

Using Traits of Web Macro Scripts to Predict Reuse

Chris Scaffidi¹, Chris Bogart¹, Margaret Burnett¹, Allen Cypher², Brad Myers³, Mary Shaw³

¹ Oregon State University, {cscaffid, bogart, burnett}@eecs.oregonstate.edu

² IBM Research-Almaden, acypher@us.ibm.com

³ Carnegie Mellon University, {bam, mary.shaw}@cs.cmu.edu

Corresponding Author

Christopher Scaffidi

cscaffid@eecs.oregonstate.edu

School of Electrical Engineering and Computer Science, Oregon State University

1148 Kelley Engineering Center, Oregon State University, Corvallis, OR 97331-4501

541-737-5572 (phone)

360-935-7708 (fax)

To help people find code that they might want to reuse, repositories of end-user code typically sort scripts by number of downloads, ratings, or other information based on prior uses of the code. However, this information is unavailable when code is new or when it has not yet been reused. Addressing this problem requires identifying reusable code based solely 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 actual reuse. These tests confirmed that the traits generally did correspond to higher levels of reuse as anticipated. We then developed a machine learning model that uses these traits as features to predict reuse of macros. Evaluating this model on repository logs showed that its accuracy is comparable to that of existing machine learning models for predicting reuse—but with a much simpler structure. Sensitivity analysis revealed that our model is quite robust; its quality is greatly reduced only when parameters are set to such extreme values that the model becomes inordinately selective. Testing the model with individual traits revealed those that provided the best predictions on their own. Based on these results, we outline opportunities for using our model to improve repositories of end-user code.

Keywords: end-user programming; end-user software engineering; repositories; reuse; web macros

Portions of this paper previously appeared as C. Scaffidi, C. Bogart, M. Burnett, A. Cypher, B. Myers, and M. Shaw. Predicting Reuse of End-User Web Macro Scripts, *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009)*, Corvallis, OR, September 2009, 93-100.

1 Introduction

End users sometimes complete programming tasks by reusing existing code. Examples of reusable code include spreadsheets, educational simulations, and web macros—scripts that automate form-filling and navigation operations in a web browser [15]. Interviews of office workers have revealed the importance of reuse in web macro programming [14]. One motivation for reuse is that copying and customizing a script sometimes is faster than creating one from scratch. Another motivation is that an end user may not know how to perform a task, so a macro not only automates the task, but it also shows how to accomplish the task.

Since not all end-user code is equally reusable, repositories typically provide a few limited features aimed at helping people to find reusable code. For example, many repositories display download counters, ratings, and other popularity measures for each script, as indicators of whether other people have successfully reused scripts. Repositories can also use these popularity measures to help sort search results, or as a form of “scoring system” to facilitate reuse. However, using these popularity measures to identify reusable code is problematic, since they depend on information that is available only after at least one person has tried to reuse a script. This is a serious limitation because it introduces a circularity—in order for these tools to help people find useful code, many people must have *already* found that code.

IBM’s CoScripter repository of web macros illustrates these limitations [14], which are shared by repositories of spreadsheets [35], educational simulations [31], and other end-user code. To find a reusable macro in the CoScripter repository, users can sort scripts by number of executions in the past week, number of times scripts have been marked as a “favorite,” and similar popularity measures based on prior use. But these measures are unavailable for most scripts. Of the 4681 scripts that are publicly visible as of April 2010, only 27 (0.6%) have been run in the past week, and only 55 (1.1%) have ever been marked even once as a favorite. Consequently, the repository’s popularity-based sorting features essentially place the vast majority of scripts in a blanket coarse-grained category of “unpopular” (or perhaps “not yet popular”), thus providing virtually no help in distinguishing reusable code from less reusable code.

Reusability of code must therefore be predicted based on information other than popularity. Prior work has shown that doing so is feasible in the context of professional programming. For example, one approach computes object-oriented metrics that empirically correspond to reusability [1]. This approach is attractive because it operates directly on code, requiring no information about popularity; moreover, most of these metrics are straightforward to compute, so it is easy for a programmer to understand why certain

code is identified by the model as reusable while other code is not. While this approach is appropriate for Java-like code, it is not applicable to scripts and spreadsheets that lack object-oriented structure. Likewise, for similar reasons, it is not possible to apply other existing approaches that rely on types, inheritance hierarchies, call graphs, specifications, or careful documentation.

Our approach is to use traits of end-user code in order to predict what is likely to be reused—even before it becomes popular—as a proxy for identifying what code is reusable. In order to identify code that is likely to be reused, we have developed a new model that uses traits of scripts as machine learning features, then predicts whether each script will ever be reused. Unlike popularity-based approaches, our model requires only information that is available at the moment that a script is created (though our model could be extended in future work to also incorporate information generated after script creation); consequently, it can be applied even to code that has not yet been downloaded, marked as a favorite, or otherwise reused or rated. Unlike existing approaches aimed at object-oriented code, the code traits used by our model are extremely simple and do not depend on object-oriented structure, call graphs, types, or similar trappings of the kinds of languages more typically used by professional software engineers. Because our model has a simple structure, it is also potentially conducive to explaining its predictions, and we anticipate that it will even be possible someday to automatically generate explanations to users of why a repository has identified particular end-user scripts as likely or unlikely to be reused. To our knowledge, ours is the first model that can provide any prediction about the reuse of scripts in that vast mass of “unpopular” code that repositories currently leave unsorted.

We have tested this model on CoScripter repository logs, showing that it can predict with 70-80% recall (at 40% false positive rate) whether a script will ever be reused—an accuracy comparable to that of comparable models used on similar software engineering problems. It is also robust: when we analyzed the sensitivity of our model to parameter perturbations, we found that model quality was largely unaffected for all but the most extreme parameter values. We tested the model’s accuracy when trained on individual traits, and learned that the traits with highest usefulness to the model were those related to author expertise and indicators of a script’s mass appeal.

The remainder of this paper is organized as follows. In Section 2, we introduce web macro scripting and discuss the limitations of existing models for identifying reusable code. In Section 3, we identify web macro traits that might plausibly serve as predictors of reuse, and we discuss statistical tests aimed at assessing how well these empirically do correspond to levels of reuse. In Sections 4 and 5, we present and evaluate our model that combines script traits to predict reuse. In Section 6, we discuss limitations and

threats to validity. In Section 7, we conclude by outlining opportunities for future work, including the development of repository enhancements based on our new model.

2 Related Work

End-user programmers are the millions of people who, although it is not their main job, create spreadsheets, scripts, and other programs to automate computations and organize data [27]. Web macro scripting is a form of end-user programming that automates repetitive tasks on the web, such as navigating and filling out forms. As we describe in this section, the traits of end users' web macro scripting are such that existing approaches have not addressed the need to identify macros that people would want to reuse.

2.1 CoScripter

IBM's CoScripter web macro tool (formerly called Koala) records end-user programmers' actions in the Firefox browser as re-playable "macro scripts" [15]. For example, a script might instruct the browser to submit a form at www.aa.com to look up a flight's status (Fig. 1). The author can edit the script, possibly giving it a title, modifying recorded instructions, or adding comments. He can change literal strings to variable references (which are read at runtime from a per-user configuration file called the Personal Database) and can insert "mixed-initiative" operations that cause CoScripter to pause at runtime while the user makes decisions and performs actions manually.

All scripts are stored on a wiki so that other users can run, edit, or copy-and-customize them. By default, each script is publicly visible and modifiable, though its author can mark it "private" so that it is not visible to others. Containing scripts created by thousands of users during the past year, the CoScripter wiki provides a wealth of data for studying the reuse of end-user code.

The rapid adoption of CoScripter is somewhat remarkable because CoScripter does not yet support some basic primitives provided by most professional programmers' languages, such as conditionals, loops, and structured data. Personal Database variables are untyped and read-only (scripts cannot update variable values). Scripts can call one another, but they cannot pass parameters. Moreover, only a few CoScripter macros in the entire repository actually include inter-script calls.

2.2 Approaches to finding reusable code, based on prior use

The CoScripter wiki's user interface has provided six forms of information about the reusability of each macro:

1. The number of users who have downloaded it
2. The number of times it has been executed
3. The average rating that it has received from users
4. The number of times that it has been marked as a “favorite” by users
5. The reviews/comments that users have typed about the macro
6. Whether or not the macro is mentioned on the homepage or IBM’s weekly email newsletter to CoScripter users.

The first five of these pieces of information are popularity measures, which require that someone has previously tried out a macro. This limits their applicability—in fact, so few macros receive ratings that IBM removed the ratings feature from the wiki entirely. Meanwhile, putting a script on the homepage or newsletter requires an administrator to notice and evaluate the script. In practice, administrators do not have time to review all of the scripts. As a result, most scripts mentioned on the homepage are tutorials created by administrators with the specific intent of mentioning them on the homepage. This could be rectified without forcing administrators to review *all* macros if they had a more effective model for identifying a small set of macros that are very likely to be reusable. They could then review this set and choose a few for advertisement on the homepage or newsletter.

Other end-user programming repositories share the limitation that someone must try out code before its reusability can be evaluated. For example, the Matlab File Exchange repository shows download counters, ratings, and reviews [9]. The Forms/3 spreadsheet repository helps users evaluate code by letting them interactively try out spreadsheets, or see what outputs the spreadsheets compute from sample input values [35].

2.3 Approaches for finding reusable code written by professional programmers

Some repositories for professional programmers rely on information about prior uses of components. For instance, collaborative filtering makes recommendations of the form, “People like you found the following components to be helpful” [18][22], which requires that some programmers try a certain piece of code before it can be recommended to other programmers.

Other 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 not available in the context of end-user scripting.

For example, many repositories for professional programmers recommend components for reuse based on call graphs, inheritance hierarchies, method signatures, and similar information based on types [1][8][17][23]. It is also possible to use call graphs and code complexity metrics to predict whether code contains defects [4][11][13][20][21][25][28]. This is appropriate in repositories for Java-like code but less so for CoScripter macros, which rarely call each other, and do not contain conditionals or loops, inherit from each other, or contain statically typed variables. Other recommendations for professional programmers come from version information about classes that are often modified simultaneously [13][37], but with few calls between end-user programmers' scripts, there is little reason why they should be modified together. Still other recommendations depend on components' functional specifications [10], documentation and similar text [6], or other carefully provided meta-information [16], but end-user programming processes tend to be very informal and rarely include investing time in specification, documentation, or rich annotations [9][29][35].

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 [33]. Such a model's information requirements are not very restrictive, since 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 macro reuse.

3 Script traits that correspond to reuse

Unable to apply most existing approaches directly to the end-user programming setting, we turned to the broad principles and preconditions that undergird reuse in general. Biggerstaff and Richter explain that reuse involves four fundamental steps [2]: finding, understanding, modifying, and composing code. In our context, web macros rarely call one another, so composability is unlikely to play a major factor in determining script reuse. In addition, while high modifiability is obviously desirable, scripts are probably more reusable if they do not require modification prior to reuse.

Thus, predicting reuse of scripts requires identifying information that reflects the findability, understandability, and modification requirements of scripts prior to reuse. To support the use cases

described in the introduction, this information must be available at script creation-time. 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). The broad categories are:

- *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 [25] 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 [5]. Preconditions might limit flexibility and impede reuse.

3.2 Data sources and forms of reuse

Using the logs from the CoScripter repository over a six month period, we extracted the 937 public scripts that appeared in the repository during that time. This repository is primarily used by non-IBM employees, since IBM employees mainly use a small company-internal repository instead. For each of the 937 scripts, we also retrieved the initial source code from version control. We considered four forms of reuse, all of which are logged by the CoScripter wiki:

- *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 because it seemed reasonable to assume that such executions would 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 to try predicting binary reuse 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 [3]: 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. Based on these considerations, 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. We then marked a script as reused (according to the binary criteria above) if a periodic program or a human user appeared to have run it.

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 Method for testing candidate traits

For each trait, we divided scripts into two groups based on whether the 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 variables described in Section 3.2. Finally, for each combination of trait and form of reuse, we performed a one-tailed z-test of proportions. In cases where the correspondence between script trait and reuse was actually opposite what we expected, we also report whether a one-tailed z-test would have been significant in the opposite direction.

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

reuse variables. Thus, the correction establishes a conservative 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 of testing candidate traits

The rightmost four columns of Table 1 (in Section 3.1) summarize the empirical results of our statistical tests. Non-empty cells in these columns 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 $p < 0.05$ [34]).

Overall, 21 of the 35 script traits generally corresponded to reuse as we hypothesized—that is, for each of these 21 traits, more tests were significant in the direction that we anticipated than not. Far more tests were significant in the anticipated direction than the 7 of 140 that would have been expected to be significant at $p < .05$ by chance. Thus, these results suggest that the predictiveness of these 21 traits are not mere statistical anomalies.

In addition to confirming that reuse statistically corresponds to many end-user script traits available at script creation time, these results revealed interesting insights and questions regarding the relationship between traits and script reuse. In some cases, the questions create research opportunities about the dynamics of script creation and reuse in the CoScripter community. In Section 7, we return to these questions and discuss their implications for future work.

First, different reuse measures corresponded to different traits. This suggests that predicting different kinds of reuse will require using different traits. Therefore, in subsequent sections, we will introduce and evaluate a machine learning model that can be weighted with different traits to predict different kinds of reuse.

Second, focusing on the Length traits, longer scripts were more likely to be reused than shorter scripts. Therefore, the promise of higher functionality appears to outweigh the risks of more bugs and of increased difficulty of understanding longer code (counter to our hypothesis). The latter risk might have

been mitigated by the fact that the correspondence was between script length and *author* reuse, not general reuse. That is, longer web macros still might be less understandable to people other than the author.

Third, several traits motivated by findability corresponded to higher reuse by other users but not to reuse by the author. In other words, it appears that a script needs to be findable if one person is going to reuse another person’s script, but that it is not so important for a script to be findable in order that its author can reuse it. We see two explanations why findability traits are more important for reuse by others than they are for reuse by authors. First, the CoScripter wiki provides a special screen where authors can easily see all of their own scripts regardless of whether those scripts have findability traits; we suspect that this greatly simplifies authors’ reuse of their scripts and reduces the importance of findability traits for reuse by the author. Second, the findability traits largely related to mass appeal—that is, whether the script is likely to appeal to the interests of a broad range of users. But an author seems unlikely to create a script unless that script has some appeal to the author, regardless of whether the script has mass appeal. Therefore, the presence or absence of findability traits related to mass appeal should have little effect on reuse by the author, which is what we observed empirically.

Finally, traits based on authorship suggest that there may be two interesting subpopulations of repository users. The `prev_created` and `prev_selfexec` traits defined in Table 1 positively corresponded to self-exec reuse; that is, if a person has previously created and reused his own macros, then he appears likely to continue doing so. Yet `prev_created` and `prev_selfexec` also negatively corresponded to other-exec, so when authors frequently create macros for their own reuse, such macros seem to be of little interest to other users. Conversely, there also appears to be a second class of users who are very other-centered. Specifically, `prev_otherexec`, `prev_otheredit`, and `prev_othercopy` were negatively related to self-exec but positively related to other-exec. Such macros seem to be created by users who were intent on creating useful macros that other people would execute. Moreover, other people rarely needed to edit or clone these macros, suggesting that the macros might have been of such high reusability that they did not need any modification before later 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, given a form of reuse, selects rules that predict that reuse variable on a training set of scripts. 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 algorithms for training and using this model, we evaluate it by comparing its quality to that of several models commonly used on other software engineering problems.

4.1 Training and using the predictive model

Our algorithm trains a predictive model (Fig. 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 \bar{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 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 (Fig. 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 it 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 .

4.2 Design considerations

Designing our model involved choices that resulted in differences from existing machine learning models.

First, when selecting predictors, TrainModel chooses a threshold based on differences in proportions rather than commonly used entropy-based measures [24]. We made this decision because difference-in-proportions corresponds more directly than difference-in-entropy to the principle that ideal predictors match all reused scripts but no un-reused scripts. After making this decision, we empirically evaluated its impact (Section 5.4) and found that our choice contributed to model quality.

Second, another option would have been to select a threshold maximizing the difference in means (between reused and un-reused scripts), rather than the difference between above-threshold proportions. Evaluating this decision (Section 5.4) showed that it had a negligible impact.

Finally, as noted in the introduction, we ultimately hope to use our model to develop system features that identify reusable code to end-user programmers, and it will be reasonable for users to question why a recommendation was made. Typical machine learning algorithms combine predictors into complex structures, such as a decision tree or a graph [24]. Explaining recommendations generated from complex

models requires complex explanations [12], which can be unintuitive to users [32]. 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 script. Simplicity’s risk is that it might reduce the model’s quality, compared to more complex machine learning models, so we must empirically compare our model’s accuracy to that of more complex models.

5 Evaluation

Ten-fold cross-validation is the usual method used when training and evaluating a model on data [24]. We used this approach to evaluate the quality of our model, variants of our model, configurations of our model with specific parameter settings, and executions of our model using just a single trait at a time.

5.1 Quality measures

A variety of quality measures have been used in machine learning research. Our approach of training a model on script traits in order to predict reuse resembles the standard approach in software engineering defect prediction, which trains a model on module traits (static code traits or process data) in order to predict the presence of defects. The primary quality measures used in that literature are False Positive (FP) and True Positive (TP) rates [4][13], defined as in Fig. 4.

TP is the same as the recall measure used in information retrieval [24], indicating the fraction of interesting items (reused scripts) correctly identified. FP indicates the fraction of uninteresting items (unreused 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 [24], but FP is often preferred over precision in software engineering, since FP is more robust than precision to small changes in the experimental data [19].

In general, TP and FP trade off against one another. TP can typically be raised by adjusting a threshold so that a machine learning model classifies more objects in the “positive” category (in our case, classifying macros as reusable). This simultaneously tends to raise FP, since the model will also begin to classify objects as positive when in fact they should have been placed in the negative category. Consequently, machine learning models are typically tested at a range of different thresholds [24]. In addition, the quantity TP-FP is often valuable to compute. High values are ideal; conversely, an algorithm that completely randomly placed objects in the positive and negative categories would result in TP-FP equaling zero.

5.2 Evaluating model quality

For each form of reuse, we performed validation at varying levels of α and β (Fig. 5). Adjusting α and holding β constant, or vice versa, generated the full range of TP and FP. For example, for the self-exec reuse variable, FP is closest to 0.4 at $\alpha = 0.16$ and $\beta = 5$; for the others, FP is also closest to 0.4 at $\alpha = 0.16$ but with $\beta = 3$. On the vertical axis, TP was in the range 0.7-0.8 when FP reached 0.4. Conversely, TP reached approximately 0.7 when FP ranged from 0.05-0.35, depending on the form of reuse.

Such cutoffs of FP=0.4 or TP=0.7 are arbitrary, since parameters can be adjusted in any machine learning model so that TP goes up (along with FP) or FP goes down (along with TP). (We consider alternative parameter settings in Section 5.5.) It is the *difference* between TP and FP that matters most, in terms of quality.

Examining the difference between TP and FP, the prediction quality of our model was comparable to that of other research that applies machine learning to software engineering problems. For example, $TP \leq FP + 0.3$ for many defect prediction models [19][25]. 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.

5.3 Comparison to other models

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 [25].

However, these models are substantially more complex than our model, which is of the form, "This code will be reused if at least β of the following constraints are true: $a_1 \geq \tau_1, a_2 \geq \tau_2, a_3 \geq \tau_3, \dots$ " With logistic regression, the prediction takes the form, "This code will be reused if $z/(1+e^{-z})$ exceeds a certain threshold, where $z = k_1a_1 + k_2a_2 + k_3a_3 + \dots$ ", with the k_i being coefficients determined through regression. With J48, which internally uses a decision tree, the prediction can be explained as a disjunction of all the possible routes from the tree's root to leaf nodes. With Naïve Bayes, the prediction is still more complicated, requiring a lengthy product of conditional probabilities, which cannot be succinctly summarized.

With this complexity in mind, it is not surprising that deeply-nested models like those generated from Naïve Bayes and J48 have been found in experiments to be unintuitive to users [12][32], so we also compared our algorithm to one that generates fairly explainable models. Specifically, we compared to PART, a “straightforward and elegant” algorithm [7] that produces a tree like J48; however, the trees produced by PART are usually relatively shallow.

We used the Weka machine learning toolkit implementation of the algorithms [36], which exposes few tunable parameters for models, so we recorded the quality measures using model designers’ suggested (default) parameters. Table 2 summarizes the quantitative results, which Fig. 5 displays as large triangles for graphical comparison to our model. Comparing these results to the TP scores of our model at corresponding values of FP (Fig. 5), we found negligible differences in quality between our model and these alternative models. Thus, our model not only achieves equally high quality as these models, but it does so in a way that we hope will be amenable to automatically generating concise explanations of why the model made particular predictions.

5.4 Comparison of model variants

When designing the training algorithm, we had the option of several other heuristics for selecting the optimal threshold for each adjusted trait (Section 4.2). To evaluate our design decision, we repeated the evaluation using variants of our model that selected each adjusted trait’s optimal threshold τ_i based on difference-in-means or difference-in-entropy rather than difference-in-proportions (Table 3). The results confirmed our design decision, as difference-in-proportions yielded slightly higher quality than difference-in-means and substantially higher quality than difference-in-entropy.

5.5 Exploration of model sensitivity with respect to α and β parameters

Our model includes two tunable parameters α and β . If the model’s quality were highly sensitive to the specific settings of these parameters, then the model would be fairly brittle and difficult to apply to other data sets. On the other hand, if the model’s quality were largely insensitive to the specific settings, then one or both parameters might be set a constant, thereby simplifying the model and reducing the need to search for settings when applying the model in practice. In order to judge the model’s sensitivity to these parameters, we computed the maximal difference between TP and FP that could be obtained for given settings of α or β .

The training algorithm’s α parameter controls a pre-filter that essentially performs quality-control on the script traits, in order to prevent the training algorithm from creating predictors using traits that have little

information content. As shown by Fig. 6, we found that raising α (making the training more selective) did not substantially improve model quality—and, in fact, raising α even appeared to slightly reduce model quality in a few cases. This indicates that while pre-filtering the traits appears to be helpful for reducing the variance in prediction accuracy, it is not particularly helpful on average.

The β parameter controls how many predictors must be matched by a script in order for the model to predict that the script will be reused. A higher parameter setting makes the algorithm more selective, reducing both FP and TP. As shown by Fig. 7, we found that raising β did not strongly affect model quality as long as β remained lower than approximately 15. Beyond this point, however, the model appeared to become too selective, as model quality gradually decreased.

Overall, these results indicate that our model is quite robust. Its quality is greatly reduced only when parameters are set to such extreme values that the model becomes inordinately selective.

5.6 Evaluation of individual trait usefulness

To evaluate which traits were most useful for predictions, we attempted to train the model using only individual traits. In each case, we recorded the resulting value of TP-FP, as a measure of the information gain provided by that trait. When only a single trait is activated, the resulting TP-FP is exactly equal to the difference $\rho(R_{m, a_j}, \tau_j) - \rho(\bar{R}_{m, a_j}, \tau_j)$ used for selecting traits for inclusion during model training (Fig. 2). Therefore, this value of TP-FP indicates the maximal α for which a trait will be included in the model when other traits are present.

For example, we tested the model on *self-exec* with only the `keywrd_sim` trait. The resulting TP-FP was 0.23. This means that $\rho(R_{m, a_j}, \tau_j) - \rho(\bar{R}_{m, a_j}, \tau_j)$ was also 0.23 for that trait and that this trait would be used to generate a predictor any time that $\alpha \leq 0.23$. Only when α exceeds 0.23 is this trait prevented from generating a predictor. A trait with a higher value of TP-FP would be activated even for higher α thresholds, since such a trait has higher information content.

Fig. 8 displays the resulting values of TP-FP, as well as each trait’s value of TP-FP averaged over all four forms of reuse. Overall, the median average of TP-FP is 0.07, indicating that the traits by themselves generally did not contain a great deal of information. Except for a few traits (below), it is in combination that these traits are able to make useful predictions, rather than on their own.

The most useful category overall appeared to be that of Author traits, as 7 of these 10 traits had average TP-FP values that exceeded the overall median of averages. Moreover, even the worst of these 7 had a higher average TP-FP value than the best traits in any other category. The next most useful categories were those of Mass appeal and Length traits, as 5 of these 8 traits had average TP-FP values that exceeded the overall median of averages. These categories were followed by the Language and Flexibility categories, with Advanced Syntax and Preconditions producing no traits of particularly high usefulness.

5.7 Adapting the model in order to predict absolute reuse counts

As explained in Section 3.1, the currently-available absolute counts of web macro reuse probably are dominated by information cascades. Nonetheless, to explore whether we might someday adapt our model to predict absolute levels of reuse, 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 Fig. 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 (Fig. 9), though 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 suggest that it might be possible to adapt our model to predict absolute levels of reuse by other users. A substantially different model will be needed for absolute levels of self-exec reuse.

6 Discussion of Limitations and Threats to Validity

We have presented a machine learning model that uses traits of web macros to predict whether those scripts will be reused in four different ways. We identified candidate traits by considering what aspects of web macros might provide information about findability, understandability, and the need to modify scripts during reuse. We presented statistical tests showing that most of these traits corresponded to reuse as we hypothesized. We described the machine learning model that combines these traits to make predictions, and we tested this model with a range of different configurations. We found overall that the model performs about as well as many models used to solve other software engineering problems. While these results are encouraging, our process embodies a number of limitations and threats to validity.

6.1 Limitations

The primary limitation of this approach is that it only predicts reuse, not reusability. More precisely, it attempts to identify scripts that will eventually be reused, given enough time (several months in our experiment) and interest by other users, rather than the scripts that that *would* be easy to reuse if somebody happened to have a need for them. As noted in Section 2.3, measures are now being developed

for quantifying the reusability of professionally-written object-oriented code. However, no such measures are currently available for quantifying the reusability of web macros (or, for that matter, most other kinds of domain-specific end-user code), so it is impossible to develop a model that can predict the reusability of web macros until measures of web macro reusability become available.

A second limitation is that the model makes binary predictions of reuse rather than predictions of absolute amounts of reuse. Preliminary investigation did reveal a hint of a correspondence between absolute reuse and number of traits matched. However, the relationship was merely encouraging rather than compelling. Substantial model refinement will be required to make absolute predictions. (If measures of macro reusability become available, they might be represented on a ratio rather than binary scale, so predicting reusability 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 script is created. While this implies that the model can be used on scripts 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 script is created. Future work could aim to identify and incorporate such traits. The result would ideally be a model that can be applied in a basic form when a script is created, and the model's predictions would gradually increase in accuracy as more information becomes available.

6.2 Internal 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 reuse are accurate because the traits actually reveal information about why reuse occurs. The alternative possibility is that the traits really offer little or no information about reuse, but that the model's accuracy instead resulted from statistical effects (such as random chance or from overfitting during machine learning). One way to uncover whether the traits truly contain information about reusability would be to ask users about these traits, to find out how people decide which scripts to reuse and to learn if these traits make sense to them as predictors. Since we have relied on logs rather than interviews of users, we cannot state with certainty that the traits do contain information about reusability.

Nonetheless, there are several reasons why we believe that the traits actually do reveal information about reuse. First, every one of our script characteristics was motivated by a consideration of how findability, understandability, and modifiability (or lack of need for modification) would be reflected in script

characteristics. Second, most statistical tests showed that the traits corresponded to reuse as anticipated. Finally, in each case where characteristics corresponded to reuse in the opposite direction than anticipated, theoretical arguments explained the results (Section 3.4).

There are several reasons why we do not believe that statistical effects dominated in this experiment. First, we observed far more numerous significant statistical tests on traits than would be expected by chance, even using the extremely over-conservative Bonferroni correction. Second, except for PART, the other machine learning models generally performed nearly as well as ours (albeit with a more complex internal structure); therefore, random chance or overfitting would also need to affect these existing algorithms as well as ours, which seems unlikely due to the diversity and long track record of these algorithms. Finally, model quality was not particularly sensitive to parameter settings, except when β was raised to the point of extreme selectivity, so overfitting by selection of “lucky” parameter values is an implausible explanation for the model’s performance.

While these considerations suggest that the model is a valid method for predicting reuse, we do not argue that our results show that *modifying* a script’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 macro 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 [30]. We have essentially shown that our model is useful for predicting the phenomenon of web macro reuse; however, we have not yet shown that it is useful for guiding the design of macros in a way that raises or lowers reuse.

6.3 External validity

External validity, or generalizability, is concerned with the question of whether similar results would be obtained in other similar experiments. In our context, external validity is most strongly limited by the generalizability of script characteristics to other end-user programming platforms. Many characteristics are likely to generalize directly to web macros created in other programming tools, as few characteristics are specific to CoScripter (`forum_posts`, `ibm`, `ctl_click`, and `mixed_init`). Moreover, many might generalize at an abstract level to other kinds of end-user programming, such as spreadsheets. For example, while it would not be meaningful to count the number of lines of code `total_lines` in a spreadsheet, it would be meaningful to count the number of cells containing formulas. Just as `total_lines` corresponded to macro reuse, the number of cells containing formulas might correspond to spreadsheet reuse, since in each case, larger programs have more functionality, which might increase the value and likelihood of

reuse. Given appropriate repository logs, spreadsheet characteristics like these could be tested using the process demonstrated in this paper.

Our overall model has already worked well for four forms of reuse. Because it is defined in terms of reuse variables and characteristics, it does not require that the programs under consideration should actually be web macro scripts. Thus, while other code characteristics might be more suitable for other kinds of programs, the model and the process of training and applying the model should still be relevant.

7 Conclusions and Opportunities for Applying the Model

We have presented a model that can predict whether a web script will be reused by the script's author or other end-user programmers. All of the traits used as machine learning features by this model were motivated by considerations of web macro findability, understandability and modification requirements, and statistical tests confirm that these traits do generally correspond to reuse as anticipated.

Our model combines these traits to make predictions that are as accurate as those made by other models for similar software engineering problems. The model's accuracy also meets or exceeds that of more complex models whose predictions would be more difficult to explain to users. In particular, our results show that our model's accuracy varies little over a range of its parameter values, suggesting that we could further simplify our model by hardcoding these parameters, with little loss of accuracy.

Moreover, we have empirically identified which traits that have the highest information gain. The best were related to macros' authorship, mass appeal and length, while other moderately useful traits included those related to flexibility and language. While it might have seemed obvious that mass appeal and language would affect reuse, it was certainly not obvious that many other traits such as authorship and flexibility should have had a strong statistical relationship to multiple forms of reuse, nor that all these traits together would have contained enough information to make such accurate predictions of reuse.

Our future work will aim to deepen our understanding of how individual traits affect model quality, as well as to test and broaden its generalizability. In particular, we will use data from other end-user repositories to measure how well the model can predict reuse of other kinds of code, how well its accuracy holds up when only subsets of traits are available, how much its accuracy improves when our traits are augmented with information generated *after* script creation, how well the model can predict other forms of reuse, and how well its accuracy holds up when we train it on one data set and test it on later user logs (thereby testing its ability to predict future reuse based on historical data). In addition,

because this approach predicts reuse rather than reusability, interviews or surveys of users could explore whether these traits actually affect their decisions about what code to reuse.

In addition, we will empirically follow up on interesting speculative points raised by our results. For example, as noted in Section 3.4, it appears that there might be two interesting subpopulations of users in the CoScripter community. One group seems to frequently create macros that are suitable for the authors' reuse but of little value to the community at large, while another group of users seems to create macros that are of broader community value. Similar groups have previously been observed in the context of CAD programming (where people in the latter group are called "gardeners") [26], but such groups have not yet been empirically observed or characterized in the context of web scripting. Further analysis of repository logs and interviews of users would help to clarify whether these groups do exist, what their motivations truly are, what their programming practices are, and what problems they encounter that researchers might seek to solve.

As we gain a deeper and broader understanding of our model's strengths and limits, as well as a clearer picture of how reuse fits into the dynamics of script creation and reuse by the community, we will also develop concrete designs for new repository enhancements based on our model. For example, we may extend search engines to incorporate predictions of code value. Specifically, 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 be oft-reused.

Adding such a metric to a search engine's list of criteria would lessen the relative importance of raw popularity, thereby helping to break the circular logic of popularity-begets-popularity and dampening the information cascade feedback loop present in typical repositories. This could reduce the problem of highly-ranked but unreusable scripts, whose popularity is perpetuated by the clicks of curious first-time users. Conversely, it could also enable users to find promising scripts that have not yet been "discovered" by the masses. Moreover, it could foster collaboration by enabling users to find potential programming partners whose scripts are not yet popular but who can write high-quality code. We hope to provide these and many similar benefits by breaking the circular logic reflected in popularity-centric repositories of end-user code.

8 Acknowledgements

This work was supported by the EUSES Consortium via NSF ITR-0325273, by NSF grants CCF-0438929, CCF-0613823, and IIS-0917366, and by an IBM International Faculty Award. Opinions, findings, and recommendations are the authors' and not necessarily those of the sponsors.

9 References

- [1] V. Basili, L. Briand, and W. Melo. A Validation of Object-Oriented Design Metrics as Quality-Indicators. *IEEE Transactions on Software Engineering* (22), 10, 1996, 751-761.
- [2] T. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions. *IEEE Software* (4), 2, 1987, 41-49.
- [3] S. Bikhchandani, D. Hirshleifer, and I. Welch. A Theory of Fads, Fashion, Custom, and Cultural Change as Informational Cascades. *Journal of Political Economy* (100), 5, 1992, 992-1026.
- [4] G. Boetticher, T. Menzies, T. Ostrand, and G. Ruhe. 4th International Workshop on Predictor Models in Software Engineering. *Companion Proceedings of the 30th International Conference on Software Engineering*, 2008, 1061-1062.
- [5] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. End-User Programming in the Wild: A Field Study of CoScripter Scripts. *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2008, 39-46.
- [6] D. Čubranić and G. Murphy. Hipikat: Recommending Pertinent Software Development Artifacts. *Proceedings of the 25th International Conference on Software Engineering*, 2003, 408-418.
- [7] E. Frank and I. Witten. Generating Accurate Rule Sets Without Global Optimization. *Proceedings of the 15th International Conference on Machine Learning*, 1998, 144-151.
- [8] G. Gui and P. Scott. Coupling and Cohesion Measures for Evaluation of Component Reusability. *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, 18-21.
- [9] N. Gulley. Improving the Quality of Contributed Software and the MATLAB File Exchange. *Proceedings of the 2nd Workshop on End User Software Engineering*, 2006, 8-9.
- [10] R. Hall. Generalized Behavior-Based Retrieval. *Proceedings of the 15th International Conference on Software Engineering*, 1993, 371-380.
- [11] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow. Comparing Design and Code Metrics for Software Quality Prediction. *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, 2008, 11-18.
- [12] U. Johansson, L. Niklasson, and R. König. Accuracy vs. Comprehensibility in Data Mining Models. *Proceedings of the 7th International Conference on Information Fusion*, 2004, 295-300.

- [13] M. Lanza, M. Godfrey, and S. Kim. 5th Working Conference on Mining Software Repositories. Companion *Proceedings of the 30th International Conference on Software Engineering*, 2008, 1037-1038.
- [14] G. Leshed, M. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise. *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems*, 2008, 1719-1728.
- [15] G. Little, T. Lau, A. Cypher, J. Lin, E. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of the 25th SIGCHI Conference on Human Factors in Computing Systems*, 2007, 943-946.
- [16] V. Lucena. Facet-Based Classification Scheme for Industrial Automation Software Components. *Proceedings of the 6th International Workshop on Component-Oriented Programming*, 2001.
- [17] D. Mandelin, L Xu, R Bodík, and D Kimelman. Jungloid Mining: Helping To Navigate the API Jungle. *Proceedings of the 2005 SIGPLAN Conference on Programming Languages Design and Implementation*, 2005, 48-61.
- [18] F. McCarey and M. Cinneide. Rascal: A Recommender Agent for Agile Reuse. *Artificial Intelligence Review (24)*, 3-4, 2005, 253-276.
- [19] 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'. *IEEE Transactions on Software Engineering (33)*, 9, 2007, 637-640.
- [20] T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering (33)*, 1, 2007, 2-13.
- [21] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of Ceiling Effects in Defect Predictors. *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, 2008, 47-54.
- [22] A. Michail. Data Mining Library Reuse Patterns Using Generalized Association Rules. *Proceedings of the 22nd International Conference on Software Engineering*, 2000, 167-176.
- [23] R. Mili, A. Mili, and R. Mittermeir. Storing and Retrieving Software Components: A Refinement Based System. *IEEE Transactions on Software Engineering (23)*, 7, 1997, 445-460.
- [24] T. Mitchell. Machine Learning, *McGraw-Hill*, 1997.
- [25] 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.
- [26] B. Nardi. *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, 1993.

- [27] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, 207-214.
- [28] S. Schach, and X. Yang. Metrics for Targeting Candidates for Reuse: An Experimental Approach. *Proceedings of the 1995 Symposium on Applied Computing*, 1995, 379-383.
- [29] J. Segal. *Professional End User Developers and Software Development Knowledge*. Technical Report 2004/25, Department of Computing, Faculty of Mathematics and Computing, The Open University, Milton Keynes, United Kingdom, 2004.
- [30] D. Sjöberg, T. Dyba, B. Anda, and J. Hannay. Building theories in software engineering. *Guide to Advanced Empirical Software Engineering*, 2008, 312-336.
- [31] G. Stahl, T. Sumner, and A. Repenning. Internet Repositories for Collaborative Learning: Supporting Both Students and Teachers. *Proceedings of the 1st International Conference on Computer Support for Collaborative Learning*, 1995, 321-328.
- [32] 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.
- [33] G. Suryanarayana, M. Diallo, J. Erenkrantz, and R. Taylor. Architectural Support for Trust Models in Decentralized Applications. *Proceedings of the 28th International Conference on Software Engineering*, 2006, 52-61.
- [34] D. Wackerly, W. Mendenhall, and R. Scheaffer. *Mathematical Statistics with Applications*, Duxbury Press, 2001.
- [35] R. Walpole and M. Burnett. Supporting Reuse of Evolving Visual Code. *Proceedings of the 1997 IEEE Symposium on Visual Languages*, 1997, 68-75.
- [36] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2005.
- [37] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering (31)*, 6, 2005, 429-445.

- Fig. 1. The current step of the script (left) causes CoScripter to highlight Flight Number (right) and fill it in from the user's "Personal Database" configuration file (lower left).
- Fig. 2. Selecting predictors during model training
- Fig. 3. Predicting if a script will be reused
- Fig. 4. Definitions of True Positive (TP) and False Positive (FP) quality measures
- Fig. 5. Model quality at various parameter values. Section 5.3 compares results to other models
- Fig. 6. Model quality was little improved (or even reduced slightly) by raising alpha beyond a small value of 0.05. Except for the other-edit reuse variable, model quality did not vary sharply depending on the specific value of alpha.
- Fig. 7. Raising beta beyond approximately 15 generally reduced model quality by a great deal. For lower values of beta, model quality varied over a much narrower range.
- Fig. 8. Model quality when trained on individual traits. Bars indicate TP-FP for each trait alone. The number along the vertical axis indicates the average TP-FP over all bars for that trait. Averages are bolded if they exceed the median average of 0.07.
- Fig. 9. Absolute level of reuse rose sharply with the number of matches for three out of four predictors; for self-exec, no clear trend was visible.

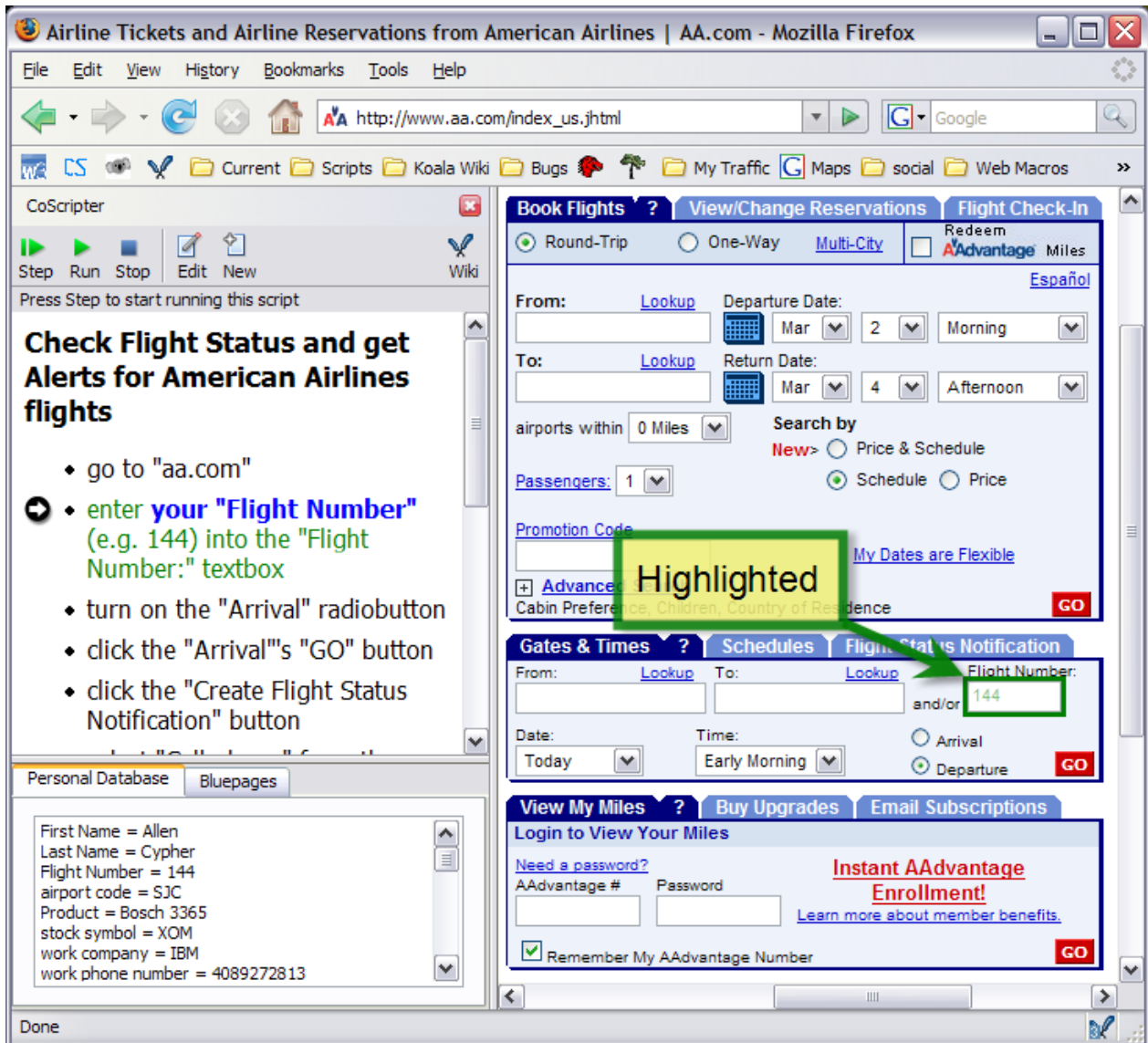


Fig. 1. The current step of the script (left) causes CoScripter to highlight Flight Number (right) and fill it in from the user's "Personal Database" configuration file (lower left).

TrainModel

Inputs: Training scripts $R' \subseteq \text{Repository } R$,

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 $\bar{R}_m = \{s \in R' : m(s) = 0\}$

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 $p(R_m, c_i, \mu_i) \geq p(\bar{R}_m, c_i, \mu_i)$

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

Compute threshold (through exhaustive search)

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

If $p(R_m, a_i, \tau_i) - p(\bar{R}_m, a_i, \tau_i) \geq \alpha$

then add the following predictor to Q ...

$q_i(s) = 1$ if $a_i(s) \geq \tau_i$ and 0 otherwise

Return Q

Fig. 2. Selecting predictors during model training

EvalScript

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

Let $n\text{matches} = 0$

For each $q_i \in Q$

 If $q_i(s) = 1$ then $n\text{matches} = n\text{matches} + 1$

If $n\text{matches} \geq \beta$ then return 1 else return 0

Fig. 3. Predicting if a script will be reused

Prediction	Actual	
	<i>Reused</i>	<i>Not reused</i>
<i>Reused</i>	a	b
<i>Not reused</i>	c	d

$$TP = \frac{a}{(a + c)} \quad FP = \frac{b}{(b + d)}$$

Fig. 4. Definitions of True Positive (TP) and False Positive (FP) quality measures

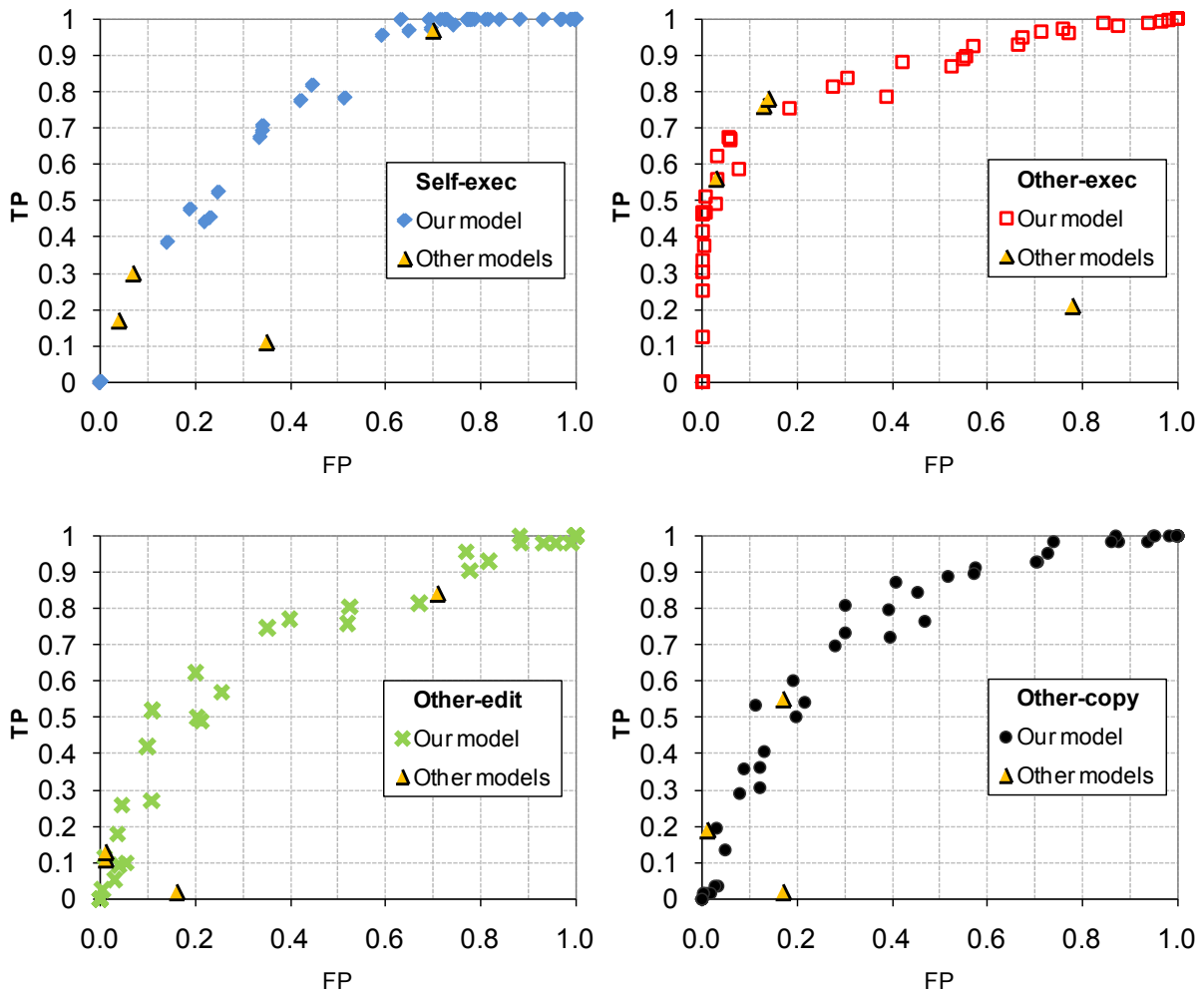


Fig. 5. Model quality at various parameter values. Section 5.3 compares results to other models

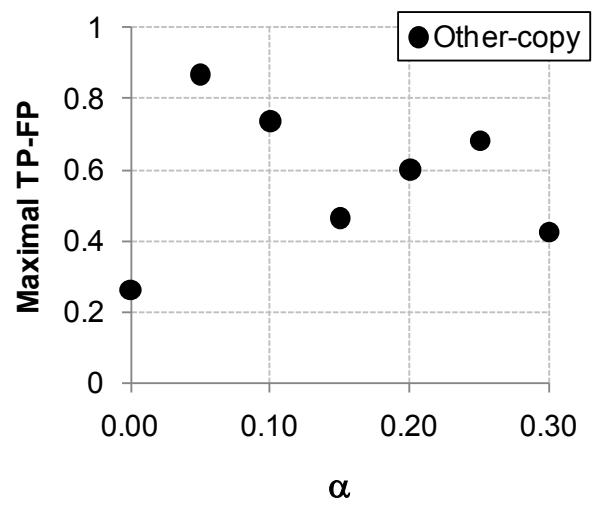
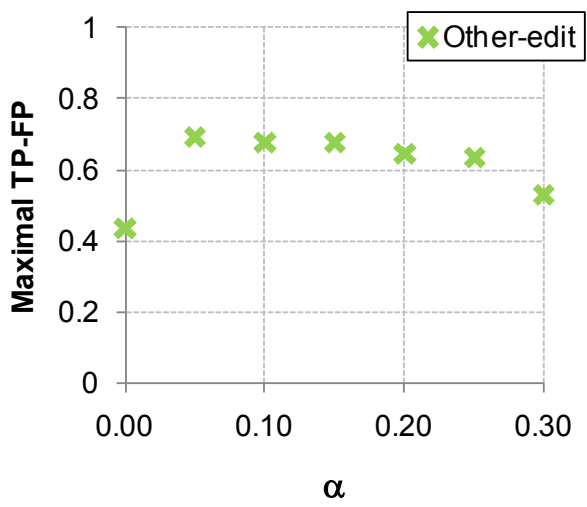
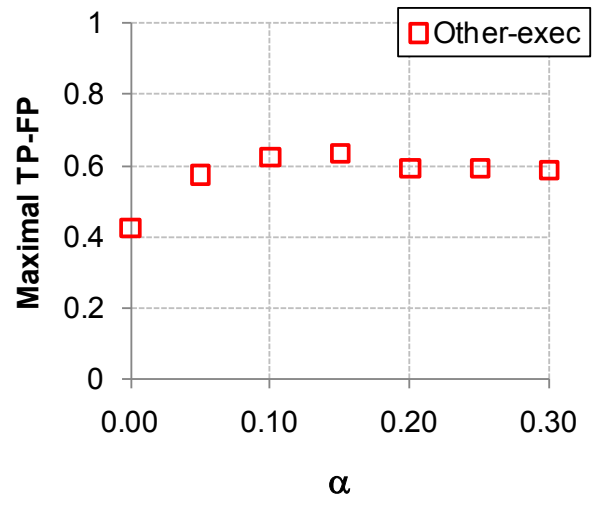
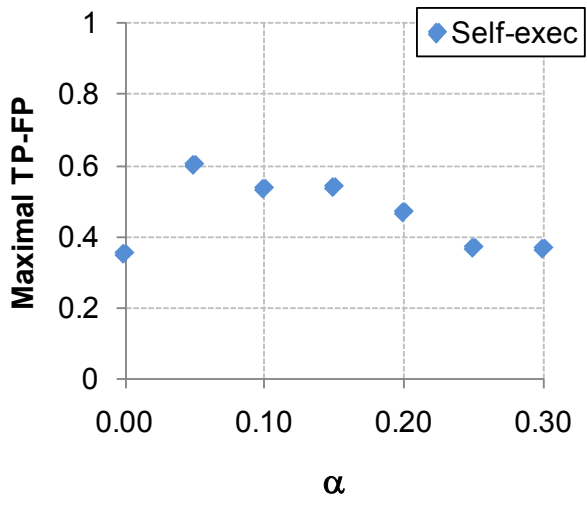


Fig. 6. Model quality was little improved (or even reduced slightly) by raising alpha beyond a small value of 0.05. Except for the other-edit reuse variable, model quality did not vary sharply depending on the specific value of alpha.

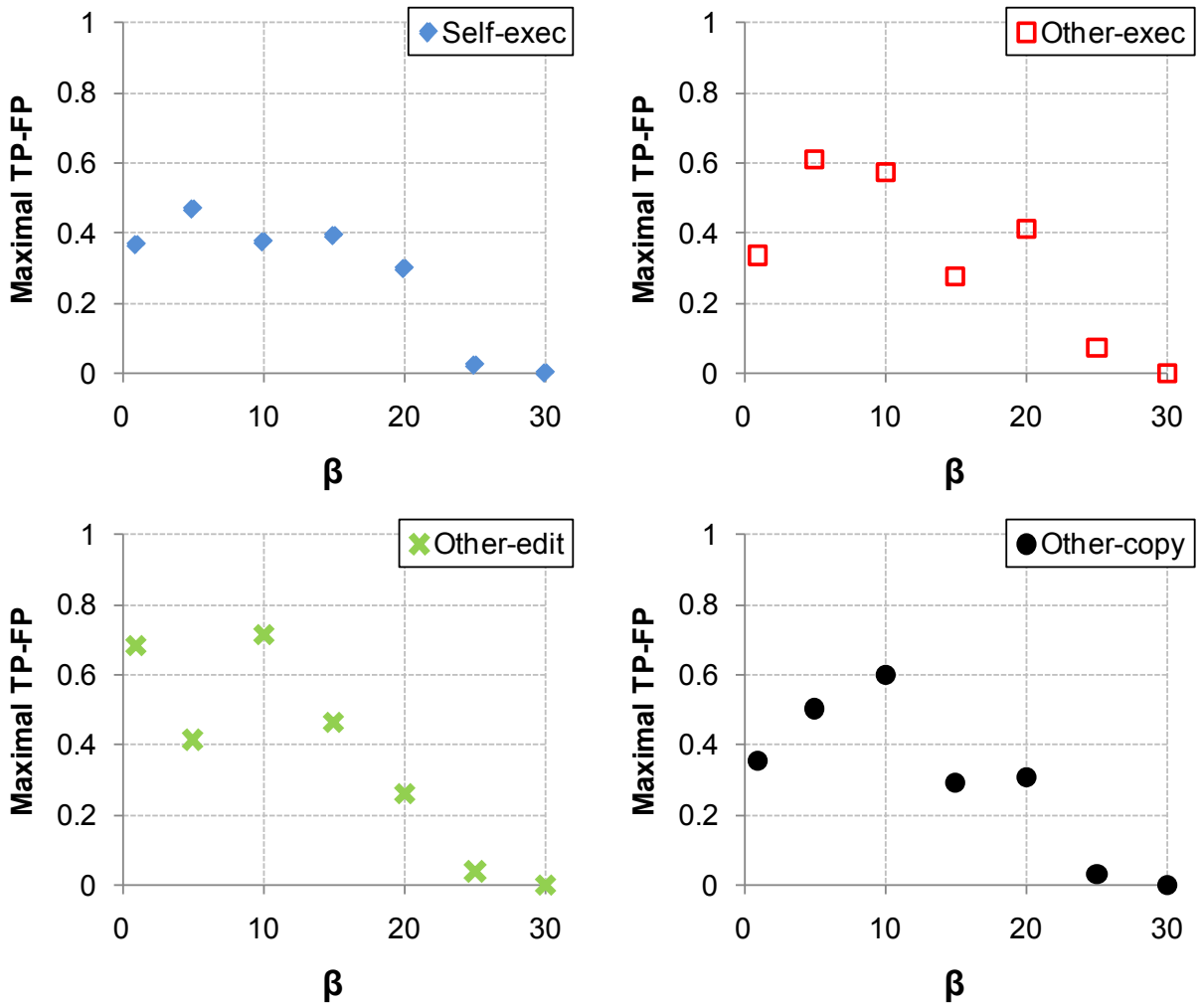


Fig. 7. Raising beta beyond approximately 15 generally reduced model quality by a great deal. For lower values of beta, model quality varied over a much narrower range.

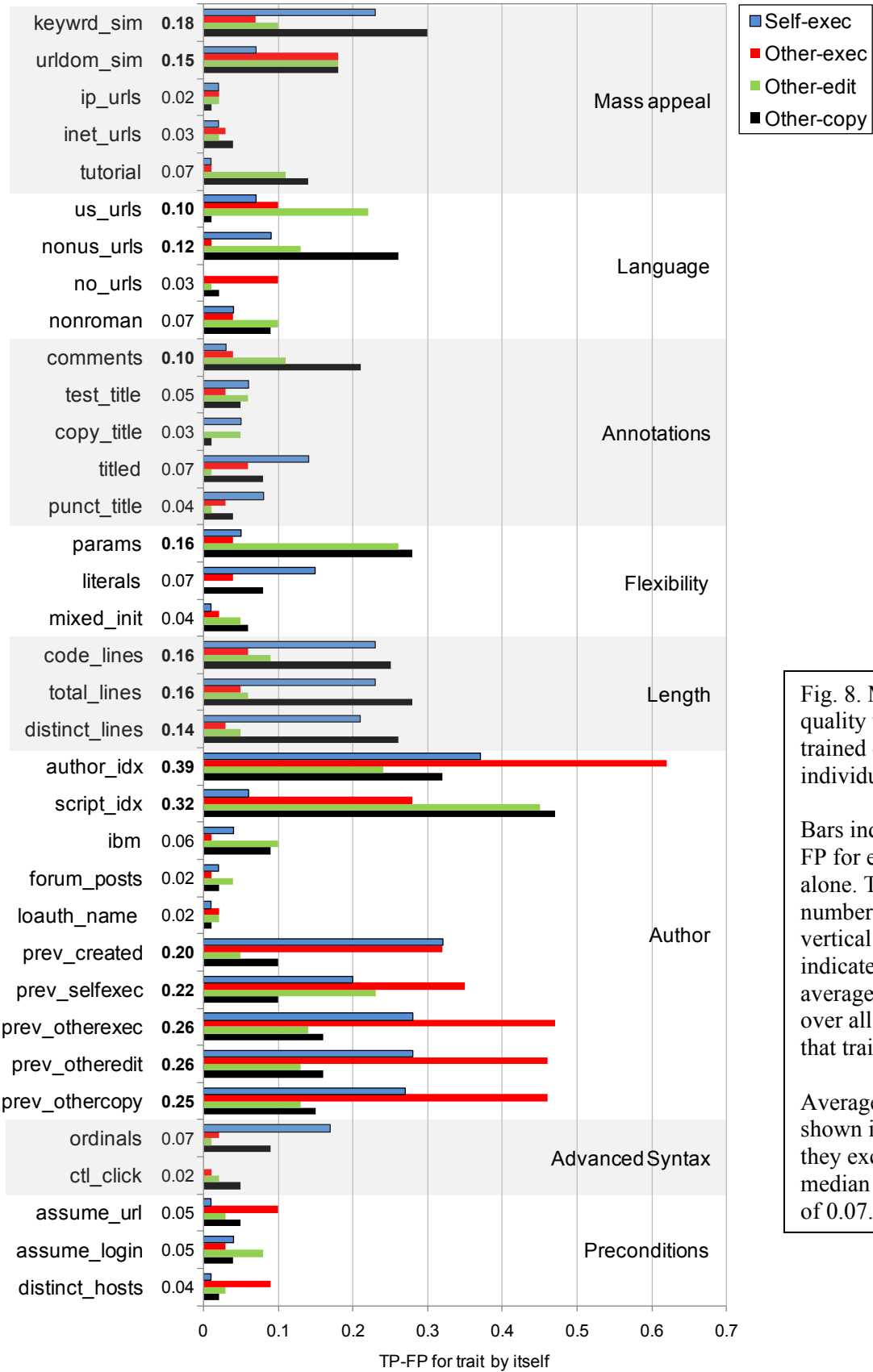


Fig. 8. Model quality when trained on individual traits.

Bars indicate TP-FP for each trait alone. The number along the vertical axis indicates the average TP-FP over all bars for that trait.

Averages are shown in bold if they exceed the median average of 0.07.

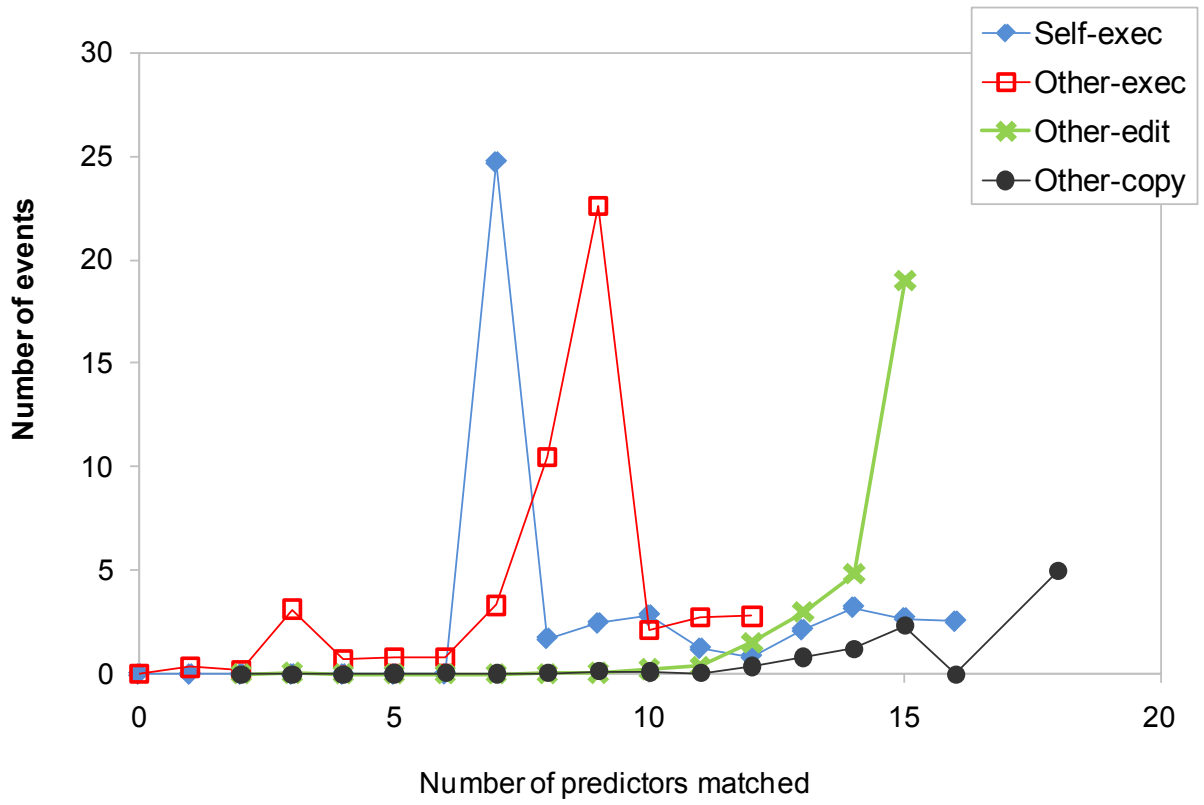


Fig. 9. Absolute level of reuse rose sharply with the number of matches for three out of four predictors; for self-exec, no clear trend was visible.

Table 1. Candidate script traits introduced above. A + (-) hypothesis Hyp indicates that we expected higher (lower) levels of reuse to correspond to the trait. Section 3.4 discusses Empirical Results.

Table 2. Quality measures of comparison models

Table 3. True positive (TP) at false positive (FP) of approximately 0.4, with variant methods for selecting each trait's threshold

Table 1. Candidate script traits introduced above. A + (-) hypothesis Hyp indicates that we expected higher (lower) levels of reuse to correspond to the trait. Section 3.4 discusses Empirical Results.

Trait			Empirical Results				
Catg	Name	Meaning	Hyp	Self Exec	Other Exec	Other Edit	Other Copy
Mass appeal	keywrđ_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 <i>nonus_urls</i> and <i>us_urls</i> 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 (<i>code_lines</i> + <i>comments</i>)	-	+			
	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	+		+		+++
Precond.	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	-		+++		

Table 2. Quality measures of comparison models

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

Table 3. TP at FP of approximately 0.4, with variant methods for selecting each trait's threshold

	Selecting thresholds based on...		
	Proportions	Means	Entropy
<i>Self-exec</i>	0.78	0.78	0.62
<i>Other-exec</i>	0.79	0.79	0.40
<i>Other-edit</i>	0.77	0.70	0.50
<i>Other-copy</i>	0.72	0.70	0.62