On the Evaluation of Recommender Systems with Recorded Interactions

Romain Robbes
REVEAL
Faculty of Informatics
University of Lugano
romain.robbes@lu.unisi.ch

Abstract

Recommender systems are Integrated Development Environment (IDE) extensions which assist developers in the task of coding. However, since they assist specific aspects of the general activity of programming, their impact is hard to assess. In previous work, we used with success an evaluation strategy using automated benchmarks to automatically and precisely evaluate several recommender systems, based on recording *and* replaying *developer interactions. In this paper, we highlight the challenges we expect to encounter while applying this approach to other recommender systems.*

1. Introduction

Developing software systems is a complex and difficult task relying on a large skill set, including program comprehension, creative thinking, problem solving and algorithmic skills. To assist developers as they program, Zeller envisions the future IDE as featuring a set of recommendation and assistance systems, each focusing on a type of commonly recurring problem [12]. We refer from now on to this type of tools simply as recommenders. Zeller's vision is already partially fulfilled, as state-of-the-art IDEs such as Eclipse already feature several recommenders, such as:

- Code completion, which assists the the seemingly simple task of typing program statements;
- Error correction, which, based on compilation warning and errors, proposes automated edit operations to address common classes of errors, such as importing missing namespaces (Eclipse's Quickfix);
- Change prediction, exemplified in tools such as eROSE, recommends entities to change alongside currently changed entities, based on the history of previous changes to the system [13];
- Navigation aids, such as Mylyn [4] or NavTracks [10], which based on what the programmer is currently looking at, recommend other entities to look at.

If all these recommenders are intuitively useful, having more definitive proof is difficult. The most common evaluation strategy that comes to mind is to perform a user study. A simple experimental protocol is to gather two groups of people, and ask them to perform a given development task, one group with the help of the recommender system, the other without. Each subject is either observed while they perform the task by the experimenter, or asked to fill a questionnaire after completing the task. From this data, the improvement that the recommender yields can be quantified. Many variants of this design exist, but they all share the following shortcomings:

Many variables: Each developer has a distinct experience with programming languages, tools, and a different way to solve problems. Some might type much faster than other. In short, there are many variables that could explain an observed variation in productivity. A larger number of subjects is needed to smooth out individual differences.

Subjectivity: Since coding relies on an array of skills, the developer or the observer themselves may have trouble discerning the impact of the recommender. How can a developer evaluate the accuracy of a code completion engine when he is focusing on finishing a non-trivial development task? Answering a questionnaire afterwards may hence yield vague or subjective answers that are hard to interpret.

Expensive: Recruiting a sufficient amount of people and carefully laying out the experimental protocol is time-consuming and potentially expensive. "Dry runs" of the experiment are necessary to weed out mistakes in the protocol. Finding the subjects for the experiment is also a difficult task as people value their time.

Hard to reproduce: User studies must have a very detailed protocol in order to be repeated. Lung *et al.* documented the hurdles they went through when they attempted to reproduce an experiment [5]. Reproducing user studies is hence hard and uncommon.

Of course, a user study is essential to ensure the good usability of a recommender, but these shortcomings mean that incremental improvements to a recommender are hard to evaluate this way. Indeed, the smaller the increment in productivity is, the larger the group of users need to be in order to rule out statistical error. The difficulty in reproducing a study and to compare the results of two studies is a further impediment to gradual optimizations. Such an optimization of the recommender is however essential to ensure that it is as accurate and useful to the developers as it could be. Without it, the algorithms and heuristics used by the recommender may be far from optimal.

There is however an evaluation strategy that is better suited to the comparative evaluation that is needed to optimize the algorithms at the heart of recommenders: Automated benchmarks [9]. An automated benchmark is a fully automatic process that takes as parameters a recommendation algorithm to evaluate and the data to evaluate it on, and computes the accuracy of the algorithm. This allows easy reproducibility and comparison between variants of the algorithm, making the technique ideal to optimize a recommender system.

In order to evaluate recommender systems in this way, the challenge lies in gathering the data necessary for the evaluation. In previous work, we introduced Change-Based Software Evolution (CBSE), which models with accuracy how software systems evolve over time [6]. CBSE relies in recording the changes as they happen in the IDE. Based on the change data we gathered on several systems, we applied a benchmarking strategy in order to evaluate several variants of two recommender systems, Code Completion and Change Prediction. The focus of this paper is hence to draw from this experience and outline the challenges that lies ahead in order to generalize this approach to other recommenders.

2 Benchmarking Recommenders

The approach we propose is based on the openness of state-of-the-art IDEs. IDEs such as Eclipse, Squeak or Visualworks allow one to easily monitor and record how the developer is using the IDE to incrementally develop a piece of software. Such an IDE issues all kinds of events (necessary for its internal architecture) in response to actions the developer perform. Examples of such events are navigation events indicating that the developer is looking at a given part of the system, tool usage event indicating that the developer is using the refactoring engine, the compiler or the debugger, and edition events describing how the developer changes the system, from keystroke events to higher-level events such as addition of entities.

If the IDE is open enough and allows access to these events to third-party extensions, one can *record* these events, and, ensuring that they are descriptive enough, *replay* them at will in an automated manner. If the information is accurate enough, such an approach allows one to effectively simulate –in an entirely automatic way– the interactions of the developer with the IDE as he is building the system.

Automation is key to allow the definition of interaction benchmarks as a methodology to repeatedly and accurately measure the accuracy of recommender systems. It allows one to inexpensively run several variants of the same recommendation algorithm and compare their performance with a precisely and objectively computed metric. This allows one to truly optimize the recommendation algorithm and make the recommender as accurate as possible.

Assuming that a recorded interaction history *H* is available, computing the accuracy *A* of a recommender *R* on the interaction history proceeds as shown by Algorithm 1 (*E* is the model of the system and the developer that the recommender uses).

```
Input: $H$, $R$
Output: $A$
foreach Interaction $I$ in $H$ do
    if $I$ is of interest to $R$ then
        Ask $R$ to predict $I$, given the environment $E$
        Compare $R$ to $I$ and update $A$
    end
    Replay $I$ in order to update $E$
end
```

Algorithm 1: Computing the accuracy of a recommender on an interaction history

Of course, Algorithm 1 assumes that the interaction history *H* exists. For this to be the case, one must follow the following steps:

Frame the problem of the evaluation of the recommender in terms that allow the automation. This implies isolating only the parts of the recommender that are relevant to the task, such as the central algorithm providing recommendations from unnecessary parts like the GUI.

Identify the information needed by the recommender to function properly, and from it, the interactions that need to be recorded to rebuild the information needed by the recommender.

Define the prediction format that the recommender uses and the kind of interaction triggering its evaluation.

Define the accuracy measure that will be computed. Depending on the types and format of the predictions, different measure will work, such as a single accuracy measure or both precision and recall.

Record interactions. Once the type of information needed is defined, one has to gather it by monitoring the activity of a large enough set of developers as they work for a long enough period of time, so that the set of interaction histories gathered can be deemed representative.

We now illustrate this process with the two examples in which we applied it successfully. In both cases, we reproduced and introduced several variants of each recommenders, and improved on the state of the art.

Example: Code Completion We used recorded interactions to measure the accuracy of several code completion engines [7]. Code completion is a recommender used to assist typing that presents to the user a list of words or function names that may correspond to the word the programmer is presently typing, saving her keystrokes.

To automate testing, we consider the completion engine only, that is the part of the recommender taking as input the prefix of a word, and returning a list of candidates matches. Since variants of the code completion engine rely on the state of the program and the recent changes to the system, the interactions we recorded were the changes made to build the system. Of these, the interactions of interest to the recommender were the changes that inserted new method calls in the system: The completion engine was asked to return a list of candidates which ideally contained the name of the method being inserted. The list was cut off at ten items, as a longer list of candidates was deemed too long to be read in its entirety by the programmer. To compute the accuracy, we measured the index of the correct match in the list, and rewarded correct matches that were in the first items, and for shorter prefixes (we tested each insertion of a method call by asking for the recommender's guess with a prefix of 2, 3 ... up to 8 letters). The data used in the benchmark were the recorded changes performed on 8 small to medium scale systems, totalling more than 3 years of development.

Example: Change Prediction Our evaluation of change prediction approaches [8] was very similar as it relied on the same recorded interactions, the changes to the system as they evolve. Change prediction attempts to predict the entities (classes or methods) that the programmer is going to change after the one he changes, in order to either remind him to change them (error prevention) or to provide easy access to them (navigation assistance).

The portion of the change predictor we tested was the algorithm that, given entities changed in the past, proposed a list of potential change-prone entities. The data recorded was the change sequence developers made when building programs. The entities of interest for which the change predictors was tested where the sequence of changed entities, filtering out repetitive changes to the same entities and changes originating from automated tools. The accuracy of the change predictor was defined as the similarity between a list of n change-prone entities returned by the predictor with the actual n next changing entities. The data set we used was very similar (it contained one additional change history from a Java program).

Note that a similar evaluation strategy was used by previous change prediction approaches, but based on SCM transactions, rather than recorded change sequences [3][13][11][2]. Using SCM transactions instead of recorded IDE interactions is more convenient, since a lot of data is already available, but less accurate, since the data is more coarse.

3 Challenges in Further Applications

We believe this approach is applicable to other kinds of recommenders and we expect the same benefits from its application. However, certain particularities of the approach mean that care must be taken in fulfilling the steps we described above. Indeed, recording the data is a costly and time-consuming operation. Hence the kind of data that has to be recorded must be carefully defined upfront. In the following we make a tentative list of recommender systems that we envision being evaluated in the same way, and highlight the needs for each of them.

Code Completion and Change Prediction: In our comparison of approaches, missing data prevented us to reproduce every approach we wished. The navigation information necessary for some approaches was missing. Further, a precise notion of task (*i. e.* the set of entities related to a task) was only approximated. This missing data is needed to further improve our results.

Task Detection: The missing notion of task context could be a recommendation by itself. We would like to annotate our change information with task information and experiment with several approaches to detect them.

Clone Detection: The presence of duplicated code in code bases is an established fact, and several tools exist to detect it. Based on our recorded change histories, we could annotate changes that introduce new clones in the system as interactions of interest.

Clone Evolution: A possible solution to the clone problem is to co-evolve clones when one of them changes [1]. The techniques proposed so far are based on simple string rewriting. An automated benchmark comparing the actual changes with the future changes could assess whether more complex techniques are needed.

Error prevention and correction: With the necessary annotations of the change data, tools such as Quickfix could be formally evaluated, and new heuristics fulfilling their shortcomings could be defined.

Based on the potential applications, we identified the following issues that are open to discussion:

Additional sources of information. An effort is needed to identify all the necessary sources of information to be recorded, beyond those that we already identified, changes and navigation information. Tool support should then be implemented to record this data.

Annotations of the interactions. Annotations are needed to mark the entities of interest for each recommender, such as clones, tasks, errors and the interactions causing and/or solving them. A systematic review of the recorded interaction is needed to annotate them, and tool support is needed to perform it efficiently. Such a review would also filter out interactions featuring unwanted behavior, such as cases where the developer was in the wrong track for a part of the session.

Recording more data. The amount of data we recorded so far is still small, and some of it is incomplete. To provide more significant result, an effort is needed to record much larger interaction histories.

Community involvement. Recording a large amount of data, implementing the necessary tools, and improving on the state of the arts of recommenders once the infrastructure is there is a significant effort for which we welcome members of the community. In particular, a shared effort to record development histories of student projects would be the most immediate way to gather a larger amount of data.

4 Conclusion

Recommenders are a growing part of a programmer's tool set, yet optimizing them remains a difficult problem. We presented a general approach to evaluate the performance of recommenders in a systematic way, allowing incremental optimization of a recommender's overall usefulness to developers. The approach is based on the record and replay of programmer interaction histories in order to repeatedly simulate the activity of a developer. We outlined some of the challenges that we need to overcome in order to adapt the approach to various kinds of recommenders, namely identifying the kind of information one needs to record, recording of a large and representative enough set of interaction histories, annotating the interaction history in order to emphasize the relevant interactions when needed, and the development of a common infrastructure allowing the sharing of the data and the easy dissemination of the results necessary to foster a community around recommenders [9].

References

[1] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th Interanational Conference on Software Engineering (ICSE 2007)*, pages 158–167, 2007.

[2] T. Girba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, Sept. 2004. IEEE Computer Society.

[3] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3):335–367, 2006.

[4] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proceedings of SIGSOFT FSE 2006*, pages 1–11, 2006.

[5] J. Lung, J. Aranda, S. M. Easterbrook, and G. V. Wilson. On the difficulty of replicating human subjects studies in software engineering. In Robby, editor, *ICSE*, pages 191–200. ACM, 2008.

[6] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, 2008.

[7] R. Robbes and M. Lanza. How program history can improve code completion. In *Proceedings of ASE 2008 (23rd ACM/IEEE International Conference on Automated Software Engineering)*, pages 317–326. ACM Press, 2008.

[8] R. Robbes, M. Lanza, and D. Pollet. A benchmark for change prediction. Technical Report 06, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, October 2008.

[9] S. E. Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *ICSE*, pages 74–83. IEEE Computer Society, 2003.

[10] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005)*, pages 325–335. IEEE Computer Society, sep 2005.

[11] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Transactions on Software Engineering*, 30(9):573–586, 2004.

[12] A. Zeller. The future of programming environments: Integration, synergy, and assistance. In *Proceedings of the 2nd Future of Software Engineering Conference (FOSE 2007)*, pages 316–325, 2007.

[13] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572. IEEE Computer Society, 2004.