Expert Search on Code Repositories

Elben Shira and Matthew Lease
Department of Computer Science
University of Texas at Austin
eshira@cs.utexas.edu, ml@ischool.utexas.edu

Abstract

We consider a new expert search task: given a code base, its history, and a block of code as a query, who are experts to consult if we have a question about this block of code? We discuss conditions under which such search would be useful and methodological implications for assessing expertise in this scenario. We then propose several ranking functions and describe a pilot evaluation on an active code repository.

1 Introduction

Software projects often contain large amounts of code spread among many developers, with different developers being experts on different parts of the project. Often it is not well known which parts of the project a given developer is most familiar with, and it may further change over time as the size of the code base and development team grows. Open source projects exemplify such conditions.

As typical in software development, a developer may have questions about a code block he is not familiar with, and documentation is often lacking. Thus he may wish to consult an expert who is more familiar with the code block than he is. But who are the experts for this code block? How much expertise is required may also vary depending on whether the developer wishes to merely use the code or to modify it. By reviewing the development team's history of revisions to the code block in source control, we may ascribe differing levels of expertise to different developers. For example, we may distinguish a developer who has (a) used the code block before, (b) made simple changes to it, (c) made significant revisions, (d) originally authored it, etc.

We see this as an expert search task [3]: can we rank developers by their expertise for a given code block as a function of source control history? This reflects another interesting avenue for IR methodology to impact how software engineering is practiced today [1]. After defining the task more formally, we discuss implications for expertise assessment, propose ranking functions, and present a pilot evaluation.

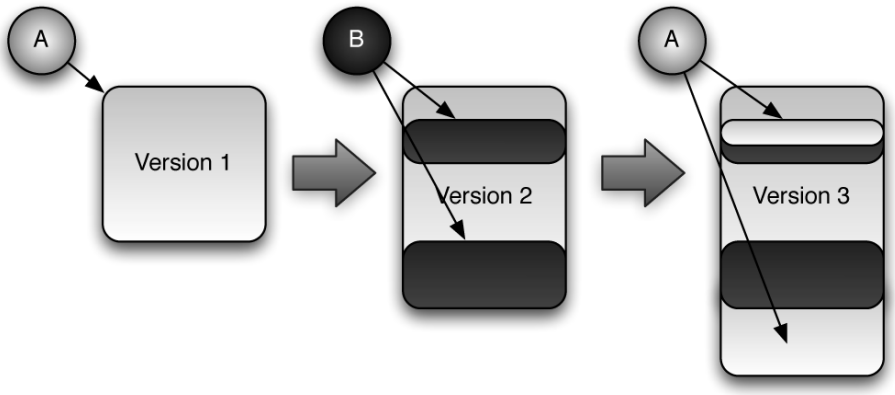

Figure 1: A version control example.

2 Version Control Systems

Version control systems track the changes that developers make to code in a project. Some version control systems such as SVN operate by saving the changes, known as deltas, between subsequent commits. Rolling back and forth between commits involve replaying these deltas. Other version control systems such as Git save the actual blobs of data, not the deltas. But for our purposes, we will consider the delta model, even when using Git.

An example of a file being version controlled is shown on Figure 1 on page 2. Here, we see two authors, A and B, modifying a file. Author A first creates the file. Author B then modifies and adds to the file. And finally, author A makes more changes.

We us version control systems to recall the history of the code. This history is useful because it tells us who edited what.

Most version control systems track changes on a per-file basis. Because the filename metadata is known, it is natural to consider the file as a unit of code. Though version control systems track line changes, we cannot at the moment infer what logical unit (e.g. a function, class, or module) an arbitrary line of code belongs to.

3 Features

To infer expertise from the code base, we consider features based on size, recency, and connectivity of the code base. We focus on the case of developer expertise being a function of that developer modifying the code (e.g. writing a function) rather than using the code (e.g. calling a function). A **block**, denoted b (and b_0 for the empty block), is lines of code that will be submitted into the IR system. A block may span across multiple logical units of code, like a function, class, or file. Because of limitations in today's version control systems, we assume that a

block is a file. A **query** is a tuple containing a block and other metadata such as time.

- $Lines(b) \Rightarrow Integer$. Number of lines in block b.
- $\Delta(b_i, b_j) \Rightarrow Block$. The difference between blocks b_i and b_j. Note that if there are lines in b_j that are not in b_i, then the resulting block will have "negative" lines.
- $History(b) \Rightarrow [Block]$. The history of b: $[b_0, b_1, \ldots, b_n]$, where $b = b_n$. Ordered by contribution time.
- $Contribution(a, b) \Rightarrow Integer$. Number of lines author a contributed to b.
- $Aging_e(t) = max(e^{-\lambda t}, 0)$. The exponential decay function, where λ is the decay constant and t is the time in days.

Other functions are built on top of the basic functions:

- $Lines(b_i, b_j) = Lines(\Delta(b_i, b_j))$.
- $AllLines(b) = \sum_{i=1}^{|History(b)|} Lines(b_{i-1}, b_i)$. The number of lines contributed to b over all contributions.
- $Contribution(a, b_i, b_j) = Contribution(a, \Delta(b_i, b_j))$.
- $AllContributions(a, b) = \sum_{i=1}^{|History(b)|} Contribution(a, \Delta(b_{i-1}, b_i))$. The total number of lines contributed by a to all contributions to b.
- $AllContributionsOverTime(a, b) = \sum_{i=1}^{|History(b)|} Aging_e(t) \cdot Contribution(a, \Delta(b_{i-1}, b_i))$. The total number of lines contributed by a to all contributions to b over time t. The unit for time is days.

4 Scoring Functions

Given a block b, author a, and scoring function F, we infer an expertise score $Score_F(a, b)$. The more likely a is an expert on b, the greater the value of $Score_F(a, b)$. $Score_{LC}$ considers only the latest contributions of a block:

$$Score_{LC}(a, b) = \frac{Contribution(a, b)}{Lines(b)} \tag{1}$$

$Score_{AC}$ uses all the contributions the author has done on this block.

$$Score_{AC}(a, b) = \frac{AllContributions(a, b)}{AllLines(b)} \tag{2}$$

$Score_{ACOT}$ is similar to $Score_{AC}$, but we use the intuition that a person loses expertise over time. That is, contributions years ago might not be as important

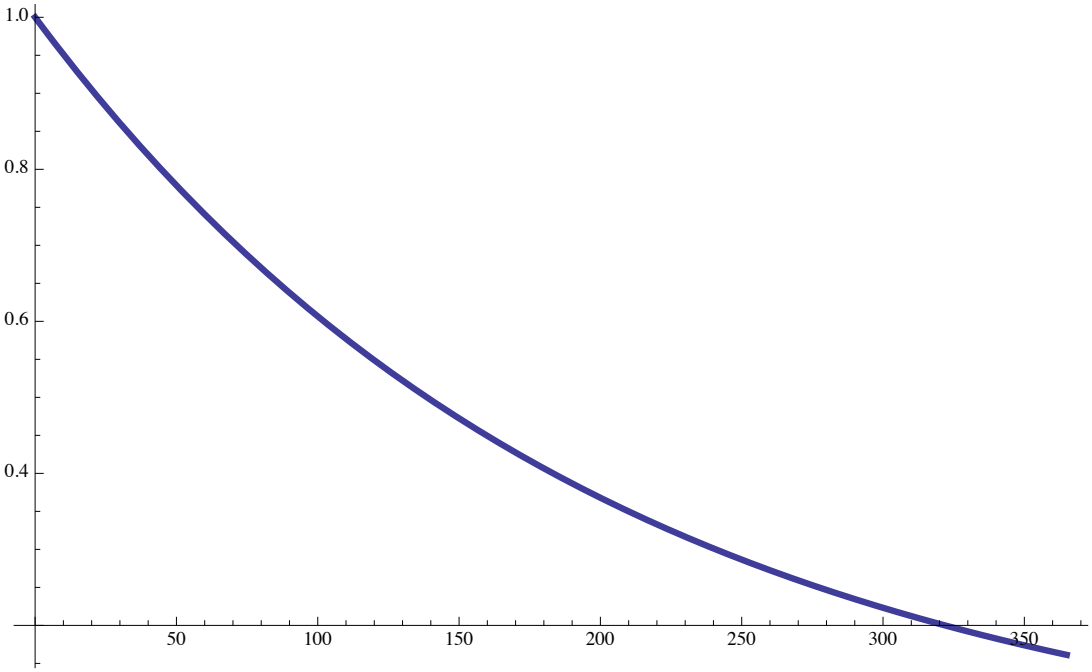

Figure 2: Decay function with $\lambda = 0.005$.

as contributions yesterday. We can capture the temporal dimension by using the decay functions $Aging_e$.

$$Score_{ACOT}(a, b) = \frac{AllContributionsOverTime(a, b)}{AllLines(b)} \tag{3}$$

Where $AllContributionsOverTime$ is defined as:

$$AllContributionsOverTime(a, b) = \sum_{i=1}^{|History(b)|} Aging_e(t) \cdot Contribution(a, \Delta(b_{i-1}, b_i))$$

As a default, we set $\lambda = 0.005$ for $Aging_e$. This decay function is shown on Figure 2 on page 4.

5 Evaluation

We can evaluate our IR system by comparing our ranked lists with relevance judgements produced by human evaluators. That is, we give queries to the developers of a code repository and they give us the relevant experts for each query.

If we consider these human queries as the ground truth, then our methodology is similar to the TREC-styled method of evaluation. Thus, often-used

	Relevant			
	A	B	C	D
Judge A	♡	♡	♡	
Judge B		♡	♡	
Judge C	♡	♡	♡	♡

Table 1: Relevance judgements for File 1.

evaluations such as MAP (mean average precision) and P@n (precision at rank n) could be used to compare the human evaluations to the system evaluations. We could use $Score_{LC}$ as our baseline (since people often use this method) to compare $Score_{AC}$ and $Score_{ACOT}$.

But from our experiments, we realized that our human relevance judgements may not be a suitable ground truth. TREC-styled evaluations deploy domain experts to mark documents as relevant. While some there are some criticisms and retorts [4], this style of evaluation is widely accepted. Our relevance judgements, however, depend not a different kind of expertise; we depend on the judges' memories of who contributed to what. Even in a medium-sized project spanning a couple of months, asking humans to recall their team member's contributions is unlikely to produce correct judgements. Thus, until further work, we must break away from traditional TREC-styled evaluations. Instead, we present and discuss our findings without comparisons to a ground truth.

We evaluate ranking functions on a source-controlled repository containing about 2,000 lines of Python code written by five developers. Our implementation is publicly available in a library called Carnival[1]. We run our experiments assuming a fixed date of November 24, 2010. The last commit to the repository was September 28, 2010. Dates are significant because of the time decay function in $Score_{ACOT}$. Our pilot evaluation analyzes two code block queries: File 1 and File 2. For each file, we asked three developers to make a binary assessment of their own expertise, as well as that of two other developers. Judgments are shown in Table 1 and Table 2. We see in File 1, for example, that B judged B and C as experts, but not A.

Table 1 on page 5 shows relevance judgements for File 1 by three authors on three authors. For example, author A judged authors A, B, and C as relevant. Author B judged authors B and C as relevant.

Now consider Figure 3 on page 6[2]. We find that author A ranked first by $Score_{LC}$, but author C is ranked first by $Score_{AC}$ and $Score_{ACOT}$. Notice that author B did not judge author A to be relevant. Are our rankings wrong or the human judgements wrong? We think it is the latter. Notice that author A considered themselves to be relevant. This probably means that author A was truly relevant, but author B did not remember or know of author A's contributions to File 1.

[1]Carnival is open-sourced under the MIT license and can be downloaded at http://github.com/eshira/carnival.

[2]The unknown author is due to commits on a machine not owned by a single developer.

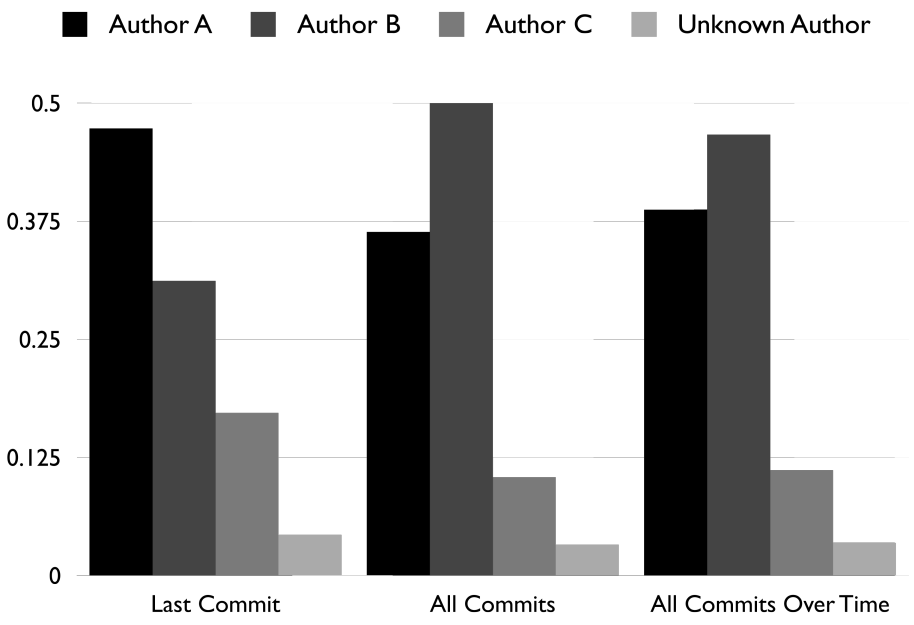

Figure 3: Scores for a File 1.

	Relevant		
	A	B	C
Judge A		♡	♡
Judge B		♡	
Judge C	♡	♡	♡

Table 2: Relevance judgements for File 2.

Also note that if we were to only look at the edits of the last commit (which easily done in git via a "git blame file1"), author A would be considered the resident expert. But using $Score_{AC}$ and $Score_{ACOT}$, we find that author B has contributed more to the history of the code than author A has. Our scoring functions reveal this hidden knowledge.

File 2's relevance judgements are on Table 2 on page 6. Note that while author A considered author C as relevant, author B did not. The rankings (Figure 4 on page 7) show that author C did not contribute a single line of code to File 2. Why did author A mark author C as relevant? It turns out author C is relevant for another file in the project, a file containing unit tests for File 2. This connection, however, is not available with our localized scoring functions.

And note that author A made small contributions to File 2, but no one marked author A as relevant, not even author A. This suggests a threshold for relevance.

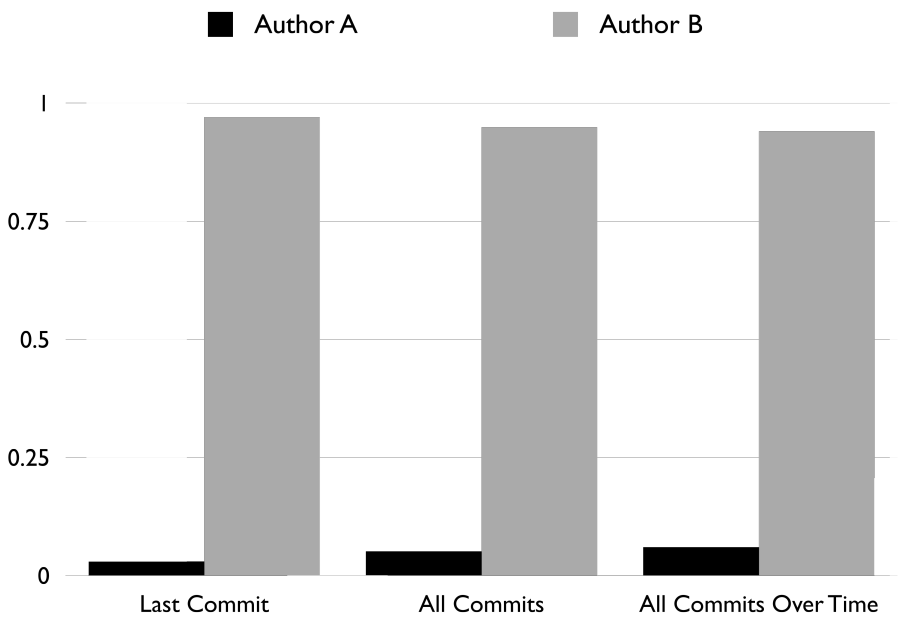

Figure 4: Scores for File 2.

6 Dicussion

Our pilot evaluation only considered a code block at the granularity of an entire file. This reflects a limitation of most current version control systems which are not effective at tracking changes at granularity smaller than a file. However, there has been work tracking movements of logical units of code (e.g. classes, functions) [2], and our plan is to build on such finer granularity tracking in future work.

Another limitation is that the current ranking functions cannot make connections between blocks. As we saw with File 2, if an author writes many unit tests for File m but has not changed m itself, Carnival does detect this expertise for m. There are likely other such connections to address.

The proposed ranking functions infer expertise only by additive code contributions. But consider the case of a senior developer refactoring a novice's code to be half the size. The senior developer should be ranked higher but will not as currently implemented. Thus we would like to factor in non-additive modifications and a prior for each developer based on all of their code revisions recorded in the repository.

While we evaluated Carnival by comparing our rankings with expertise judgments produced by human evaluators, the judgments obtained were not as reliable as we had hoped because developers had difficulty recalling their team members' contributions to the code base. Consequently, we plan to adopt a

different approach for future expertise judging. By asking developers to mark which blocks they are experts for, instead of asking them to mark expertise of other developers for a certain block, we believe our judgments will provide a firmer foundation for subsequent evaluation.

References

[1] R. Hoffmann, J. Fogarty, and D.S. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, 2007.

[2] M Kim and D Notkin. Discovering and representing systematic code changes. *Proc. Software Engineering*, 2009.

[3] C. Macdonald and I. Ounis. Using relevance feedback in expert search. *Advances in Information Retrieval*, 2007.

[4] E Voorhees. The philosophy of information retrieval evaluation. *Evaluation of cross-language information retrieval systems*, 2002.