Modeling Programmer Navigation:

A head-to-head empirical evaluation of predictive models

David Piorkowski[1], Scott D. Fleming[1], Christopher Scaffidi[1], Liza John[3],
Christopher Bogart[1], Bonnie E. John[2,3], Margaret Burnett[1], Rachel Bellamy[2]

[1]Oregon State University
Corvallis, Oregon 97331
{piorkoda,sdf,bogart,cscaffid,burnett}
@eecs.oregonstate.edu

[2]IBM T.J. Watson Research
Hawthorne, New York 10532
{bejohn,rachel}@us.ibm.com

[3]Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
liza.john@gmail.com

Abstract—**Software developers frequently need to perform code maintenance tasks, but doing so requires time-consuming navigation through code. A variety of tools are aimed at easing this navigation by using models to identify places in the code that a developer might want to visit, and then providing shortcuts so that the developer can quickly navigate to those locations. To date, however, only a few of these models have been compared head-to-head to assess their predictive accuracy. In particular, we do not know which models are most accurate overall, which are accurate only in certain circumstances, and whether combining models could enhance accuracy. Therefore, we have conducted an empirical study to evaluate the accuracy of a broad range of models for predicting many different kinds of code navigations in sample maintenance tasks. Overall, we found that models tended to perform best if they took into account how recently a developer has viewed pieces of the code, and if models took into account the spatial proximity of methods within the code. We also found that the accuracy of single-factor models can be improved by combining factors, using a spreading-activation based approach, to produce multi-factor models. Based on these results, we offer concrete guidance about how these models could be used to provide enhanced software development tools that ease the difficulty of navigating through code.**

Keywords-software maintenance, debugging, program investigation, program navigation, information foraging

I. INTRODUCTION

An essential aspect of software engineering is fixing bugs and adding features. When developers perform such maintenance tasks on unfamiliar code (or code that they no longer remember well), they need to gather a great deal of information before they can begin to edit the code. For example, they need to find answers to questions about where certain features are implemented and how different pieces of the code relate to one another [15]. During this search for information, developers gradually build up and mentally test hypotheses about how the code works and how to modify it in order to complete the maintenance task [8]. Unfortunately, in gathering this information, developers spend an excessive amount of time navigating through code. In one laboratory study, developers engaged in maintenance tasks spent up to 35% of their time merely navigating through the code [8].

Various tools have been provided to accelerate these time-consuming navigations [2][4][13][14][17][18][20][21]. Internally, each tool uses a model to identify code that the programmer is likely to need to visit, and the tool then provides a window of navigational shortcuts to that code. These models are typically based on a single factor. Some of them are based on mining logs of how people have edited or navigated through the code in the past [4][13][17][20][21], whereas others make predictions based on textual similarity between methods, call-invocation relationships between methods, inheritance between classes, and other structural relationships within code [2][14][18].

One problem is that we do not yet know which of these models most accurately predicts programmer navigations. Moreover, we do not yet know if any models work well in general but work poorly for predicting certain kinds of navigations. Comparing the accuracy of these models is important because a tool is unlikely to be useful if its underlying predictive model is inaccurate. Conducting such an evaluation requires testing the models head to head on navigation data from sample maintenance tasks. To date, this has only been done with a small subset of the models (specifically, those that can predict when a developer will revisit methods that were previously viewed) [11].

The contribution of the current paper is to compare how well a broad range of models can predict different kinds of navigation actions. We investigate two particular questions:

RQ1: How accurately do different models predict programmer navigations, overall? We are especially interested in how well accuracy can be increased by combining single-factor models into multi-factor models.

RQ2: How does each model's accuracy vary depending on the operationalization of programmer navigation? We are particularly interested in whether accuracy differs between an operationalization from the literature (e.g., [11]) that is based on where the programmer clicks and a new operationalization based on what code is in the programmer's view.

II. BACKGROUND

Empirical studies have revealed many cases where developers navigate through code. During maintenance tasks in a laboratory study, developers spent 35% of their time looking through code to understand it and to plan changes [8]. Even after learning about code, developers often need to relearn the same information through repeated navigations. Interruptions are a major cause for these revisits. In field studies, developers faced major interruptions approximately once per hour; 40% of interruptions were not immediately followed by a return to the original task, and restarting a task typically required going back to locations in code to recover mental state [3][11]. Overall, code navigation is essential to finding defects, fixing defects, implementing features, porting code, documenting code, extracting reusable code, and virtually every software engineering activity.

Certain code navigations were particularly common, especially in maintenance tasks. At the start of a task in laboratory and field studies, a developer usually searched for a location in the code that could serve as an initial focus point [15]. These navigations often took the form of keyword-based queries in the IDE to find code textually related to a bug report or feature enhancement request. Over the course of a task, each developer typically began to ask questions about how pieces of code were related, such as through method-invocation [4][15]. Following up on these questions broadened the investigation's scope to include more code. Eventually, each developer learned enough to proceed with making edits. During tasks, developers frequently scrolled between adjacent locations in code [11][13]. Another frequent navigation was to revisit previously read code, whether later in the day [11], later in a particular task [16], or even within seconds of an earlier visit (i.e., ―jitter") [17]. Revisits were most frequent toward the end of each task [16].

Various tools seek to ease these navigations by providing shortcuts to places in the code that the developer might go (e.g., [2][4][13][14][17][18][20][21]). Several tools provide shortcuts to locations that other developers historically have visited after the current location [4][13][17]. Other tools provide shortcuts to code that has historically been modified at the same time as the current code [19][21]. Still other tools provide shortcuts to code or other artifacts based on textual similarity, method-invocation, or other structural relationships [2][14][18].

Parnin and Gorg noted that the input for generating these shortcuts is the developer's *current* location [11]. In response, they proposed to use the developer's recent *history* of navigations to help filter, reweight, or otherwise refine the list of shortcuts offered by tools [11]. For example, they argued that if a tool could use the developer's navigation history to identify places where he or she is likely to go next, then the list of shortcuts (from any of the tools above) could be filtered to only include those that the developer is most likely to want next. In addition, if a tool could identify places where the developer is likely to go next, then the tool could provide a shortcut list directly (rather than simply filtering another tool's list). Achieving such enhancements therefore requires a model that can *use the developer's navigation history to predict the next navigation*. Motivated by the Mylar tool (which simply shows shortcuts to recently visited code) [7], Parnin and Gorg evaluated four predictive models. In the current paper, we take Parnin and Gorg's approach further and *evaluate how well these and a broader range of models can predict the next navigation based on the developer's navigation history*.

III. Models of Programmer Navigation

Expanding on Parnin and Gorg's evaluation, we evaluated a variety of predictive models of programmer navigation. Each of the models uses a single factor in making its predictions (thus we refer to these models as *single-factor models*). We focused on factors underlying promising prior approaches. For instance, recency and frequency of navigation were Parnin and Gorg's most predictive factors [11]; working set is a widely used concept and is emerging as a foundation for important IDEs (e.g., [7]); and factors of textual similarity and code structure are also widely used (e.g., [2][10]).

Each of our models represents a developer's Java programming session in Eclipse as a sequence of navigations to methods. More formally, the developer's navigation history H for a particular programming session is a sequence of methods $(m_1, m_2, ..., m_n)$ such that for all m_i in H, $m_i \neq m_{i+1}$.

Given the navigation history up to a certain point in a session $H_j = (m_1, m_2, \ldots, m_j)$, each model attempts to predict the next method m_{j+1}. Leading up to the current point in the session, the programmer will have opened one or more source files in the Eclipse editor. The set M_j approximates the set of methods that the programmer currently knows to exist. It comprises all methods that are defined or referenced in the previously opened source files (including files that were open at one time, but have since been closed). Because the developer must open the source file containing a method before navigating to that method, the methods in H_j constitute a subset of the methods in M_j.

In attempting to predict m_{j+1}, the models rank the methods in $M_j - \{m_j\}$ from least likely to most likely. To produce this ranking, each model creates a mapping A_j from each method in $M_j - \{m_j\}$ to an activation value (a real number) such that it is more likely (according to the model) that the developer will navigate to methods associated with greater activation. The models use this activation function to produce a ranking function R_j such that R_j is A_j with rank-transformed activation values (from highest to lowest). Activations of equal value are replaced by the average of the ranks involved. The models vary primarily in how they calculate A_j.

The *Recency model* assigns higher activation values to methods that the programmer visited more recently. Formally, the model defines A_j such that for each method m in $M_j - \{m_j\}$, if m has been visited previously by the programmer, then $A_j(m)$=the max sequence number for m in H_j; else, $A_j(m)$=0.

The *Working Set model* is a variant of the Recency model that is parameterized by a history length Δ, and assigns an activation of 1 to the last Δ methods visited and activation of 0 to all other methods. Formally, for all m in $M_j - \{m_j\}$ such that m is in $\{m_{j-1}, m_{j-2}, ..., m_{j-\Delta}\}$, $A_j(m)$=1; else, $A_j(m)$=0.

The *Frequency model* assigns higher activation values to the methods that the programmer has visited the most times in the past. More formally, for all methods m in $M_j - \{m_j\}$, $A_j(m)$ is the number of occurrences of m in H_j.

The *Bug Report Similarity model* assigns higher activation values to methods that are textually similar to a bug report. More formally, for all methods m in $M_j - \{m_j\}$, $A_j(m)$ is the *tf-idf* [1] weight for the words of the bug report with respect to the text of m. In this computation of tf-idf, the corpus consists of words taken from the set of methods M_j, including the current method m_j. *Stop words*, which are short common words like ―the," are removed prior to the computation. Additionally, *camelCase* words are broken up to enable textual matching on their component words.

The remaining single-factor models incorporate a notion of the cost of navigating between particular methods, and assign higher activations to methods that cost less to get to from the current method. Each such model maintains a graph G_j such that each method in M_j maps to a different vertex of G_j. Most of the models vary only in how edges in the graph are defined. Given the graph for a particular model, for all m in $M_j - \{m_j\}$, $A_j(m)$ for that model subtracts from $|M_j|$ the length of the shortest path from m to the current method m_j.

The *Within-File Distance model* constructs G_j such that edges connect methods that are textually adjacent in a source file. More formally, for each method m in M_j, an undirected edge in G_j connects m to the methods contiguous to m (i.e., immediately before and after m in the source file).

The *Forward Call Depth model* constructs G_j such that each method m is connected by a directed edge to each method called in the body of m. More formally, for all pairs of methods m_a, m_b in M_j, G_j contains a directed edge from m_a to m_b if and only if the method text of m_a contains a call to m_b.

The *Undirected Call Depth model* is identical to the directed version, except the directed edges in G_j are replaced with undirected edges.

The *Source Topology model* constructs G_j as a *source topology graph* that, in addition to method vertices, includes vertices for projects, packages, classes, interfaces, and variables as well. The source topology graph includes edges for ―has a" relationships between these elements (e.g., class inheritance), for ―calls a" relationships, and for within-file adjacency relationships. More formally, the set of classes C_j includes every class or interface referenced in a file that the programmer has opened so far. Similarly, the set of variables V_j, and the set of packages P_j, include every variable and package, respectively, referenced in a file so far. For every element e in $M_j \cup C_j \cup V_j \cup P_j$, the source topology graph contains a unique vertex that maps to e. The graph contains an edge between two elements e_a and e_b if e_a calls e_b, e_a has e_b, or e_a and e_b are both methods and are adjacent in a source file.

IV. Observational Method

We observed a programmer debugging two different open-source software applications in two separate sessions. We opted to collect rich, in-depth data (hours of audio and video) from this one participant as opposed to sparser data from a group. Observing a single participant limits the generality of our results, but since even a single participant generates numerous behaviors to be encoded and analyzed in detail, this is a common approach in understanding complex human behavior.

The participant was a senior-level undergraduate student majoring in computer science. He had 3 years total of programming experience (with an emphasis on Java), had 1.5 years of experience programming as an undergraduate research assistant, and was a skilled user of Eclipse. He had never seen the bugs or source code he was asked to modify.

We recorded audio of what was said, video of the participant, and screen-capture video. Additionally, we configured the participant's Eclipse environment to log actions and the contents of opened files with the PFIG plugin [10].

A. jEdit Session

For the first session, the participant's task was to fix a bug in jEdit, a text editor for programmers. jEdit is a relatively mature software project, comprising 98,652 non-comment lines of Java code. The bug (Figure 1) was still open (not yet fixed by jEdit maintainers). We did not explicitly use the think-aloud method for this session, but our data collection did include audio and video of anything the participant happened to say.

BUG: Problem with character-offset counter.

In the lower left corner of the jEdit window, there are two counters that describe the position of the text cursor. The first counter gives the number of the line that cursor is on. The second counter gives the character offset into the line.

The character-offset counter is broken. When the cursor is at the beginning of a line (i.e., before the first character in the line), jEdit shows the offset as 1. However, the offset should begin counting from 0. Thus, when the cursor is at the end of the line, it will display the number of characters in the line rather than the number of characters plus 1.

Figure 1. Bug report text for the jEdit task.

Prior to the task, the jEdit source code was loaded into an Eclipse environment, the bug report was provided on paper, and the participant was asked to fix the bug using whatever procedure he would normally employ, except directly asking another person (e.g., he had access to the internet). He successfully fixed the bug in 29 minutes.

B. Memoranda Session

For the second session, the participant worked on a bug in Memoranda, a diary manager and tool for scheduling personal projects. The program comprised 13,906 non-comment lines of Java code. This time, the bug (Figure 2) was a closed bug (1108171), so we could obtain the developers' patch for the bug to compare to our participant's fix.

BUG: Note lost when switching projects - ID: 1108171
Details:
When a note A (date A) for project A is active, switch project. Note B (date A) in project B is totally replaced by note A and note B is LOST forever!
note B is the first note of project B on date A.
Additional Comments:
This only happens if note A and note B are of the same date

Figure 2. Bug report text for the Memoranda task.

In this session, we used the think-aloud method [6] to elicit a verbal protocol from the participant. Regarding the possibility of effects on our measures, numerous studies have shown that instructing participants to ―verbalize their inner dialogue" (as opposed to asking them to ―explain" what they are doing) elicits comparable performance to when the participants are not thinking aloud [5].

The Memoranda source code containing the bug was loaded into an Eclipse environment, the bug report was provided on paper, and the participant was instructed to fix the bug using whatever procedure he would normally employ, except asking another person or, since the bug had been closed, accessing the Memoranda project's bug tracker and SVN logs. He successfully fixed the bug in just under 1 hour and 25 minutes. However, his fix introduced a new defect into the code.

V. Analysis Method

A. Coding Navigations

We used both an automatic process and a manual process to encode navigations to methods made by the participant. We also coded navigations according to their context in Eclipse (Table I).

A *click-based navigation* is a change in text cursor position from one method to another method caused by the developer clicking a button or clicking in the Java Editor. Our PFIG Eclipse extension logged both the position of the text cursor when the participant clicked within source code files and the textual content of that source code. As with other IDEs, Eclipse's affordances can influence developer navigations (e.g., closing a tab jumps the cursor to another open file). Since modern IDEs were our context of interest, we expected such behaviors and did not attempt to control for them.

In addition to clicking, the participant often navigated to methods without explicitly clicking in the source code. For example, he used the mouse wheel to scroll down through a long file, pausing to read methods. We extracted these actions manually using video annotation software. We first imported the PFIG logs into the video annotation software and anchored them to coincide with observable clicks in the video. We then annotated each action, with four attributes: action (left click, right click, typing a key on the keyboard, etc.), target (what button or word was clicked on, etc.), target location (in which window or pane the target was located, e.g., Package Explorer), and destination (which location was active after the action). This procedure resulted in an annotation for every user action (about 2500 hand annotations in roughly 2 hours of video) and had the side-effect of verifying the automatically recorded click-based navigations.

We used these annotations to code *view-based navigations*, which are user actions (involving any mechanism) that cause a new method to appear on screen. Specifically, we coded a view-based navigation as arriving at method *m* if: (1) the definition of *m* (including Javadoc comments) was in the vertical center of a visible source code tab; or (2) the entire definition of *m* was visible on the screen; or (3) the participant talked about *m* and the method was visible on the screen; or (4) the participant clicked in *m*. If multiple methods were visible when a new source code file was opened or a tab was switched to, we coded navigations to all fully visible methods in order from top to bottom.

TABLE I. Eclipse contexts and associated navigation actions.

Context	Navigation Actions
Debug view	Run to breakpoint, step over, step into
Java Editor tabs	Tab select, tab close
Java Editor	Click in method, scroll, Open Call Hierarchy
Package Explorer view	Open element
Call Hierarchy view	Open element
Find Utility	Find next
Java Outline view	Open element

B. Evaluating predictions

We evaluated a model's predictions by assessing whether it included the correct method (i.e., got a ―hit") in its top-*N* predictions. If a model produced more than *N* predictions with rank $\leq N$, we ignored the predictions with the lowest rank. Formally, given the correct method *m*, $R_j(m) \leq N$, and the number *T* of methods tied at rank $R_j(m)$ in R_j, floor(T/2)+floor($R_j(m)$) $\leq N$ indicated a hit.

VI. Results: Single-Factor Models

We used both click-based and view-based navigations to assess the accuracy of each single-factor model in predicting programmer navigation during debugging and to understand what accounted for the accuracy of the various models.

A. Analysis of Click-Based Navigation

To evaluate the models' predictive accuracy, we counted the number of times that the top-*N* ranked methods returned by a model included the method to which the programmer actually navigated next. We tested values of *N* from 1 to 20 (a reasonable size for a shortcut list in a development environment). Figure 3 aggregates the results for each model from the jEdit and Memoranda sessions.

Looking at only the top-ranked method predicted by each model (i.e., *N*=1), no single-factor model was particularly accurate (max hit ratio of 4.1%). However, with *N*=3, the hit ratio for Recency and Working Set rose to above 20%, and with *N*=10, these models had hit ratios near or above 50%.

The low accuracy of Bug Report Similarity is rather surprising because a prior study [9] found that during debugging programmers tended to navigate to methods with similar text to the bug report. The cause of the discrepancy remains an open question; for instance, it may arise from differences in the wording of the bug.

To better understand why some models were more accurate than others, we analyzed the individual navigation actions that the participant performed and the Eclipse interface contexts (i.e., distinct elements or Perspectives in the Eclipse GUI) in which he performed those actions. Table II depicts the number of times that the participant performed a navigation action in each context and gives the hit ratio (*N*=10) of each model for predicting the navigations in each context.

Table II shows that the overall high accuracy of Recency and Working Set largely resulted from their accurate predictions of navigations in the context of Eclipse's Debug view and Java Editor tabs. These models were able to accurately predict navigations in the Java Editor tabs context because tab close and tab select actions always take the

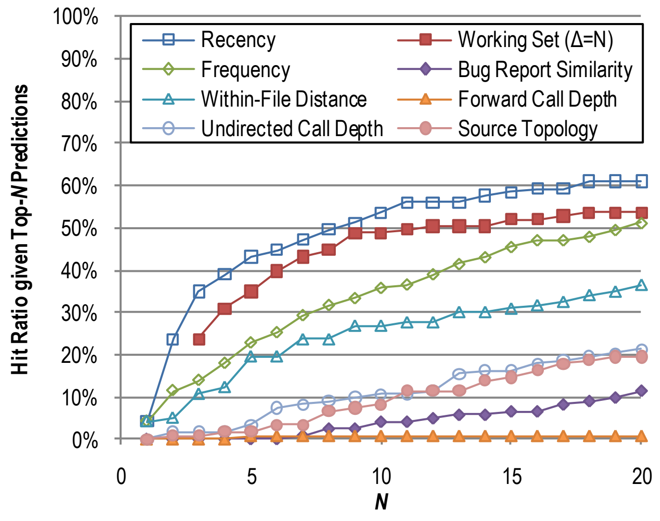


Figure 3. Model accuracy for click-based navigations, indicating the percentage of navigations that fell in the top-*N* predictions of each model.

programmer to locations that he has previously visited. The two models were also able to accurately predict navigations in the Debug view because the participant used the debugger to repeatedly step over the same sections of code.

The modeling results presented so far both agree with and expand upon the work of Parnin and Gorg [11]. Two of their models, Least-Recently Used (LRU) and Least-Frequently Used (LFU), correspond to our Recency and Frequency models, and our results for those models are consistent with those of Parnin and Gorg (i.e., Recency was more accurate than Frequency). We expand on Parnin and Gorg's work by evaluating models that go beyond caching (e.g., Bug Report Similarity and Source Topology). However, as with Parnin and Gorg's results, the most accurate model overall was Recency.

B. *Analysis of View-Based Navigations*

When we extended the analysis to include all 676 view-based navigations (a considerable increase from the 123 click-based navigations), model accuracy shifted noticeably from both our results for click-based navigations and the results of Parnin and Gorg. In particular, Within-File Distance and Source Topology exhibited a substantial improvement in accuracy. Figure 4 depicts view-based results.

As with the click-based navigations, all single-factor models exhibited low accuracy at *N*=1 with none exceeding a hit ratio of 4%. However, for *N*=3, the hit ratio of the Within-File Distance model jumped to 83.4%, just 3.4% shy of its *N*=20 hit ratio of 86.8%.

Our analysis of view-based navigations, as seen in Table III, highlights the importance scrolling operations (which account for the vast majority of our 676 view-based navigations). The Within-File Distance model benefited most from the profusion of scrolling navigations because the majority of scrolling navigations were to the method above or below the current one.

Table III shows that Source Topology owes its accuracy to scrolling navigations in the Java Editor as well. For navigations in the Java Editor, the model's hit ratio was

TABLE II. MODEL ACCURACY (CLICK-BASED DATA) BROKEN DOWN BY THE CONTEXT IN WHICH NAVIGATION ACTIONS OCCURRED AT N=10. CONTEXTS INCLUDE INDIVIDUAL ELEMENTS (MIDDLE 7 ROWS) AND TWO ECLIPSE —PERSPECTIVES” ENCOMPASSING MULTIPLE ELEMENTS (BOTTOM 2 ROWS). THE BEST HIT RATIO FOR EACH CONTEXT IS SHOWN IN **BOLD.**

Navigation Action Context	Number of actions	Single-Factor Model							
		Recency	Working Set (Δ=N)	Frequency	Bug Report Similarity	Within-File Distance	Forward Call Depth	Undirected Call Depth	Source Topology
Overall	123	**54%**	49%	36%	4%	27%	1%	11%	8%
Debug view	45	**62%**	60%	33%	0%	38%	0%	4%	9%
Java Editor tabs	32	**69%**	53%	50%	3%	0%	3%	22%	3%
Java Editor	23	52%	39%	39%	9%	**57%**	0%	9%	9%
Package Explorer view	10	10%	**40%**	10%	10%	0%	0%	0%	0%
Call Hierarchy view	7	14%	14%	14%	14%	0%	0%	**29%**	14%
Find Utility	5	40%	20%	40%	0%	**60%**	0%	0%	40%
Java Outline view	1	0%	**100%**	0%	0%	0%	0%	0%	0%
Java Perspective	75	**48%**	43%	36%	7%	19%	1%	15%	8%
Debug Perspective	48	**63%**	58%	35%	0%	40%	0%	4%	8%

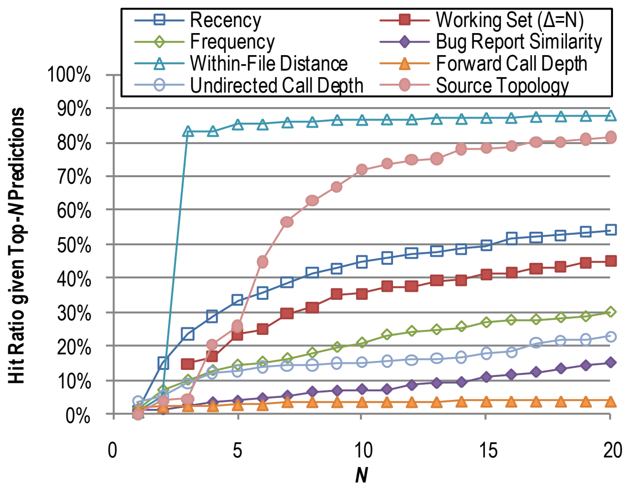


Figure 4. Model accuracy for view-based navigations.

83.1%. This result makes sense because the Source Topology graph contains edges that represent within-file adjacency relationships. However, the graph also contains edges representing other relationships, such as call relationships, that detracted from the model's accuracy in predicting the view-based navigations.

In summary, the type of navigation made a considerable difference in the number of navigations and the accuracy of certain models. Previous work did not account for the broad number of ways that programmers can navigate. In particular, the view-based navigations captured a multitude of scrolling navigations not logged by the click-based navigations considered in related work. These scrolling navigations led

TABLE III. MODEL ACCURACY (VIEW-BASED DATA) BROKEN DOWN BY THE CONTEXT IN WHICH NAVIGATION ACTIONS OCCURRED FOR N=10.

Navigation Action Context	Number of actions	Single-Factor Model							
		Recency	Working Set (Δ=N)	Frequency	Bug Report Similarity	Within-File Distance	Forward Call Depth	Undirected Call Depth	Source Topology
Overall	676	45%	35%	21%	7%	**87%**	4%	15%	72%
Debug view	46	**59%**	57%	11%	0%	37%	2%	22%	13%
Java Editor tabs	33	**55%**	46%	39%	0%	0%	0%	6%	0%
Java Editor	574	44%	34%	21%	8%	**99%**	4%	16%	83%
Package Explorer view	10	0%	0%	10%	**30%**	0%	0%	0%	0%
Call Hierarchy view	7	**29%**	**29%**	**29%**	0%	0%	0%	0%	0%
Find Utility	5	20%	20%	40%	20%	**60%**	0%	0%	40%
Java Outline view	1	0%	0%	0%	0%	0%	0%	0%	0%
Java Perspective	559	40%	29%	22%	9%	**89%**	3%	13%	76%
Debug Perspective	117	69%	64%	14%	0%	**74%**	5%	22%	52%

Within-File Distance to achieve an overall hit ratio approaching 90%.

VII. RESULTS: MULTI-FACTOR MODELS

Given that different single-factor models did better at predicting different kinds of navigation (as shown in Tables II and III), it stands to reason that *composite models*, which combine multiple single-factor models, might predict programmer navigations more accurately than individual single-factor models. However, it is not obvious which combinations of single-factor models can provide the greatest improvement in accuracy. To gain insights into this question, we investigated the maximum accuracy that composite models might reasonably be expected to produce for our participant data.

A. *Optimal Composite Models*

To understand the potential of composite models, we computed the results for an *optimal composition* of every possible combination of the single-factor models. For each attempted prediction, an optimal composite model yields a hit if *any* of its component models would yield a hit. Table IV shows the results for combinations of five single-factor models using the click-based and view-based navigation data (considering only the models' top-10 predictions). (We elided from the table the Working Set model because its predictions largely overlapped with Recency, and the Forward and Undirected Call Depth models because they exhibited low accuracy all around.)

For the click-based navigations, a composition of Recency and Within-File Distance yielded the most accurate predictions (63.4%), an improvement of 9.8% over the most accurate single-factor model, Recency. Adding Bug Report Similarity to this composite yielded the most accurate 3-model composition, further improving the accuracy by 3.2%. The most accurate compositions of more than 3 models yielded comparatively minor improvements in accuracy.

For the view-based navigations, a composition of Recency and Within-File Distance again yielded the greatest accuracy (91.9%), improving over the most accurate single-factor model, Within-File Distance, by 5.2%. Adding additional models to this composite yielded only minor improvements in accuracy: +1% by adding Frequency and another +0.5% by adding Bug-Report Similarity.

Regardless of the type of navigations, the trend for increasing composite size was one of diminishing returns in accuracy. Compositions of all 5 models yielded improvements of only +4.8% (click-based) and +1.8% (view-based) over the most accurate 2-model composites.

In summary, our investigation of optimal composite models reveals the potential of considerably improving the accuracy of the best performing single-factor models. Our results show that a composition of Recency and Within-File Distance is a particularly promising avenue for future research. However, optimal composites are theoretical in nature, and it

TABLE IV. RESULTS FOR OPTIMAL COMPOSITIONS OF FIVE SINGLE-FACTOR MODELS (TOP-10 PREDICTIONS) ON THE CLICK-BASED AND VIEW-BASED NAVIGATION DATA. **BOLDED** VALUES INDICATE MAXIMUMS FOR A GIVEN COMPONENT SIZE AND DATA SET.

Composite Size	Recency	Frequency	Within-File Distance	Bug Report Similarity	Source Topology	Click-Based Navigations		View-Based Navigations	
						Hit Ratio given Top-10 Predictions	Diff. with Recency (53.7%)	Hit Ratio given Top-10 Predictions	Diff. with Within-File Distance (86.7%)
2	✓	✓				55.3%	+1.6%	49.1%	-37.6%
	✓		✓			**63.4%**	**+9.8%**	**91.9%**	**+5.2%**
	✓			✓		57.7%	+4.1%	49.9%	-36.8%
	✓				✓	56.9%	+3.3%	83.6%	-3.1%
		✓	✓			52.0%	-1.6%	89.6%	+3.0%
		✓		✓		39.0%	-14.6%	27.1%	-59.6%
		✓			✓	41.5%	-12.2%	76.5%	-10.2%
			✓	✓		30.1%	-23.6%	87.1%	+0.4%
			✓		✓	29.3%	-24.4%	87.0%	+0.3%
				✓	✓	12.2%	-41.5%	72.9%	-13.8%
3	✓	✓	✓			65.0%	+11.4%	**92.9%**	**+6.2%**
	✓	✓		✓		58.5%	+4.9%	54.0%	-32.7%
	✓	✓			✓	58.5%	+4.9%	84.8%	-1.9%
	✓		✓	✓		**66.7%**	**+13%**	92.3%	+5.6%
	✓		✓		✓	64.2%	+10.6%	92.2%	+5.5%
	✓			✓	✓	61.0%	+7.3%	84.8%	-1.9%
		✓	✓	✓		54.5%	+0.8%	90.1%	+3.4%
		✓	✓		✓	52.8%	-0.8%	89.9%	+3.3%
		✓		✓	✓	44.7%	-8.9%	77.5%	-9.2%
			✓	✓	✓	32.5%	-21.1%	87.4%	+0.7%
4	✓	✓	✓	✓		**67.5%**	**+13.8%**	**93.3%**	**+6.7%**
	✓	✓	✓		✓	65.9%	+12.2%	93.2%	+6.5%
	✓	✓		✓	✓	61.8%	+8.1%	85.8%	-0.9%
	✓		✓	✓	✓	**67.5%**	**+13.8%**	92.6%	+5.9%
		✓	✓	✓	✓	55.3%	+1.6%	90.4%	+3.7%
5	✓	✓	✓	✓	✓	**68.3%**	**+14.6%**	**93.6%**	**+7.0%**

remains an open question *how* to compose models to achieve high accuracy.

B. *PFIS3 Multi-Factor Models*

To investigate the question of how to compose models, we looked to a family of four multi-factor models from our prior work. The models, known collectively as *PFIS3* [10], are based on Information Foraging Theory [12] and combine factors via spreading activation, a technique that has been successfully used in modeling both programming and web-based navigation (e.g., [10][12]). (The PFIS3 models were originally called *PFIS2*, but we have since updated how the models combine factors, described below.)

Each PFIS3 model builds upon the basic PFIS3 model (referred to simply as *PFIS3*). PFIS3 uses a *source topology+words graph* G_j that extends the source topology graph used by the Source Topology model. Formally, W_j is the set of words that occur in all the source files that the programmer has opened thus far. As with the Bug Report Similarity model, the words are processed to break up camel case words and remove stop words. The graph G_j is constructed the same way as the source topology graph, except that it also contains a unique vertex for each word in W_j. Additionally, it contains an edge from a word vertex w to a non-word vertex x if the text associated with x (be it name, definition, or Javadoc) contains w.

The basic PFIS3 model combines two factors, source-code topology and method-text similarity, using an algorithm based on *spreading activation* [12]. Formally, each vertex in G_j has an activation value, initially 0. The activation of the current method vertex m_j is set to 1. The model spreads activation for two iterations using a standard spreading activation algorithm with α=0.85 and edge weights=1; however, on the first iteration, only word vertices receive activation, and on the second iteration, only non-word vertices receive activation. Thus, each vertex receives activation only once.

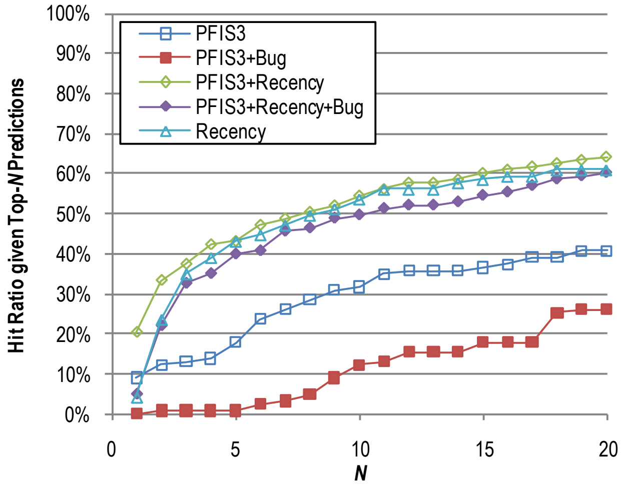
Building on this spreading-activation approach, three variants of PFIS3 add additional factors to the basic model. The variants add their factors by initializing certain vertices in G_j with additional activation. One variant, *PFIS3+Recency*, adds a recency factor by initially activating vertices associated with methods that have been previously visited, with more recently visited methods receiving greater activation. Formally, for method m_k in H_j such that $1 \leq k \leq j$, the m_k vertex is initialized to 0.9^{j-k+1}. Another variant, *PFIS3+Bug*, adds a bug-text similarity factor by initializing the activation of word vertices corresponding to words in the bug report. Formally, for all w in W_j, if w is in the bug report, then the activation of the w vertex in G_j is initialized to 1. The final variant, *PFIS3+Recency+Bug*, adds both recency and bug-text similarity factors by employing the activation initializations of both PFIS3+Recency and PFIS3+Bug.

We evaluated the PFIS3 models on the click-based and view-based navigation data and compared their results with the single-factor models. Figure 5 depicts the results.

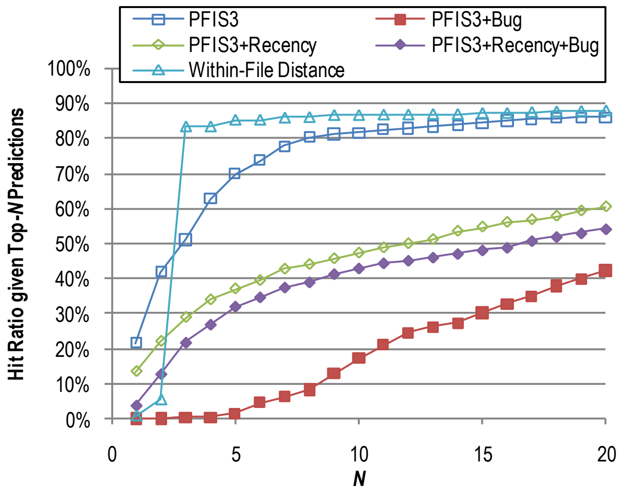
The results demonstrated that spreading activation could help to slightly increase accuracy. For the click-based navigations, the PFIS3+Recency model yielded slightly greater accuracy than the most accurate single-factor model, Recency. However, the other PFIS3 models showed less promise, failing to achieve the accuracy of Recency on the click-based navigations. Moreover, for predicting the view-based navigations, none of the PFIS3 models equaled Within-File Distance in accuracy, although the basic PFIS3 model did come close.

Although PFIS3+Recency was the most accurate model for the click-based navigations, its accuracy fell well below the potential displayed by the optimal composite models. Looking at only the optimal models of size 2 (recall that PFIS3+Recency combines 3 factors), all such optimal models involving Recency were more accurate than PFIS3+Recency (with differences in hit ratio (N=10) from 0.8% to 8.9%).

PFIS3+Recency's predictions stood out as being consistent in their hit ratios across the click- and view-based navigations. PFIS3+Recency was the most accurate model for the click-



(a) Click-Based Navigations



(b) View-Based Navigations

Figure 5. Accuracy for the PFIS3 variants for (a) click-based and (b) view-based navigations. For comparison, each graph includes the results of the most accurate single-factor model.

based navigations, and only two single-factor models were more accurate for the view-based navigations (Within-File Distance and Source Topology).

Considering only the models' top prediction (i.e., *N*=1), PFIS3 and PFIS3+Recency were considerably more accurate than the single-factor models for both the click- and view-based navigations. The greatest hit ratio that any single-factor model reached for either data set at *N*=1 was 4.1%. In contrast, PFIS+Recency exhibited a hit ratio of 20.3% for the click-based data and 13.5% for view-based, and PFIS3 exhibited a hit ratio of 8.9% for the click-based data and 21.6% for the view-based data.

In summary, PFIS3's spreading activation based approach for composing factors demonstrated promise for one composition in particular, PFIS3+Recency. The model exhibited the greatest accuracy of any model in the context of click-based navigations. Furthermore, the model's predictions were reasonably robust across navigation types, and the model's top prediction was correct substantially more often than any of the single-factor models. However, the other PFIS3 models' spreading activation based approach was less successful in composing other factors to yield accurate models. Thus, our results both suggest the potential of a spreading activation based approach and motivate the investigation of new approaches to composing factors.

VIII. Discussion and Conclusion

Our investigation of models of programmer navigation found that (1) Recency was the most accurate model for predicting click-based navigations, which was consistent with prior work [11]. However, (2) scrolling actions were strikingly common, and accounting for them with a view-based operationalization of navigation revealed much higher accuracy for Within-File Distance and lower relative accuracy for Recency than previously reported. (3) Bug Report Similarity exhibited low accuracy all around—a surprising finding given that many approaches use bug reports as input (e.g., [2]). Our evaluation of optimally composed multi-factor models revealed (4) the high potential of composing Recency and Within-File Distance to enhance accuracy. Finally, (5) our spreading activation based multi-factor model, PFIS3+Recency, stood out as demonstrating consistent performance across both click- and view-based navigations.

Our work suggests the potential for more accurate models in the future. It is too early to say how accurate models must be to be useful to tools: we are still in the formative stages of our model research. Our analysis of optimal composite models suggests that hit ratios of 68%–94% (given top-10 predictions) may be possible by composing single-factor models, and we anticipate still more potential for improvement with context-aware models.

Although our models' raw predictions are not intended to be immediately useful suggestions for programmer navigation, our overarching goal is to predict programmer navigations accurately enough to reduce navigation cost (1) by offering ways to skip steps in a developer's navigation path and (2) by bringing information from the predicted places directly to the developer. For example, a tool might combine model predictions with code summarization to deliver summaries of code that programmers would likely visit. Alternatively, the model predictions might be used to impose an ordering on existing lists such as to-dos. These and similar tools would leverage the ability to predict where a developer will navigate next to proactively retrieve, synthesize, and visualize information from those locations, ultimately helping to accelerate information-foraging during software maintenance.

IX. Acknowledgments

This work has been supported in part by IBM OCR grants to Oregon State Univ. and Carnegie Mellon Univ., and by AFOSR grants FA9550-09-1-0213 and FA9550-10-1-0326.

X. References

[1] Baeza-Yates, R., Ribeiro-Neto, B. *Modern Information Retrieval*, Addison Wesley Longman, 1999.

[2] Cubranic, D, and Murphy, G. Hipikat: Recommending pertinent software development artifacts. In *ACM/IEEE ICSE*, 408-418, 2003.

[3] Czerwinski, M, Horvitz, E, and Wilhite, S. A diary study of task switching and interruptions. In *Proc. ACM CHI*, 175-182, 2004.

[4] DeLine, R, Khella, A, Czerwinski, M, and Robertson, G. Towards understanding programs through wear-based filtering. In *Proc. ACM Softvis*, 183-192, 2005.

[5] Ericsson, K. A. Valid and non-reactive verbalization of thoughts during performance of tasks. *J. Consciousness Studies, 10*, 1-18, 2003.

[6] Ericsson, K. A. and Simon, H. A. *Protocol Analysis: Verbal Reports as Data*,. MIT Press, 1993.

[7] Kersten, M, and Murphy, G. Mylar: A degree-of-interest model for IDEs. In *Proc. ASOD*, 159-168, 2005.

[8] Ko, A, Myers, B, Coblenz, M, and Aung, H. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans. Soft. Eng*,*32*, 971-987, 2006.

[9] Lawrance, J, Bellamy, R, and Burnett, M. Scents in Programs: Does Information Foraging Theory apply to program maintenance? In *Proc. IEEE VL/HCC*, 15-22, 2007.

[10] Lawrance, J, Burnett, M, Bellamy, R, Bogart, C, Swart, C. Reactive information foraging for evolving goals. In *ACM CHI*, 25-34, 2010.

[11] Parnin, C, and Gorg, C. Building usage contexts during program comprehension. In *Proc. IEEE ICPC*, 13-22, 2006.

[12] Pirolli, P. *Information Foraging Theory: Adaptive Interaction with Information*. Oxford University Press, 2007.

[13] Robillard, M, and Murphy, G. Automatically inferring concern code from program investigation activities. In *IEEE ASE*, 225-234, 2003.

[14] Schummer, T. Lost and found in software space. In *Proc. HICSS*, 2001.

[15] Sillito, J, Murphy, G, and De Volder, K. Questions programmers ask during software evolution tasks. In *Proc. ACM FSE*, 23-34, 2006.

[16] Sillito, J, De Voider, K, Fisher, B, and Murphy, G. Managing software change tasks: An exploratory study. In *Proc. ISESE,* 1-10, 2005.

[17] Singer, J, Elves, R, and Storey, M. Navtracks: Supporting navigation in software maintenance. In *Proc. IEEE ICSM*, 325-334, 2005.

[18] Sinha, V, Karger, D, and Miller, R. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Proc. IEEE VL/HCC*, 187-194, 2006.

[19] Storey, M, Ryall, J, Bull, R, Myers, D, and Singer, J. TODO or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proc. ACM/IEEE ICSE*, 251-260, 2008.

[20] Ying, A, Murphy, G, Ng, R, and Chu-Carroll, M. Predicting source code changes by mining change history. *IEEE Trans. Soft. Eng*., 30, 574-586, 2004.

[21] Zimmermann, T, Weißgerber, P, Diehl, S and Zeller, A. Mining version histories to guide software changes. *IEEE Trans. Soft. Eng*., 30, 429-445, 2005.