Improving Software Developers’ Fluency by Recommending Development Environment Commands

Emerson Murphy-Hill
Department of Computer Science
North Carolina State University
Raleigh, North Carolina
emerson@csc.ncsu.edu

Rahul Jiresal and Gail C. Murphy
Department of Computer Science
University of British Columbia
Vancouver, Canada
jiresal.murphy@cs.ubc.ca

ABSTRACT
Software developers interact with the development environments they use by issuing commands that execute various programming tools, from source code formatters to build tools. However, developers often only use a small subset of the commands offered by modern development environments, reducing their overall development fluency. In this paper, we use several existing command recommender algorithms to suggest new commands to developers based on their existing command usage history, and also introduce several new algorithms. By running these algorithms on data submitted by several thousand Eclipse users, we describe two studies that explore the feasibility of automatically recommending commands to software developers. The results suggest that, while recommendation is more difficult in development environments than in other domains, it is still feasible to automatically recommend commands to developers based on their usage history, and that using patterns of past discovery is a useful way to do so.

Categories and Subject Descriptors
D.2.6 [Software Engineering]: Coding Tools and Techniques

Keywords
discovery, software developers, commands, IDEs

1. INTRODUCTION
Software tools can help developers meet users’ increasing demands for better software. For instance, research suggests that documentation tools can help developers use unfamiliar application programming interfaces to create code several times faster [19], code smell tools can help developers make more informed judgments about design [15], and debugging tools can help developers fix bugs more successfully [7].

To use these tools, developers issue commands to their development environment, whether that environment is an editor like emacs [29] or an integrated development environment like Eclipse [28]. Commands may range from simple text editing functions, such as copying text, to sophisticated programming actions, such as generating code or synchronizing with a repository. In 2009, logs collected from over 1 million users interacting with the Eclipse development environment indicate the use of over 1100 distinct commands.

Although the community of all software developers use a wide range of commands, any given developer tends to use only a small number. After reaching their own personal fixpoint, developers tend to face difficulties discovering and becoming aware of other helpful commands. Consider the open resource command in Eclipse that allows a developer to simply type the first few characters of a file, press return and the desired file is opened almost instantly. Several bloggers have praised open resource, including “10 Eclipse Navigation Shortcuts Every Java Programmer Should Know” [23]. Other bloggers have noted it is the “biggest time-saver I’ve stumbled upon” [24], and a “command I use religiously” [25]. However, despite the acclaim given to this command, based on Eclipse Usage Data Collector [26] data for May 2009, of the more than 120,000 people who used Eclipse, more than 88% do not use open resource.

This command discovery problem can be addressed in many ways. One way is through searchable documentation, yet the developer must know of a command’s existence in order to search for it. Another way is to improve the visibility of a command, such as through a special toolbar, yet this comes at the expense of the discoverability of other commands that are not on the toolbar. Another way is through tip-of-the-day messages, which recommend a random command at regular intervals, yet the suggested command may have no bearing on what a developer actually needs.

In this paper, we address the command discovery problem by making custom command recommendations to a developer by analyzing the developer’s past history of command usage and comparing it to a larger community’s use of development commands. This approach builds on a similar approach taken by Matejka and colleagues [9, 12] to recommend commands useful in a computer-aided drawing application, AutoCAD. Since Eclipse includes about 1100 distinct commands while AutoCAD contains about 750 [9], recommending commands in development environments appears to be more challenging; indeed, our results in Section 5 suggest that existing algorithms do not perform as well with Eclipse data as they do with AutoCAD data.

We extend Matejka and colleagues’ work by introducing four novel algorithms for recommending relevant commands to...
software users. We evaluated six algorithms in two evaluations: one automated evaluation and the second with live software developers. In the automated evaluation, we found that our best novel algorithm made recommendations with about 30% accuracy, which is about 20% higher than recommendations produced by the best previous algorithm. Although 30% accuracy may seem low, we believe adopting a small number of commands at a continual rate is an appropriate way to learn new commands and that delivery mechanisms are available to make it possible for developers to learn those recommendations that are relevant. We discuss these points in further detail in Section 8.

We first review previous work on feature awareness and command recommendations (Section 2). We then present a proof-of-concept study that explores discovery trends in Eclipse to help explain the motivation behind our novel algorithms (Section 3). We then describe the existing algorithms we applied and the novel algorithms we developed (Section 4) before presenting the results of an automated exploration of the effectiveness of the algorithms (Section 5). We also performed a study with developers to gain human insight into the value of the recommendations (Section 6). We discuss various aspects of our approach (Section 7) and future work (Section 8) before summarizing the paper (Section 9).

2. RELATED WORK

Several pieces of previous research have attempted to investigate the problem of command awareness and ways to recommend commands to users of complex software. We divide the latter work into three categories: human-to-human, inefficiency-based, and history-based recommendation.

2.1 Software Feature Awareness

The challenge of making software learnable has been a topic of considerable study among the human-computer interaction community. One aspect of learnability is awareness of what functionality is available in a user interface. When Grossman and colleagues conducted a study of learnability issues for users of AutoCAD, a computer-aided drafting application, they found that a “typical problem was that users were not aware of a specific tool or operation which was available for use” [6]. Although less well-documented, the awareness problem has been described in integrated software development environments as well, such as Campbell and Miller’s discussion of discoverability as a barrier to refactoring tool use [2].

2.2 Human-to-Human Recommendation

Perhaps one of the most natural, most overlooked, and most effective ways to learn is from a peer. In the general educational literature, situated learning, where a person learns something new from another person during routine activities, has been suggested as a mechanism by which knowledge may spread “exceedingly rapidly and effectively” [8, p. 90].

Learning software from peers appears to be effective as well. Suppose Hilaria needs to submit a bug report, but she does not know the command to use to attach a patch to the bug report. Hilaria asks a colleague, who shows Hilaria the menu item and hotkey for attaching a patch. This scenario is known as Over the Shoulder Learning, or OTSL for short, where a software user learns from another user at the same computer [20]. In the context of software development, Cockburn and Williams as well as Müller and Tichy have suggested that software developers learn effectively in this way during pair-programming [3, 13]. For example, Hilaria might be programming with a peer, notice that the peer is refactoring without using the refactoring commands, and may then recommend to the peer that they try a refactoring command. Our research has suggested that although discovering tools from peers in such a way is effective, it is relatively rare [16]. Moreover, a software developer cannot discover a new command using OTSL nor pair programming when she is working without other developers or she is the most fluent command-user in her peer group. The approach that we propose in this paper, using a recommender system, does not depend on a developer’s peer group.

2.3 Inefficiency-based Recommendation

One approach that does not rely on peers is a recommender system that makes command recommendations based on inefficient behavior. The Spyglass system, for example, works in a manner quite similar to the way that Hilaria recommended a refactoring command during pair programming [21, 22]. For example, when Hilaria is using Eclipse, Spyglass might notice that she often uses the find references command repeatedly on methods. Spyglass would then recommend that Hilaria instead use the show call hierarchy command, reasoning that the task that Hilaria is performing is walking up a call tree, a task that can be accomplished in fewer steps using the recommended command.

A problem with this approach is that the recommender system must be pre-programmed to recognize inefficient behavior as a precondition for each and every command that it want to recommends. For example, how such a systems identifies inefficient behavior prior to recommending the find references command is entirely different from how it recognizes inefficient behavior prior to recommending the open type hierarchy command. On the other hand, a history-based recommender system like the one in this paper, can recommend one command as easily as the next, without any special command-specific programming.

2.4 History-Based Recommendation

Other researchers have used history-based recommenders, the approach taken in this paper, for software systems in other domains. We briefly outline previous research here, and more extensively discuss the advantages and disadvantages of the algorithms in Section 4.

The Toolbox system created by Maltzahn and Vollmar creates a database of commands invoked by Unix users in order to identify commands that users have discovered recently [11]. Using the information in the database, Toolbox sends an email interview to a user who discovers a new command. In the email, Toolbox solicits information about how the new command works. The response is then distributed via an online newsletter, so that other members of the community can discover the command as well. This system relies on users completing the interview, whereas the algorithms described in this paper do not require user interaction for a recommendation to be made.

Linton and colleagues created OWL, a system that makes command recommendations to Microsoft Word users [10]. OWL observes what commands the community is using, and makes recommendations to individuals to change their command usage behavior. For example, if OWL observes that
the community of Word users use the find command more than any other command, yet you do not use it at all, OWL would recommend that you use find as well.

More recently, Matejka and colleagues proposed using a history-based recommender called CommunityCommands to recommend commands to users of AutoCAD, a computer-aided drafting system [12]. Matejka and colleagues used two collaborative-filtering algorithms to compare a user’s command history with the history of many other users of AutoCAD, and used the comparison to make a recommendation. The researchers showed that CommunityCommands’ best algorithm could predict the next command that an AutoCAD user would discover next about 27% more accurately than Linton’s algorithm. The researchers also showed that their algorithms could make command recommendations that AutoCAD users preferred over the recommendations made by Linton.

In this paper, we build on this work in two ways. First, we extend their recommender systems into a new domain, software development environments. Second, we explicitly build upon each of their algorithms to create several new algorithms, which we discuss in Section 4.

3. DISCOVERY TRENDS IN ECLIPSE

Several of the algorithms we investigate in this paper model how users discover commands as a means to predict new, potentially relevant commands. Why might this be a fruitful approach? To answer this question, in this section we present a small proof-of-concept study of how Eclipse users discover two sets of commands.

To perform this study, we use data from the Eclipse Usage Data Collector (UDC). This data set contains information submitted voluntarily by users of Eclipse, information including events that capture which Eclipse plugins are loaded, which user interface views and perspectives are shown, and which commands users invoke. Events in this data set are tagged by timestamps and anonymous user identifiers. We examined data from 2009, which includes a total of about 1.3 billion events from more than a million users.

We filtered this UDC data in two ways. First, we only included command events, such as invocations of the open resource command. Second, we included events only from users who reported data every month in 2009 because we wanted to improve our chances of exploring real, long-term discovery trends. This reduced the data set to about 22 million command uses from about 4300 Eclipse users.

To examine discovery trends, we chose to focus our study on two sets of commands in Eclipse and make two specific hypotheses:

- **CVS Commands.** CVS commands enable developers to interact with their CVS source revision repositories. Commands include update and commit, which enable developers to get the latest version from and commit to their repositories, respectively. We chose to study CVS commands because we view one command in particular — synchronize — as an advanced command that users would be more likely to discover after discovering other CVS commands first. We hypothesized this discovery ordering because synchronize is not strictly a necessary command to use CVS, but it may improve productivity.

- **Java Editing Commands.** Java editing commands enable software developers to more quickly edit their code. Examples of such commands include tools to toggle comments and perform refactoring. We chose to study editing commands because our previous experience with refactoring tools [17] suggests that one command in particular — rename — invokes a very basic refactoring tool, and thus we hypothesize that it is likely discovered before other refactoring commands.

We tested these hypotheses by building discovery graphs for both of these sets of commands. Figures 1 and 2 show these graphs, as produced by GraphViz [3]. In each graph, each node represents a command that users can discover. We define a command discovery as occurring only when two conditions are met. First, a command discovery can only occur after the user’s base command usage window, which represents a period of time where we assume that all commands used in this period are tools that the developers know, and any commands used after this period are commands that the developer has discovered. In this paper, we fix the base command usage window to the first 40 sessions for which we have data. We divide a user’s command history into sessions by grouping two commands into the same session if they were invoked within 60 minutes of one another. The motivation for using sessions is to remove spurious discovery preconditions — for instance, if two commands are discovered around the same time, we assume that learning about one command was not a precondition for learning about the other. Based on a manual exploration of the UDC data, we judged that the 60 minute tolerance and the 40 session windows were reasonable values to group working periods together, although we did not perform full sensitivity analysis. Second, a command discovery occurs the first time a user uses a new command only if she uses that command a second time at some point later in the data set. We add this second condition to attempt to rule out false-positive discoveries where the user inadvertently used a command but then never used it again.
In Figures 1 and 2, an edge from node a to node b represents users who discovered a command a first and then discovered b in a later session. The thickness of a line represents the number of users who discovered a and then b. Thus, thicker lines mean that more people discovered a then b. The lightness of the line indicates the number of users who first discovered b, then a. Thus, perfectly black lines indicate that everyone who discovered a and b discovered a first, whereas very light lines indicate that just as many people discovered a first as discovered b first. To reduce the visual complexity of the graphs, we eliminated edges which represent the discovery of only a few people; for Figure 1 we excluded edges with fewer than 7 people (per direction), and in Figure 2 fewer than 14.

For example, in Figure 2, notice a fairly dark line from rename element to extract method. The line thickness indicates that 18 people first discovered the rename element command, then discovered the extract method command. The line lightness indicates that 6 people first discovered extract method, then discovered rename element. As another example, Figure 1 shows a light, thin line from showHistory to compareWithRemote. For this line, 8 people discovered the showHistory command first, and 6 people discovered the compareWithRemote command first.

In the whole data set, a large number of people use CVS (n=41,734) and Java editing commands (n=351,657), but in the filtered data set, most trends in the graphs are supported by typically fewer than 20 people. This is partly because many people only reported data for a short period of time, and partly because some people used the same tools both before and after the base command usage window.

By inspecting the two discovery graphs, we can evaluate our hypotheses with regard to the rename refactoring and CVS synchronize commands. Looking at Figure 1 sync does not actually seem to be discovered after other CVS commands, but is actually a predecessor of update and compareWithRemote. Looking at Figure 2 we can see that rename element appears as a predecessor to the three other refactoring commands shown (move element, extract method, and extract constant).

Overall, the graphs reveal that there are discovery trends common between difference Eclipse users, although we did not consistently predict those trends. Certain commands do tend to be discovered earlier than other commands, suggesting that usage data may be a valuable predictor of the order of command recommendations, and that developers with different levels of command maturity may benefit from being recommended different commands. For example, a developer who is unfamiliar with refactoring may find it more useful to first be recommended the rename command before the extract method command.

### 4. DESCRIPTION OF THE ALGORITHMS

In this section, we describe two existing algorithms (as indicated by the sub-section headings) and four new algorithms for recommending commands to users. Throughout this section, we will use a simple hypothetical example, shown in Table 1. In the table, each row represents a developer, Hilaria, Ignacio, and Chuy, each using the same development environment on different machines. Each user happens to use only two commands per day on Monday, Tuesday, and Wednesday; these commands are called Copy, Rename, Cut, Commit, Paste, Format, and Update. For example, on Monday, Hilaria uses Copy once, then Rename once; on Tuesday she uses Copy once, then Paste once; and on Wednesday she uses Copy once, and then Format once.

Using different techniques, each algorithm provides a suggestion for which command we should recommend to a user on Thursday. For example, based on Table 1 we can see that Chuy does not use command Paste nor command Format, so we could reasonably recommend either command to him. Which command is more useful?

<table>
<thead>
<tr>
<th>Developer</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hilaria</td>
<td>Copy Rename</td>
<td>Copy Paste</td>
<td>Copy Format</td>
</tr>
<tr>
<td>Ignacio</td>
<td>Copy Cut</td>
<td>Copy Cut</td>
<td>Copy Paste</td>
</tr>
<tr>
<td>Chuy</td>
<td>Cut Commit</td>
<td>Cut Commit</td>
<td>Cut Update</td>
</tr>
</tbody>
</table>

Table 1: Developers’ command use over three days.
4.1 Existing: Most Popular

Linton and colleagues’ algorithm recommends the commands that are the most-used by the community. This algorithm is executed by counting the number of times each command has been used in the entire user community. This count represents the algorithm’s confidence level (δ) that this recommendation will be useful. To make a recommendation to a specific user, the algorithm recommends commands with the highest confidence values, excluding commands that the user already uses. Suppose that we want to recommend a command to Hilaria (Table 1). This algorithm would recommend command Cut, since Cut has the most uses (and thus highest confidence level) of any command that she does not already use.

The intuition behind this algorithm is that the commands that are used the most are also the most useful. A main advantage of this algorithm is that it can make recommendations to any user, including users whose history is unknown. One disadvantage is that its recommendations are not tailored; all users are assumed to be equal. This assumption may largely hold in some kinds of applications, but it is unlikely to hold in more complex software where different users perform different tasks using different workflows in different roles. As we alluded to in the introduction, development environments are likely to be one of these complex pieces of software due to differing languages (for example, python versus ruby), developer roles (programmer versus manager), and differing tasks (fixing bugs versus adding features). These differences may make recommendations made by this algorithm less relevant.

4.2 Existing: Collaborative Filtering

Collaborative filtering algorithms can be divided into two categories. The first is user-based collaborative filtering, where the command history of the user that we would like to make a recommendation for is compared to the histories of other users, and recommendations are made based solely on the commands used by users with similar histories. The second category is item-based collaborative filtering, where similarities between commands are calculated based on the community’s usage history, and a recommendation for a single user can be made by looking for commands similar to the ones he already uses. The confidence level δ for collaborative filtering algorithms is based on similarities between users or similarities of commands. We will not explain these algorithms in any more depth here; instead, we refer to reader to Matejka and colleagues for a technical explanation on command recommendation using collaborative filtering.[12]

Looking at our example (Table 1), suppose we use user-based collaborative filtering to recommend a command to Hilaria. The algorithm would find similarities in the histories of Hilaria and Ignacio because they both use commands Copy and Paste, and thus would recommend the command Cut. If we used item-based collaborative filtering to recommend a command to Hilaria, the algorithm would notice that, for example, someone who uses Copy also uses Cut, and since Hilaria uses Copy, it would thus recommend Cut.

The intuition behind this family of algorithms is that users and commands with similar histories are good sources of relevant recommendations. The main advantage of this family of algorithms is that recommendations are tailored to specific users. One disadvantage is that the algorithms ignore sequential and temporal aspects of command usage. For instance, suppose that Table 1 displays only the first week on the job for Hilaria and Ignacio, but Chuy been working with this software for 10 years. Also suppose that when Chuy started, his history looked very much like Hilaria’s, but changed into a more “expert” pattern of command usage within his first few weeks on the job. If we ran a user-based collaborative filtering algorithm, we would not see the similarity between Hilaria and Chuy, compared to Hilaria and Ignacio. This is a problem because Chuy’s history could be an excellent indicator of what command Hilaria should discover next, because Chuy was once a novice like Hilaria and successfully transitioned into an expert. In other words, traditional collaborative filtering ignores how usage patterns change over time.

4.3 Novel: Collaborative Filtering with Discovery

In the next family of algorithms, we model how usage patterns change over time with what we call “collaborative filtering with discovery,” which are based in part on sequential pattern mining.[15] With these algorithms, we model how users discover new commands, then run standard collaborative filtering on their discovery patterns.

The algorithm operates as follows. For each user in a community, given each user’s command history, determine their base command usage, B, that is, all of the commands that they know up to a certain point in time t (a tuning parameter). Then, initialize a ruleset D= { }. Then starting at time t+1:

1. For each used command x where x ∉ B, D=D∪{y → x : y ∈ B}.

2. Update B to reflect the newly discovered commands: B = B∪{x}

Continue these two steps until the end of the user’s command history is reached. Next, we apply either user-based collaborative filtering or item-based collaborative filtering to the ruleset D for each user in the community to obtain a recommendation for the desired user. The result will be a list of command recommendations, each in the form x → y at confidence level δ, where δ propagates from the underlying collaborative filtering algorithm. Ideally, the desired user will already know command x, and not know command y.

Looking at Tuesday and Wednesday, we see that: Hilaria will use all the commands that she knows over the course of just one day. Thus, B for Hilaria is {Copy, Rename }, B for Ignacio = {Copy, Cut }, and B for Chuy is {Cut, Commit }. Looking at Tuesday and Wednesday, we see that: Hilaria first discovers Paste, then Format; Ignacio discovers Paste, and Chuy discovers Update. This leads to the rule sets displayed in Table 2.

Using the rulesets in the table, we can then apply collaborative filtering; we will use user-based collaborative filtering
in this example. Since we want to make a recommendation for Ignacio, user-based collaborative filtering will notice a similarity between the discovery rulesets for Hilaria and Ignacio; they both share \{Copy \rightarrow Paste, Rename \rightarrow Paste\}. Because of this similarity, the algorithm will suggest discovery rules that may be applicable to Ignacio (Rename \rightarrow Paste, Copy \rightarrow Format, Rename \rightarrow Format, Paste \rightarrow Format\), resulting in recommendations for commands Rename and Format. Notice that this algorithm did not recommend Update, because there is no evidence that Ignacio and Chuy discover in the same way.

The intuition behind this family of algorithms is that it makes command recommendations based on discovery styles. One major limitation of this approach is that it requires a long enough usage history to capture discovery patterns. For example, if we would like to make a recommendation to a user who only has a week’s worth of history, we are faced with two difficulties when picking the \(t\) parameter. If we pick a \(t\) that is early (say, two days into the week), any commands that we observe him “discovering” later in the week may not be discoveries at all; they may in fact be commands that he only needed later on in the week. However, if we pick a \(t\) that is late (say, just before the last day of the week), we risk not observing any new commands being used, and thus the algorithm could make no recommendations at all.

4.4 Novel: Most Widely Used

A slight extension of the Most Popular algorithm is what we call the “most widely used” algorithm. This algorithm ignores repeated uses of a command from a user, and instead each command’s confidence level \(\delta\) is based on the number of people who use that command. In our example (Table 1), suppose that we want to recommend a command to Hilaria using this algorithm. Because commands Copy, Cut, and Paste are the most widely used commands, each used by two users in the community, and Hilaria does not use Cut, the algorithm would recommend Cut to Hilaria.

The intuition behind this algorithm is that a tool used by many people is a useful tool. The advantage is that it does not give undue weight to commands that are used a lot, but by few people. One disadvantage is that commands which few people use, even if the users find them exceedingly useful, are unlikely to be recommended using this algorithm.

4.5 Novel: Advanced Discovery

Another algorithm we call “advanced discovery” is a simpler version of the collaborative filtering with discovery algorithms above. The algorithm starts by producing a ruleset for each user in the community. Then, given a user who we would like to make recommendations for and the set of commands that she has used \(B\), we create a set of all discovery rules \(D\) from the community in the form \(x \rightarrow y\) such that \(x \in B\) and \(y \not\in B\). The algorithm then recommends the commands with the highest confidence level \(\delta\), where \(\delta\) is defined as the number of occurrences of \(x \rightarrow y\) in the set \(D\) such that the user under consideration has not used \(y\).

4.6 Novel: Most Popular Discovery

This algorithm counts the number of times a a command is discovered in the community. A command is recommended when it has not already been used by the developer and has the highest \(\delta\), defined as the number of users who have used the command after the base command usage window.

For example, take the users in Table 1. The command Paste was discovered by 2 users, Format by 1 user, and Update by 1 user. With this algorithm, if we wanted to make a recommendation to Chuy, we would recommend Paste.

The intuition behind this algorithm is that it recommends commands that many other people have discovered in the past. The main disadvantage is the same as with the “Most Popular” algorithm; it may recommend commands that are not relevant to the tasks or style of the user.

5. AUTOMATED EVALUATION

We conducted an automated comparison of algorithms to investigate three research questions:

1. How feasible is automated recommendation of development environment commands?
2. How does automated recommendation of development environment commands compare to automated recommendation of commands in other software?
3. How do our novel algorithms compare to existing algorithms?

5.1 Implementation

We implemented each algorithm in Java. Our implementation uses data structures from the Sequential Pattern Mining Framework [39], an open-source data mining framework. Our implementation also uses collaborative filtering algorithms from Apache Mahout [27], an open source machine learning library. However, we modified Mahout’s machine learning algorithms to align with the algorithms described by Matejka and colleagues [12].

5.2 Methodology

To investigate our research questions, we conducted a \(k\)-tail evaluation as described by Matejka and colleagues [12]. A \(k\)-tail evaluation first divides each user’s command history into a training set and a testing set. The testing set contains the last \(k\) tools that the developer discovered, and...
the training set contains the entire usage history up until the first command in the testing set was discovered. Finally, the evaluation runs each algorithm on the training set, and compares the recommendations made by each algorithm to the commands in the testing set. To illustrate the evaluation, consider again Table 1 which shows how three imaginary users use commands. First, setting \( k=1 \), we create the training set for each user (for example, Hilaria = \{Copy, Rename, Copy, Paste, Copy\}) and the corresponding testing sets (for example, Hilaria = \{Format\}). Finally, we train each algorithm on the data in the training sets, and use the output of the algorithm to predict the command(s) in the testing set. In our example, a successful outcome for an algorithm is that, given the command usage for Hilaria, the algorithm recommends Format. In simple terms, a more successful algorithm will be one that can more often predict the last commands discovered by a set of users.

This type of evaluation has several advantages but also limitations, when compared to a live evaluation on real users (see Section 5). One advantage is that it allows the algorithm to make predictions for a very large number of users, reducing any biasing effect a single user might have. It also allows us to evaluate the algorithm with minimal disturbance to users, allowing us to avoid possible Hawthorne effects [4]. On the other hand, the evaluation only approximates the quality of a recommendation because it may falsely classify a use of a command as a discovery because the user may actually be using the command again after using it before data collection began, rather than discovering the command for the first time. Moreover, this evaluation only measures whether the algorithms correctly predict what command that users naturally discovered, not what command would have been most useful for them to discover.

We created two variants of the k-tail evaluation because the standard k-tail described by Matejka and colleagues makes a key assumption, namely that the last command used by a user is actually a useful discovery of a command by that user. However, an analysis of the UDC data suggests that some Eclipse users only used certain commands once; rather than being useful discoveries, these may instead have been instances where the user tried a command and did not find it useful. To mitigate this risk, we created two variants: k-tail multi-use and k-tail multi-session. With k-tail multi-use, we only include commands in the test set which have been used multiple times. With k-tail multi-session, we only include commands that have been used in multiple sessions. The original k-tail evaluation, where a single use constitutes a discovery, results for Eclipse were about 24% lower than the recommendations for AutoCAD. In Section 7, we discuss why such a difference may exist.

With regard to our second research question, the results suggest that recommendations for Eclipse commands are more difficult to achieve correctly than for AutoCAD. For comparison, on the graph we mark the three results obtained by Matejka and colleagues with an asterisk (*) [12]. For the three comparable algorithms, Most Popular, User-Based Collaborative Filtering, and Item-Based Collaborative Filtering, results for Eclipse were about 24% lower than the recommendations for AutoCAD. In Section 6, we discuss why such a difference may exist.

With regard to our third research question, overall our novel algorithms faired favorably compared to the existing algorithms. Linton’s original Most Popular algorithm [10] had the lowest percentage of correct recommendations. User-Based Collaborative Filtering using Discovery, a novel algorithm we introduced, faired the best in two out of three evaluations, being beaten in the Standard evaluation by Advanced Discovery. We were surprised how often Most Widely Used was correct, considering that it is such a straightforward extension of Linton’s original algorithm. Overall, this evaluation suggests that the User-Based Collaborative Filtering using Discovery algorithm provides the most useful recommendations of the algorithms studied.

The algorithms’ performance varied significantly. The most efficient, Most Popular, took a few milliseconds to make each recommendation, whereas User-Based Collaborative Filtering with Discovery took up to 10 minutes to make recommendations for a user. We view even 10 minutes as acceptable because, in a practical implementation, each user’s recommendations would be computed on her own computer, in the background. Such distributed recommender system algorithms have been explored by Ali and van Stam [1].

6. LIVE EVALUATION

We performed a second study to evaluate the quality of our recommendations. In this study, we again borrow from Matejka and colleagues [12] and try to make recommendations to real developers, and then ask for their feedback.
about the quality of recommendations. This study complements the automated study in that it significantly alleviates the automated study’s main limitation, that is, that it was unclear whether the recommendations were truly useful.

6.1 Participants and Dataset

While the Matejka and colleagues study was able to contact and recruit AutoCAD users because their own company developed AutoCAD, we did not have any direct access to Eclipse users through the Eclipse Foundation, due to privacy concerns. Therefore, we took two small convenience samples of software developers; we treat each sample of developers separately in the remainder of this paper. We felt that taking a relatively small sample was appropriate at this stage in the research, before building a tool appropriate for field deployment and evaluation. We also felt that a convenience sample was acceptable, given that our methodology was double-blind, as we explain in Section 6.2. We again used the 2009 dataset to train the algorithms.

The first set of developers we call the “experts”: these 4 people had extensive software development experience and 5 to 8 years of Eclipse experience. All were located in Vancouver, Canada. The second set of developers we call “novices”: these participants were a mix of software developers and students, all with some Eclipse development experience. Each used Eclipse for Java development, and had between 1 month and 6 years of experience with Eclipse, with a median of 2 years of Eclipse experience.

6.2 Methodology

Matejka and colleagues used Linton’s Most Popular, User-Based Collaborative Filtering, and Item-Based Collaborative Filtering to generate 8 recommendations per algorithm for a total of 17 users of AutoCAD. Without knowing which algorithm recommended which command, users responded to a survey containing questions about novelty and usefulness of the command recommendations. For novelty, users were asked to use a 5-point Likert scale to rate their agreement with the statement “I was familiar with the command,” from “strongly agree” to “strongly disagree”. For usefulness, users rated on a 5-point Likert scale their agreement with the statement “I will use this command.”

We delivered the recommendations slightly differently to the two different participant sets. Whereas Matejka and colleagues sent their questions via survey, we asked the experts to rate the commands during a face-to-face or phone-based interview. This allowed us to infer why the commands were or were not novel and useful. For the novices, we evolved our methodology slightly; the second author showed the participant each command using screencasts or live screen sharing, told them what keyboard shortcut (if any) invoked the command, then asked participants to rate the novelty and usefulness of the command over the phone. This difference between the delivery methods for novices and experts presents a threat to the validity of our study, specifically, that novices may have rated novelty and usefulness differently than experts due in part to differences in delivery methods.

To reduce bias for both novices and experts, the first author of this paper produced the recommendations from each algorithm, combined and randomized the commands, then gave the list to the second or third author to use to interview the participants. In this way, the study was double-blind; neither the interviewer nor the participant knew which algorithm produced which command.

6.3 Results

Some algorithms were not able to recommend commands for every user; we did not have long enough command usage histories for several participants to make recommendations with the Collaborative Filtering with Discovery algorithms. This was the case for 1 of the 4 experts and 7 of the 9 novices. Participants were recommended a mean of 31
commands each because there was overlap in the command recommendation set produced by each algorithm.

Overall, how many recommendations were rated novel by participants depended on the user group. For experts, participants reported that only about 26% of commands were novel. For the novices, participants reported that about 80% of recommended commands were novel. In both cases, it was surprising that many commands were not considered novel by participants. In discussing why those commands were not novel, participants felt that either (1) they use an alternative command to the recommended one that they find suits them better or (2) that they already use the command. We suspect that when developers reported that they already use the recommended command, the reason is that the Eclipse Usage Data Collector does not always record a command usage in the data or records the command in a different way whether invoked via a key binding or a tool menu. These results suggest that further work on the data collection side of UDC to capture all command invocation events will improve the usefulness of all algorithms’ recommendations.

For the commands that participants rated as novel, Figure 4 displays how many commands each algorithm produced that participants rated as either useful or not useful. The left column lists the name of the algorithm. Asterisks indicate that, as we mentioned, these algorithms could not produce recommendations for some users. The middle major column indicates the number of useful and not useful recommendations for novices, while the right major column indicates the same for experts.

For experts, most algorithms produced fewer useful recommendations than not useful ones. Two exceptions that produced just as many useful as not useful recommendations were Item-Based Collaborative Filtering and User-Based Collaborative Filtering using Discovery. Item-Based Collaborative Filtering using Discovery was the only algorithm to produce more useful recommendations than not useful ones. This result is consistent with our automated evaluation that Collaborative Filtering with Discovery algorithms do well, as compared to the other algorithms.

For novices, all algorithms produced more useful recommendations than non-useful ones. The best performers were Linton’s Most Popular and the Item-Based Collaborative Filtering with Discovery algorithms, both of which produced zero not-useful recommendations. The Item-Based Collaborative Filtering and User-Based Collaborative Filtering with Discovery algorithms also performed well, having only one not-useful recommendation each. User-Based Collaborative Filtering produced the most useful recommendations, but also produced the most not-useful recommendations. Overall, the results where Collaborative Filtering with Discovery performing relatively well are consistent with our automated evaluation, although the results are more difficult to draw conclusions about because the Collaborative Filtering with Discovery algorithms were not able to produce recommendations for many novice users. The surprise from this evaluation is that Linton’s Most Popular did so well, when it performed so poorly on the automated evaluation. We discuss why this might be in the next section.

Despite these promising results, they likely do not reflect actual acceptance rates of recommendations if those recommendations were delivered by a fully automated system. Specifically, these results may overestimate the likelihood of a successful recommendation because a person delivered the recommendation. However, we are currently in the process of building a recommender system that accompanies tool recommendations with social endorsements from a developer’s peers [14]. Such endorsements may improve the likelihood that a recommendation is accepted. Nonetheless, a recommender system algorithm, such as the ones evaluated here, remain crucial to the success of this proposed system.

7. DISCUSSION

Overall, the results suggest that our Collaborative Filtering using Discovery algorithms provide relatively good command recommendations. However, our Collaborative Filtering using Discovery algorithms also require a longer-term usage history to make accurate recommendations than do simpler algorithms. Perhaps a reasonable compromise is a hybrid-based approach, where simpler algorithms are used to recommend commands to developers with little or no history, and more advanced discovery algorithms are used for developers with a more extensive history.

Our results suggest that good recommendations can be obtained by modeling short discovery patterns, such as that the rename command is usually discovered before the move element command. Our technique is based on sequential pattern mining [15], which in the past has been used to determine customer buying habits over time. Our technique could be expanded to more fully implement sequential pattern mining by modeling longer and more sophisticated patterns of discovery. Other data mining algorithms may also improve the quality of command recommendations further.

We were surprised that command recommendations for software developers were significantly (about 24%) less accurate than recommendations for AutoCAD users in k-tail evaluations. Although we are uncertain of the exact cause of this difference, we believe that it may be due to how commands are invoked in Eclipse versus in AutoCAD. In Eclipse, actions (such as switching editors) may or may not be invoked through commands, depending on how they are implemented and how well the implementors adhere to command-invocation conventions. Strict, uniform adhesion is especially unlikely because Eclipse is open-source and developed by many individual contributors, in contrast to AutoCAD, which is closed-source. Thus, the command predictability differences may not be as much related to differences in domain as to differences in command framework implementation. We hope to test this hypothesis by running our recommendations over data produced by closed-source development environments, such as Visual Studio.

In the live evaluation, novices were more readily accepting of recommendations than experts. There may be several reasons for this difference. First, novices simply know fewer commands than experts. Second, with novices we included the keyboard shortcut for each recommendation, whereas with experts we did not. Perhaps the fact that the recommendations were more concrete made them more appealing. Third, novices may be more flexible and open to more possibilities in the development environments they use. In other words, as developers gain an experience with an IDE, they may become more prone to “tool inertia” [15]. If that is the case, it may be beneficial for IDE command recommender systems to make recommendations early in a developer’s career, so that the system can build a track record of success by the time the developer becomes more experienced and it becomes harder to make useful recommendations.
Most/widely/used item based/user based advanced popular collaborative filtering with discovery

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Novices</strong></td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>9</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td><strong>Experts</strong></td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Collaborative Filtering</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Discovery</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Collaborative Filtering with Discovery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: The number of commands rated as useful or not useful by participants. Useful recommendations are solid bars; not useful recommendations are white bars.

We were surprised that Linton’s Most Popular algorithm performed so well for novices, yet did so poorly in the live expert evaluation, the automated evaluation, and in the evaluations performed by Matejka and colleagues [12]. This suggests that which recommendation algorithm will be the most useful may depend in part on how mature the software user is to whom the recommendation is made.

Both evaluations were limited in that we were unable to measure recall, that is, the fraction of total relevant commands that an algorithm recommended. Calculating recall would require, for each developer, that we determine which of the hundreds of commands that they did not use were relevant. This would have been impossible in the k-tail evaluation and impractical in the live evaluation.

8. FUTURE WORK

Building on the results presented here, we feel that there are several promising areas for future work.

One area for future work is improved detection of meaningful discovery and adoption events. Our live participants were surprised at the occasional poor quality of the recommendations, such as a Subversion user who was recommended three CVS commands by the User-Based Collaborative Filtering Algorithm. The problem could be that our current discovery algorithms assume that any time a new command is used (or used repeatedly), that it is an indicator of a useful command. Alternatively, future algorithms could recognize discovery as some other data pattern. For instance, an algorithm could detect replacement behavior, where a developer frequently uses a command for a period of time, then switches to using some other command frequently. For example, a developer may frequently use Eclipse’s find references command, but may later switch to using the open call hierarchy command, because the latter is a more efficient way to complete her tasks.

While the evaluations that we performed roughly estimate how well our algorithms make useful command recommendations, future studies could evaluate usefulness in more meaningful ways. Rather than making predictions about what command users would do naturally or ask developers whether they think that they would use the recommended commands, we could instead perform a longer term study where we evaluate whether the recommended commands are actually adopted. A clever study might be able to determine more objectively whether the recommendation of a command actually results in improved productivity, higher software quality, or novel software artifacts.

Finally, future command recommender systems may benefit from including richer information than simply which command is recommended. Specifically, one participant in our live evaluation wanted to know how much time a command would save him compared to the commands he already uses. Moreover, that developer not only wanted to know whether other developers find commands useful, but who those developers were, so he could assess his trust in those commands. In previous work, we speculated that leveraging trust in this way is essential for recommending commands that the developer will fully embrace [16].

9. CONCLUSION

In this paper, we introduced the notion of improving the fluency of software developers by automatically recommending development environment commands. Building on previous work, we introduced several novel recommendation algorithms, including three that model patterns of command discovery. In a proof of concept analysis and two studies drawing on data from several thousand Eclipse users, we demonstrated the utility of discovery patterns, compared several algorithms’ ability to predict command usage, and evaluated the novelty and usefulness of the algorithms’ recommendations with real developers. Our results suggest that it is feasible to recommend integrated development environment commands to software developers, but research remains to determine whether the recommended commands result in an improvement to the development process.

Acknowledgments

We thank our study participants and Giuseppe Carenini for their help. Thanks to Wayne Beaton at the Eclipse Foundation for access to data from the Usage Data Collector.
10. REFERENCES


Appendix
<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
<th><strong>Rationale</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Similarity Function</td>
<td>Cosine</td>
<td>Used to calculate the similarity between two users or two commands.</td>
<td>Same as used by Matejka and colleagues [12].</td>
</tr>
<tr>
<td>User Neighborhood Size</td>
<td>32</td>
<td>The number of similar users from which to draw commands to recommend for user-based collaborative filtering algorithms.</td>
<td>Left unspecified by Matejka and colleagues [12]; in initial experiments with user-based collaborative filtering, we found that a value of 32 provided good results.</td>
</tr>
<tr>
<td>Base Command Usage Window</td>
<td>40 sessions</td>
<td>See Section 4.3</td>
<td>Based on observations about UDC data; appears short enough to enable recommendations but long enough to include most commands the user knows.</td>
</tr>
<tr>
<td>k</td>
<td>1</td>
<td>See Section 5.2</td>
<td>Same as used by Matejka and colleagues [12].</td>
</tr>
<tr>
<td>Number of Recommendations</td>
<td>10</td>
<td>Maximum number of recommendations that each algorithm is allowed to produce.</td>
<td>Same as used by Matejka and colleagues [12].</td>
</tr>
<tr>
<td>α</td>
<td>1</td>
<td>Tuning parameter that serves as a scaling factor for confidence level (δ).</td>
<td>Left unspecified by Matejka and colleagues [12]; any positive value appears to produce the same results as any other positive value.</td>
</tr>
</tbody>
</table>

Table 3: Parameters used in algorithm comparison.