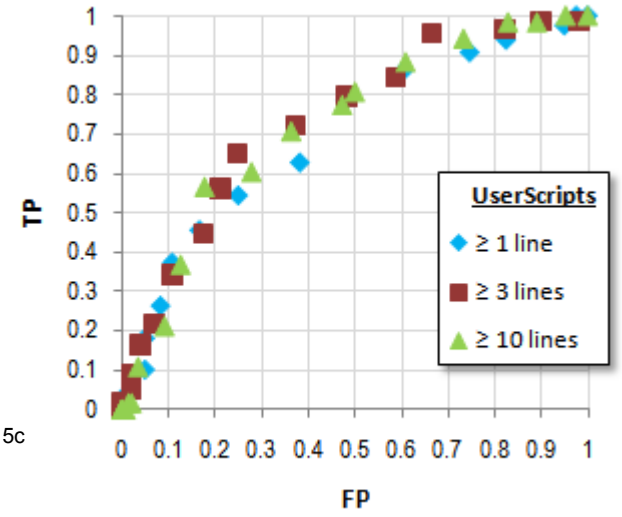

Fig 5c

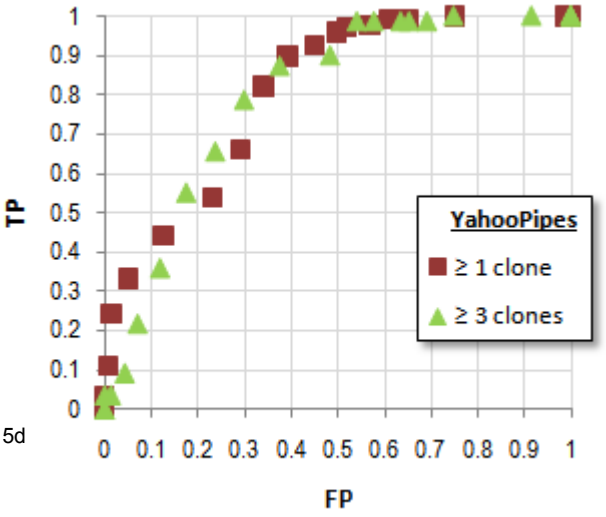

Fig 5d

Figure 5a-d (above). Accuracy for 10 different value thresholds. Figure 5e (bottom left) CoScripter data for comparison.

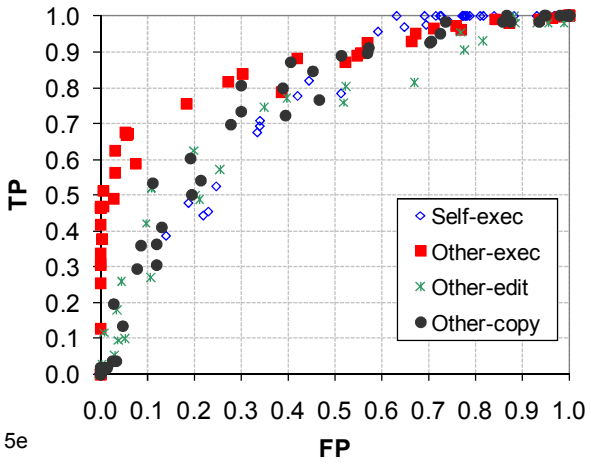

Fig 5e

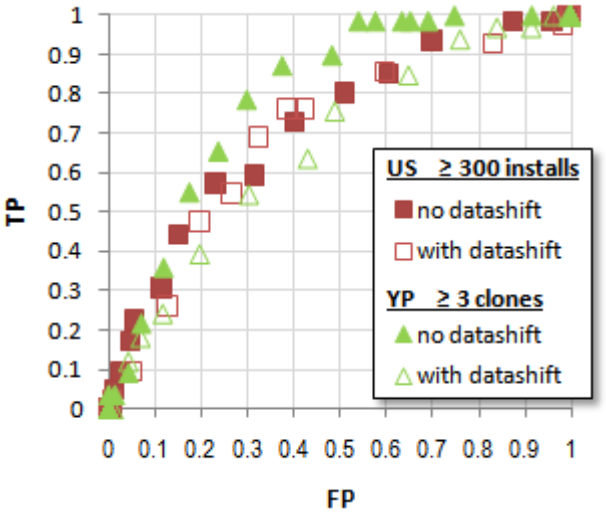

Figure 6. Little loss of accuracy when training on one dataset and testing on another (data shifting)

The implication is that the relationship between program traits and value can indeed shift a bit over 6 months, 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.

C. *Assessing Incremental Value of Computing Traits*

Before the machine learning model can be trained, each trait must be computed for each training program. 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 a peer-to-peer architecture rather than a centralized server, but this could prevent logging the data needed for computing history-based traits. Before making this choice, it would be useful to know how much the absence of history-based traits would reduce the model's accuracy.

To answer such questions, we repeated the "search for diamonds" experiment above using only subsets of traits (Figure 7). 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 7, 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 $\max p(R_m, a_i, \tau) - p(\bar{R}_m, a_i, \tau)$ (computed as shown in Figure 3). 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 [17].)

Of the four categories, 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 7 shows that omitting code-based traits reduced TP only 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, code-based traits alone did not perform as well for UserScripts as 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 7). 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 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 7).

Our results resemble those found when testing traits on CoScripter repository logs [17]. In that work, code-based traits such as counts of comments, code length, and parameters performed best. 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.

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

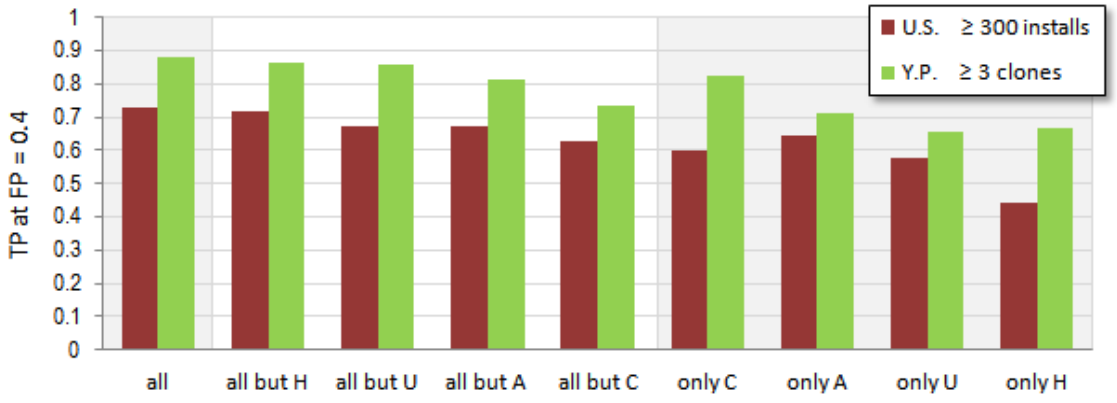

Figure 7. Model accuracy using subsets of traits (History, URL, Annotations, Code), revealing traits' relative benefit

V. Limitations and Threats to Validity

We have described a series of measurements evaluating the accuracy of a machine learning model that uses traits of web macros to predict measures of program value. We found overall that the model performs approximately as well as it did on two new repositories of web automation programs as it did on the repository where we initially prototyped the model. While these results are encouraging, our process embodies a number of limitations and threats to validity.

A. Limitations

The primary limitation of this approach is that it only predicts measures of program value, not program value directly. More precisely, it attempts to identify programs that will eventually be reused or reviewed, etc, given enough time (several months in our experiment) and interest by other users, rather than the programs that that would be considered valuable if somebody happened to have a need for them. The model's ability to predict program value could be assessed in future work by asking users directly about the value of programs, yielding a "gold standard" that we could then use to further evaluate the model.

A second limitation is that the model makes binary predictions rather than predictions of absolute levels of value. Given that the binary model has performed well in this work, an investment of further effort is now justified in order to refine the model to make absolute predictions. If direct assessments of program value become available, for example after interviewing users, these assessments might be represented on a ratio rather than binary scale, so predicting value might inherently imply refining our model to make numeric rather than binary predictions.

A third limitation is related to the fact that the model only relies on traits that can be computed at the moment when a program is created. While this implies that the model can be used on programs as soon as they are created, thereby meeting one of our design goals, it also implies that the model does not take advantage of any information that might become available after the program is created. Future work could aim to identify and incorporate such traits (e.g., by integrating our model into collaborative filtering algorithms). The result would ideally be a model that can be applied in a basic form when a program is created, with increasing accuracy as more data becomes available.

B. Threats to Validity

Internal validity refers to the question of whether a causal relationship exists as claimed. In our context, the key question is whether the model's predictions of measures of value are accurate because the traits actually reveal information about why reuse or other value-indicative events occur. The alternative possibility is that the traits really offer little or no information about value, but that the model's accuracy instead resulted from statistical effects (such as random chance or from overfitting during machine learning). However, given that these traits have now generally performed well on repositories belonging to multiple communities, it would be surprising if statistical fluctuations were entirely the cause. While these considerations suggest that the model is a valid method for predicting value, we do not argue that our results show that *modifying* a program's traits in accordance with this model will *cause* higher reuse. Put concretely, for example, we have not shown that a user can make a program more likely to be reused by adding comments to it. Models (and theories, more broadly) can be useful for explaining phenomena, predicting phenomena, and/or designing things that affect those phenomena [18]. We have essentially shown that our model is useful for predicting the phenomena of high and low measures of program value; however, we have not yet shown that it is useful for guiding the design of programs in a way that raises or lowers value.

External validity, or generalizability, is concerned with the question of whether similar results would be obtained in other similar experiments. We have found that our model produces predictions that are roughly equal in accuracy across a range of different web automation programs and ten different measures/thresholds for assessing program value. This indicates broad and reliable generalizability across web application programs.

VI. Conclusions and Future Work

We have evaluated the accuracy of an automated machine learning approach for categorizing web automation programs 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 works approximately as well for other kinds of web automation code and other indications of value, that it can predict at different levels of apparent value, and that it can predict value based on different value indicators. Accuracy was only slightly harmed by predicting future reuse events based on past rather than contemporary events. Overall, static code-based traits provided the best incremental benefit for relatively little effort. Consequently, repositories that do not yet have a community focus could benefit from our model, as they do not need ranking or commenting systems to benefit from this work, since they can identify valuable code from the artifacts alone.

A. Future Applications of Our Model

As a whole, these results indicate that the model can serve as an effective basis for future empirical studies (described in the preceding section) as well as for future repository enhancements that we will develop to help online communities obtain more value from shared code.

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

work could integrate our model into collaborative filtering algorithms, thereby yielding a recommendation engine that can be applied in a basic form when a program is created, with increasing accuracy as more data becomes available.

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, they might not recognize opportunities to make these edits. Thus, we will experiment with developing automated design critics that explain why the code might not be of optimal value to the community, and then give users suggestions about how to improve code's value. The relatively simple structure of our model, compared to alternate models that we evaluated in prior work [17], will make it relatively straightforward to generate these explanations. 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, if they find that doing so increases the value of code to themselves, and if they consider it worthwhile to provide value to the community.

Third, we will offer enhanced repository-management tools to system administrators. For example, we can provide a repository "dashboard" to give system administrators a list of likely-high-value programs (that might deserve to be advertised on the front page) as well as a list of likely-low-value programs (that might deserve to be archived from the website).

B. Moving from Finding Code to Adapting Code

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 programs that do not run correctly, that are over-specialized for the needs of a particular person. These problems are not unique to web automation programs but rather are a broader problem with code written by end users. Even code written by relatively sophisticated end users, such as engineers and scientists, are often "uncommented spaghetti code" [5]. The reason is that unlike professional programmers, end users write code to meet their needs of the moment, but that code therefore might not meet future needs or community needs.

Professional programmers encounter similar problems when they reuse code, 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 code provided by online communities.

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] M. Boulos, *et al*. Web GIS in practice VI: A demo playlist of geo-mashups for public health neogeographers. *International Journal of Health Geographics*, *7*(1), 2008, 38-53.

[3] T. Brooks. Watch this: Greasemonkey the web, *Information Research*, 10(4), 2005.

[4] S. Gibson. Mashups give defense department strategic edge. *EWeek.com*, April 2009, http://www.eweek.com/c/a/Web-Services-Web-20-and-SOA/Mashups-Give-Defense-Department-Strategic-Edge-391656/

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

[6] T. Hey, S. Tansley, and K. Tolle (eds.) *The Fourth Paradigm: Data-intensive Scientific Discovery*, Microsoft Research, 2009.

[7] V. Hoyer, *et al*. Enterprise mashups: Design principles towards the long tail of user needs. *IEEE International Conference on Services Computing,* 2008, 601-602.

[8] U. Johansson, L. Niklasson, and R. Konig. Accuracy vs. Comprehensibility in Data Mining Models. *Proceedings of the 7th International Conference on Information Fusion*, 2004, 295-300.

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

[10] G. Leshed, E. Haber, T. Matthews, and T. Lau. CoScripter: Automating & sharing how-to knowledge in the enterprise, *SIGCHI Conf. on Human Factors in Computing Sys.*, 2008, 1719-1728.

[11] M. Liu, *et al*. An exploration of mashups and their potential educational uses. *Computers in the Schools*, *25*(3), 2008, 243-258.

[12] S. Liu, and L. Palen. The new cartographers: Crisis map mashups and the emergence of neogeographic practice. *Cartography and Geographic Information Science*, *37*(1), 2010, 69-90.

[13] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. 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.

[14] T. Mitchell. Machine Learning, *McGraw-Hill*, 1997.

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

[16] R. Moser, W. Pedrycz, and G. Succi. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. *Proceedings of the 30th International Conference on Software Engineering*, 2008, 181-190.

[17] C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Using Traits of Web Macro Scripts to Predict Reuse, *Journal of Visual Languages and Computing*. *21*, 277-291.

[18] D. Sjoberg, T. Dyba, B. Anda, and J. Hannay. Building theories in software engineering. *Guide to Advanced Empirical Software Engineering*, 2008, 312-336.

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

[20] 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.

[21] S. Stumpf, V. Rajaram, L. Li, and M. Burnett. Toward Harnessing User Feedback for Machine Learning. *Proceedings of the 12th ACM International Conference on Intelligent User Interfaces*, 2007, 82-91.

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

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

[24] J. Zappen, T. Harrison, and D. Watson. A new paradigm for designing e-government: Web 2.0 and experience design. *Proceedings of the 2008 International Conference on Digital Government Research*, 2008, 17-26.