Expert Recommendation with Usage Expertise

David Ma
University of Calgary
davma@ucalgary.ca

David Schuler
Saarland University
ds@cs.uni-sb.de

Thomas Zimmermann
Microsoft Research
tz@acm.org

Jonathan Sillito
University of Calgary
sillito@ucalgary.ca

Abstract

Global and distributed software development increases the need to find and connect developers with relevant expertise. Existing recommendation systems typically model expertise based on file changes (implementation expertise). While these approaches have shown success, they require a substantial recorded history of development for a project. Previously, we have proposed the concept of usage expertise, *i.e., expertise manifested through the act of calling (using) a method. In this paper, we assess the viability of this concept by evaluating expert recommendations for the ASPECTJ and ECLIPSE projects. We find that both usage and implementation expertise have comparable levels of accuracy, which suggests that usage expertise may be used as a substitute measure. We also find a notable overlap of method calls across both projects, which suggests that usage expertise can be leveraged to recommend experts from different projects and thus for projects with little or no history.*

1. Introduction

Current approaches for automated developer recommendation systems are rooted in variations of the *Line 10 Rule* to determine experts for files. The Line 10 Rule stems from a version control system that stored the commit author in line 10 of the log message. Through the act of changing a file, the developer is considered to have gained expertise for the said file. This type of expertise is also referred to as **implementation expertise** [1]; see Section 2 for more related work. While such approaches have shown promise, the implicit criteria for recommending a developer for a file is that there must be at least one commit by the developer for the file. Thus such systems cannot be applied to files or projects with no or little history.

In earlier work, we proposed **usage expertise** [9]; the accumulation of expertise by *calling* (using) methods. Simply by calling a method, a developer demonstrates that they at the very least know what the method does (without knowing implementation details). In this paper, we argue that developers also have an implicit understanding of the existing method calls surrounding the location of change. In other words, by adding a method call developers demonstrate usage expertise for the added method call *and* the surrounding **context**. We describe the set of heuristics that we use to infer expertise from developer activities in Section 3.

In our experiments on the ECLIPSE and ASPECTJ projects in Sections 4 and 5, we compare the accuracy of recommendations based on usage expertise (including the context) with recommendations based on implementation expertise. Usage expertise, with context, can recommend with similar accuracy as implementation expertise. Furthermore, we propose a way of leveraging usage expertise to recommend developers from projects that use the same external libraries/frameworks (*recommending across projects*). The implications of this are that not only can usage expertise produce cross-project recommendations but also, that it can possibly recommend for *projects* with no or little history.

2. Related Work

Researchers in the area of recommender systems have investigated how to recommend experts based on a mined history of development. Some approaches analyze source code changes as evidence of expertise and recommend based on the size, relation or frequency of contributions [2, 6]. Some approaches chose to aggregate source code changes with artifacts related to the development process. Such examples include bug reports [1], calls to tech support [5] or metadata for changes [7]. Other research has delved into expertise modeling based on artifacts other than source code. Such examples include developer-IDE interaction patterns [8] and vocabulary used within bug reports [4].

These approaches are specific to a given project. Thus not only is a substantial history of activity required, this history is not portable to other projects. By mining usage expertise instead, we get *project independent* expertise (e.g., for external libraries) transferable *across projects*. This could allow us to recommend for newcomers to projects or the *recommendation for code with little or no history*.

3. Quantifying Expertise

To make recommendations, we must first record developer activity in the form of an *expertise profile*. With an aggregated profile we can then apply *heuristics* to infer the degree of expertise developers have for a set of methods. This paper presents an abridged account of how expertise is modeled. For more details, we refer to a technical report [3].

Expertise Profiles

Developer activity is modeled with methods and method calls as units of expertise. Given a reconstructed CVS commit by a developer, we extract the *added or changed methods* by a developer and the *frequency* of changes into the developer's implementation profile. Similarly, *added method calls* and the *frequency* of calls are extracted into the developer's usage profile. We repeat this process for all commits.

Scoring Expertise

Using the *implementation profile* of a developer and a query in form of a set of methods, we infer the developer's aptitude for the query as follows:

Frequency of changes. Committing changes is evidence of expertise and by extension more changes implies more expertise. Thus the most qualified developer for a queried set of methods will be the developer with the highest total adds/changes for members of the set.

Recency of changes. Recent changes imply that a developer is still familiar with the details. Given that commit times are represented as timestamps, the developer with the most recent knowledge (most expertise) is the developer having the largest sum of timestamps.

Similarly, we can also infer aptitude with heuristics based on *usage profiles*. We propose four possible measures because we do not know yet which heuristic yields the "best" recommendations and to gain insight into what characterizes an effective measure.

Depth of usage. Each call for a method is quantified as a linear increase in expertise for the method. Thus for a set of methods the developer with the most expertise is the developer with the largest sum of calls.

Relative depth of usage. Frequency of calls by a single developer are normalized relative to frequency of calls by the entire developer population. Commonly used methods are weighed less than rarely called methods.

Breadth of usage. Here we consider developers to have expertise for a method with at least a single call. Thus the best developer(s) for a queried set of methods are the developers who have called the most members of the set at least once.

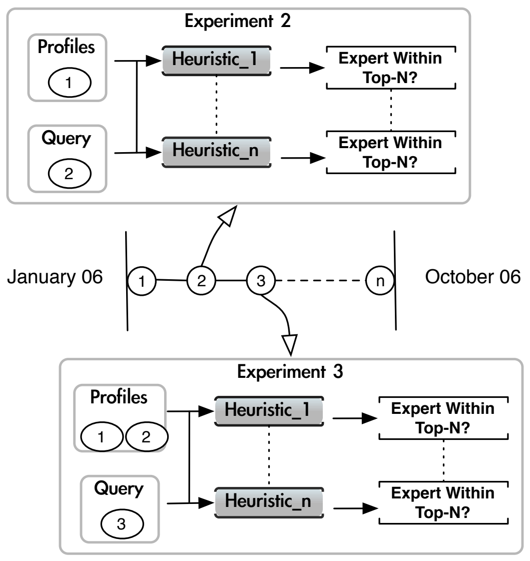

Figure 1. Experiment Overview

Relative breadth of usage. Again we consider developers calling methods at least once to have knowledge. However, calls to methods used by a large portion of developers is scored lower than methods called by only a few developers. This measure favors developers who have called the widest range of methods with an emphasis on rarely called methods.

4. Methodology

In our study we investigated the ECLIPSE and ASPECTJ projects over the period of January 2006 until October 2006. This period was selected according to three criteria. First, because of previous experience with this data. Second, we wanted sufficient volume of commits to avoid the potential for individual commits to skew data. Third, because expertise decays over time we wanted to limit the span of time.

Figure 1 depicts an overview of our experiments. Given a commit we train expertise profiles for each developer with data *prior* to the commit. Changes recovered from the commit form a set of methods (implementation query) and set of method calls (usage query). Profiles and queries serve as input to heuristics which produce an ordered list of developers with the most expertise. Expertise profiles are then incrementally updated using the changes recovered from the current commit and the above procedure is repeated again for all remaining commit operations. The accuracy of recommendations is then the percentage of times a "successful" recommendation appears within the Top-N results.

We now discuss how to form queries and how to define success for two different experiments: recommending within and across projects.

Recommending Within Projects

These experiments compare the precision of usage and implementation expertise in the context of providing recommendations for individual projects. To perform this comparison, we test with three different types of queries:

Implementation. The *methods added or changed* during a commit.

Usage. The *method calls added* during a commit

Usage with context. The *method calls added* during a commit combined with the method calls *added prior to the commit* in the same location (context).

Here a successful recommendation is when the expert (i.e., the actual commit author) appears within the Top-N recommendations. The underlying assumption is that the committed change set is evidence that the commit author had the expertise to do so. In some cases, we may recommend developers with sufficient expertise but who did not actually perform the change. Accounting for such situations would increase the hit rates. In other words, the hit rates reported should be consider as a lower bound.

Recommending Across Projects

These experiments explore the possibility of making cross-project recommendations by leveraging usage expertise. In other words, is it possible to recommend developers working in the ECLIPSE project for tasks on the ASPECTJ project (or vice versa)?

To test this we now consider commits from both projects. For any given commit we train usage profiles on data, belonging to either project, occurring prior to the commit. We limit our possible queries to only the *method calls added* during a commit (i.e., the **Usage** scenario from above).

We define a recommendation as successful when a developer from a different project is recommended within the Top-N recommendations. Note that this experiment is rather a demonstration of the feasibility of recommending developers across projects. We not yet certain whether theses recommendations are indeed “good” recommendations, this will be part of our future work.

5. Results

Implementation and usage expertise within projects.

Figure 2 illustrates our results for recommending within projects. On the left-most plot we observe favorable results when recommending for projects with small development teams (ASPECTJ). With only one recommendation (N = 1), 3 of 4 usage based measures perform close to the best implementation based heuristic. As N increases the difference quickly becomes negligible.

This trend, however, does not repeat for ECLIPSE project (middle plot). We speculate that the disparity in results could possibly be attributed to the scope of dependencies on external libraries. As a smaller and more focused project, ASPECTJ developers are likely to manipulate less code and by extension, a narrower set of method calls. Thus it is less likely that usage patterns for libraries overlaps which in turn, results in stronger evidence for the heuristics.

Context improves accuracy of usage expertise.

Given as few as the Top-2 recommendations we see that Change Frequency is roughly 75% accurate. Unsurprisingly, this means only a handful of developers contribute the bulk of contributions for most methods. The implications of this behavior can explain the marked improvement when using context (right-most plot of Figure 2). It is probable that developers understand the calls surrounding the location of change, given that they were directly responsible for adding the previous methods; evidence that querying with context is based on solid ground.

On different usage expertise heuristics.

Results do not clearly depict any one heuristic as superior. Thus we cannot yet conclude how to best leverage usage expertise for recommendation. Conversely, it may also be the case that perhaps inferring based on breadth or relative depth of expertise are indeed effective. What we can state is when considering the commit author as the only accurate expert, inferring with *Depth* yields lower hit rates. *Relative Depth* favors rarely called methods which may accurately model the behavior of commit authors (whereas *Depth* does not normalize weights). Also note that the commit author may not be the sole expert and thus these results should be considered a lower bound.

On Recommending Across Projects

Figure 3 paints a favorable picture regarding the prospects of recommending across projects. Intuitively, for the hit rate to be as high as 75%, there must be sufficient overlap in the calls to external libraries. By extension this means that recommendations are at the very least, possible. When rarely called methods are given more weight (*Relative Depth/Breadth*), we are more likely to recommend inter-project developers, given that the overlap of internal methods is likely to be low.

While we cannot claim for our experiment that the cross-project recommendations are in fact “good” recommendations, we demonstrated the feasibility of providing recommendations across projects. Assessing the quality of these recommendations is only possible with user studies; which we leave as future work.

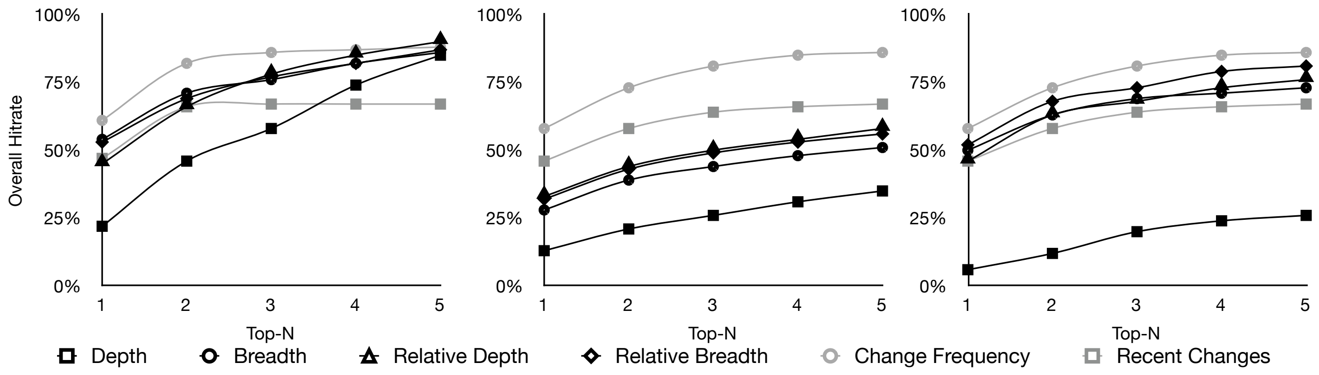

Figure 2. Recommendations within projects. Implementation (light) versus usage expertise (dark). Left: ASPECTJ implementation and usage expertise; middle: ECLIPSE implementation and usage expertise; right: ECLIPSE implementation and usage expertise (with context queries).

6. Consequences and Conclusions

In this paper we showed empirically that usage expertise produces recommendations with an accuracy comparable to implementation expertise. We also presented an approach to improve recommendations by also considering the implicit understanding of the surrounding context.

Our results also revealed that between the ECLIPSE and ASPECTJ project there is a substantial overlap of calls to external (or shared) libraries; alluding to the possibility of cross-project recommendations. While we could not assess the correctness of cross-project expert recommendations, we have demonstrated the possibility in this paper. We expect that the precision will be similar across projects to the precision within projects, for which usage expertise performs similar to implementation expertise.

To summarize, usage expertise can enhance traditional expert recommendation systems. In particular, they will allow recommendations for files and projects with little to no history and from unrelated parts of a project.

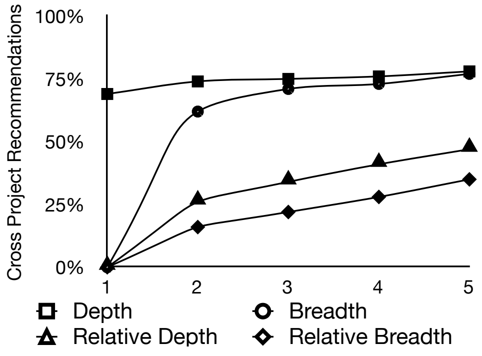

Figure 3. Recommending Across Projects

References

[1] J. Anvik and G. C. Murphy. Determining implementation expertise from bug reports. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.

[2] H. H. Kagdi, M. Hammad, and J. I. Maletic. Who can help me with this source code change? In *ICSM '08: Proceedings of the International Conference on Software Maintenance*, pages 157–166, 2008.

[3] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito. Expertise recommendation with usage expertise. Technical report, Department of Computer Science, University of Calgary, July 2009. https://dspace.ucalgary.ca/.

[4] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 131–140, May 2009.

[5] D. W. Mcdonald and M. S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, pages 231–240. ACM Press, 2000.

[6] S. Minto and G. C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of Fourth International Workshop on Mining Software Repositories*, 2007.

[7] A. Mockus and J. D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 503–512, 2002.

[8] S. Rastkar and G. C. Murphy. On what basis to recommend: Changesets or interactions? In *MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 155–158, May 2009.

[9] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the Fifth International Working Conference on Mining Software Repositories*, May 2008.