CommunityCommands: Command Recommendations for Software Applications

Justin Matejka[1], Wei Li[2], Tovi Grossman[1], George Fitzmaurice[1]

Autodesk Research
210 King St. East, Toronto, Ontario, Canada, M5A 1J7
[1]{firstname.lastname}@autodesk.com, [2]{firstname2.lastname}@autodesk.com

ABSTRACT

We explore the use of modern recommender system technology to address the problem of learning software applications. Before describing our new command recommender system, we first define relevant design considerations. We then discuss a 3 month user study we conducted with professional users to evaluate our algorithms which generated customized recommendations for each user. Analysis shows that our item-based collaborative filtering algorithm generates 2.1 times as many good suggestions as existing techniques. In addition we present a prototype user interface to ambiently present command recommendations to users, which has received promising initial user feedback.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors.

INTRODUCTION

Many of today's programs have not hundreds, but thousands of commands for a user to become aware of and learn [18]. In each release, more commands might be added, and without explicit effort on the part of the user to learn about new functionality, they are left untouched. For example, in Autodesk's AutoCAD, the number of commands has being growing linearly over time. And even with the thousands of commands available in AutoCAD, the largest group of users only use between 31 and 40 of them (Figure 1).

An inherent challenge with such systems is a user's awareness [14, 39] of the functionality which is relevant to their specific goals and needs. Awareness of functionality is not only important for learning how to accomplish new tasks, but also learning how to better accomplish existing tasks. In a potential "best case scenario", the user works with an expert next to them, who can recommend commands when appropriate.

While previous HCI literature has looked at intelligent online agents, most of this work is focused on predicting what current state a user is in, if they require assistance, and how to overcome problems [4, 8, 9, 15, 17, 20, 31]. To our knowledge, there are few examples of systems specifically focused on recommending new commands to users [24, 25]. Furthermore, such work has never been thoroughly implemented or evaluated, and has important limitations.

Systems which recommend content to users, known as "recommender systems" are very popular today in other domains. Some of the most popular movie, shopping, and music websites provide users with personalized recommendations [23, 29, 34, 36], and research in improving recommendation algorithms is an active field of research [2]. In this paper we introduce and investigate the application of modern recommender system algorithms to address the command awareness problem in software applications.

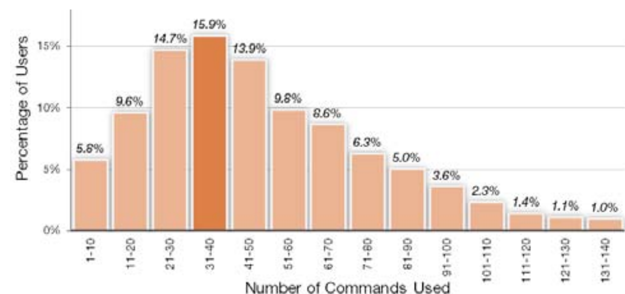

Figure 1. Histogram of the number of commands used by AutoCAD users. The largest group of users only use between 31 and 40 commands.

Our new system, CommunityCommands, collects usage data from a software system's user community, and applies recommender system algorithms to generate personalized command recommendations to each user. With CommunityCommands we hope to expose users to commands they are not currently familiar with that will help them use the software more effectively. The recommended commands are displayed in a peripheral tool palette within the user interface that the user to refer to when convenient. Thus, the system is much more ambient in nature compared to online agents such as "Clippy" or even simple techniques like "Tip of the Day". After discussing implementation details, we describe a 3 month evaluation of our recommender system algorithms, conducted with real users. Our new algorithms provided significantly improved recommendations in comparison to existing approaches.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'09, October 4-7, 2009, Victoria, B.C., Canada.
Copyright © 2009 ACM 978-1-60558-745-5/09/10 . . . $10.00.

RELATED WORK

Adaptive and Intelligent User Interfaces

Previous research has focused on adapting and optimizing the interface to the current user and context [4, 13, 29, 31], and inferring possible operations based on the user's behavior [20]. The domain knowledge base of such systems is often pre-designed and self-contained, and as such requires a large effort to build and maintain. With CommunityCommands, the knowledge base is acquired automatically from a large user community and evolves over time.

Other intelligent user interfaces are designed to observe and learn from users' actions, and accomplish personalized tasks. Examples include predicting the user's next command [8], automatically completing forms [15], maintaining calendars and emails [9, 17, 38], and assisting users in word processing tasks [26]. Most personal assistance programs analyze repetitive user behaviors, and automate the repetitions; in contrast, our system suggests useful commands that users usually have never used.

Notifications and Interuptability

Previous research [6, 7, 27] has demonstrated the harmful effects that notifications can have on a user's task performance. To compensate, there is a large body of work on determining when and how to best interrupt a user [12]. However this is a problem which remains open. CommunityCommands uses an ambient display, where the user can get the information when they are ready, thus avoiding the problems associated with interrupting the user's work flow.

Recommender Systems

Recommender systems have become an important approach to help users deal with information overload and provide personalized suggestions [16, 34, 37], and have been successfully applied in both industry and academia. Recommender systems support users by identifying interesting products and services, when the number and diversity of choices outstrips the user's capability of making good decisions. One of the most promising recommending technologies is collaborative filtering [16, 37]. Essentially a nearest-neighbor method is applied to a user's ratings, and provides the user with recommendations based on how her likes and dislikes relate to a large user community. Examples of such applications include recommending movies [30], news [34], books [23, 36], music [3], research papers [29], and school courses [10, 19]. However, research has shown that users may be reluctant to provide explicit ratings [5], and so our research considers an implicit rating system for software commands.

Recommending Commands

Little research has been conducted to help users learn and explore a complicated software package using a recommender system. Typical approaches to proactively introducing functionality to a user include "Tip of the day", and "Did you know" [32], but these are often irrelevant to the user and are presented in a decontextualized way [11].

The OWL System [24, 25] is one of the few systems we have identified as going beyond these simple solutions. The system, which is meant to run within an organization, compares a user's command usage to the average usages across the organization. The system would then make a recommendation if a command is being under-utilized or over-utilized by an individual in comparison to the group. This algorithm produces recommendations based on the assumption that all users in the community should share the same command usage distribution. It remains arguable whether this is a safe assumption to make, even within an organization, and the OWL system was never fully implemented or evaluated. But across an entire user community, this assumption is unlikely to hold. Users have different tasks, goals, and preferences, and so their recommendations should be personalized [31]. The system which Linton described is the exact type of system which recommender systems were developed to improve upon. CommunityCommands uses collaborative filtering to recommend the most relevant commands for each individual user.

DESIGN CONSIDERATIONS

There are a number of important design goals to consider when developing a command recommendation system.

User Interface Considerations

Unobtrusive. The interface should stay out of the user's way. We avoid a system that pops-up or forces the user to respond to the recommendation before continuing to work, since this type of system could be frustrating [12, 42].

In Context. The system should provide the recommendations within the application [22]. This way a recommendation can be viewed and tested with minimal switching cost.

Minimal Cost for Poor Suggestions. The interface should make the task of dealing with poor suggestions, if they do occur, lightweight to minimize frustration, allowing the user to spend more time looking at the good suggestions.

Self Paced. The user should be able to act on the recommendations when it is convenient for them.

Recommender System Considerations

Novel Recommendations. The recommendations should be commands that the user is unfamiliar with. This is in contrast to Linton's work, where recommendations were also made to increase or decrease existing command usages.

Useful Recommendations. The recommended commands also need to be useful for the user. This could mean that the command is useful immediately, or useful at some point in the future given the type of work the user does.

The combination of novel and useful recommendations leads to a two-dimensional space (Figure 2).

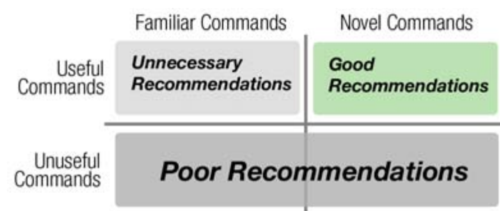

Figure 2. Map of Good, Poor, and Unnecessary Recommendations.

We consider a *good* recommendation to be a command that is both useful and novel to the user. A *poor* recommendation is a command that is not useful to the user. An *unnecessary* recommendation is a command which is useful to the user, but the user was already familiar with. Unnecessary recommendations can actually be helpful in improving the user's confidence in the system [29], but this is very dependent on the user's expectations. If the expectation is that the system will be suggesting "new" commands which may be useful, commands with which the user is already familiar may be seen as poor suggestions.

Global and Opportunistic Suggestions. The system should be able to provide global suggestions, based on the user's entire command history. However, the system could also have some knowledge about what the user is doing at the current moment so it is able to highlight suggestions which may be particularly relevant in the current context. Our work focuses mostly on global suggestions, but we will also discuss issues with opportunistic suggestions.

Support Different User Communities. The recommender system should be able to base its recommendations on different collections of users. Users may want to see recommendations generated from known expert users, a group of co-workers, or the entire user community of the software.

COMMUNITYCOMMANDS

We now describe CommunityCommands, a new system that provides personalized command recommendations using collaborative filtering algorithms. The general idea is to first compare a user's command frequencies to the entire user population. Our system then generates a top 10 list (although the list size could vary) of recommendations for that user. This top 10 list is presented in an ambient window within the user interface that the user can refer to when convenient (Figure 3).

Figure 3. CommunityCommands system overview.

Target Application

CommunityCommands is implemented within AutoCAD, a widely used architecture and design software application, made by Autodesk. We felt AutoCAD would be an excellent software package to work with, since it not only has thousands of commands, but also numerous domains of usages. While our work is implemented within AutoCAD, the concepts map to any software where command awareness may be an issue, and its usage varies across users.

Command Database

CommunityCommands requires usage data for its users to provide personalized commands. In AutoCAD, command usage histories are collected using a Customer Involvement Program (CIP). The data set we obtained is composed of 40 million *{User, Command, Time}* tuples collected from 16,000 AutoCAD users over a period of 6 months.

The "Ratings"

Typical recommender systems depend on a rating system for the items which it recommends. For example, a recommender system for movies may base its recommendations on the number of stars that user's have assigned to various titles. These ratings can be used to find similar users, identify similar items, and ultimately, make recommendations based on what it predicts would be highly rated by a user.

Unfortunately, in our domain, no such explicit rating system exists. Instead, we implicitly base a user's "rating" for any command on the frequency for which that command is used. Our collaborative filtering algorithm then predicts how the user would "rate" the commands which they do not use. In other words, we take a user's *observed* command-frequency table as input, and produce an *expected* command-frequency table as output.

Generating Recommendations

We explored two of the most commonly used collaborative filtering techniques: user-based [34] and item-based [36]. Both of the algorithms discussed have two inputs: the command history for each user in the community, and the command history for the user we are generating a recommendation, which we refer to as the *active user*.

User-Based Collaborative Filtering

User-based collaborative filtering generates recommendations for an active user-based on the group of individuals from the community that he/she is most similar to (Figure 4). The algorithm averages this group's command frequencies, to generate an expected command-frequency table for the active user. The algorithm details are described below.

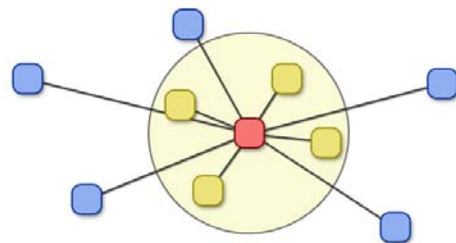

Figure 4. The active user is in red, and his expected frequency table will be compiled from his most similar neighbors (in yellow).

1. Defining Command Vectors

For user-based collaborative filtering we require a method to measure the similarity between two users. A common approach for doing this is to first define a representative vector for each user, and then compare the vectors.

A basic method is to define the command vector V_j such that each cell, $V_j(i)$, contains the frequency for which the

user u_j has used the command c_i. A limitation of using this approach is that in general, a small number of commands will be frequently used by almost everyone [41]. Thus, when comparing the vectors, each pair of users will tend to have high similarity because they will all share these popular high frequency commands. This is certainly the case in AutoCAD. For example, in Figure 5, we see that the top 10 commands make up 50% of all commands issued, and the top 100 commands make up 93% of all commands issued.

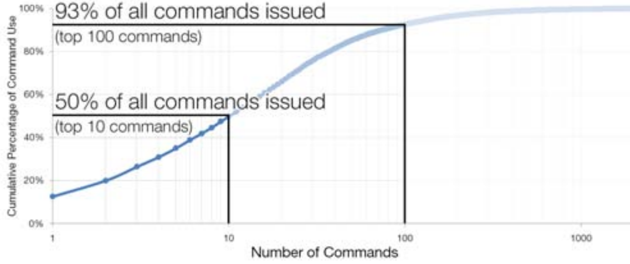

Figure 5. Cumulative percentage of command counts for the 2000 AutoCAD commands.

We need to suppress the overriding influence of commands that are being used frequently and by many users. Document retrieval algorithms actually face a similar challenge. For example, an internet search engine should not consider two webpages similar because they both share high frequencies of the words "a", "to", and "the". Such systems use a "term frequency inverse document frequency" *(tf-idf)* technique [21] to determine how important a word is to a particular document in a collection. For our purposes, we adapt this technique into a *command frequency, inverse user frequency (cf–iuf)* weighting function, by considering how important a command is to a particular user within a community. To do so, we first take the command frequency *(cf)* to give a measure of the importance of the command c_i to the particular user u_j.

$$cf_{ij} = \frac{n_{ij}}{\sum_k n_{kj}}$$

where n_{ij} is the number of occurrences of the considered command of user u_j, and the denominator is the number of occurrences of all commands of user u_j.

The inverse user frequency *(iuf)*, a measure of the general importance of the command, is based on the percentage of total users that use it:

$$iuf_i = log\frac{|S|}{|\{u_j : c_i \in u_j\}|}$$

where:

$|S|$: total number of users in the community

$|\{u_j : c_i \in u_j\}|$: number of users who use c_i.

With those two metrics we can compute the *cf-iuf* as

$$cf\text{–}iuf_{ij} = \alpha \cdot cf_{ij} \cdot iuf_{ij}$$

with α as a tuning parameter.

A high weight in *cf–iuf* is obtained when a command is used frequently by a particular user, but is used by a relatively small portion of the overall population.

For each user u_j, we populate the command vector V_j such that each cell, $V_j(i)$, contains the *cf-iuf* value for each command c_i, and use these vectors to compute user similarity.

2. Finding Similar Users

As with many traditional recommender systems, we measure the similarity between users by calculating the cosine of the angle between the users' vectors [35]. In our case, we use the command vectors, as described above. Considering two users u_A and u_B with command vectors V_A and V_B

$$\text{similarity}(u_A, u_B) = \cos(\theta_{V_A,V_B}) = \frac{V_A \cdot V_B}{\|V_A\| * \|V_B\|}$$

Thus, when *similarity* is near 0, the vectors V_A and V_B are substantially orthogonal (and the users are determined to be not very similar) and when *similarity* is close to 1 they are nearly collinear (and the users are then determined to be quite similar). As can be seen in Figure 6, using the cosine works nicely with our rating system based on frequencies, since it does not take into account the total number of times a user has used a command, but only its frequency.

We compare the active user to all other user in the community, to find the n most similar users, where n is another tuning parameter.

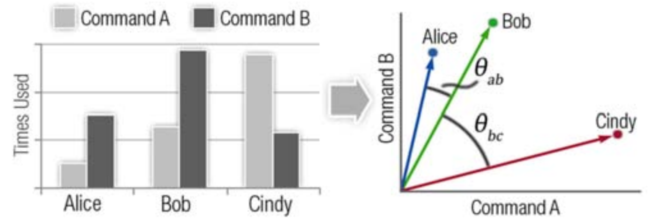

Figure 6. Simplified example of user similarity. Alice and Bob are more similar than Bob and Cindy as the angle between their command vectors is smaller.

3. Calculating Expected Frequencies

To calculate an expected frequency for each command, we take a weighted average of the command frequencies for the active user's n similar users. We define the expected frequency, ef_{ij}, for command c_i and user u_j:

$$ef_{ij} = \sum_{k=1}^{n} w_{jk}\, cf_{ik}$$

where w_{jk} is any weighting function (which can be tuned) and cf_{ik} is the frequency of command c_i and user k.

4. Removing Previously Used commands

Once we create a list of all the command frequencies, we remove any command which the user has been observed to use, preventing no known commands from being suggested.

5. Returning the Top 10 list

The final step is to sort the remaining commands by their expected frequencies. The highest 10 commands will appear in the user's recommendation list.

Item-Based Collaborative Filtering

Rather than matching users based on their command usage, our item-based collaborative filtering algorithm matches the active user's commands to similar commands. The steps of the algorithms are described below.

1. Defining User Vectors

We first define a vector V_i for each command c_i in the n dimensional user-space. Similar to user-based approach, each cell, $V_i(j)$, contains the *cf-iuf* value for each user u_j.

2. Build a command-to-command Similarity Matrix

Next, we generate a command-to-command similarity matrix, M. M_{ik} is defined for each pair of commands i and k as:

$$M_{ik} = \cos(V_i, V_k)$$

3. Create an "active list"

For the active user, u_j, we create an "active list" L, which contains all of the commands that the active user has used.

$$\boldsymbol{L_j} = \{\boldsymbol{c_i} | \boldsymbol{cf_{ij}} > 0\}$$

4. Find similar unused commands

Next, we define a similarity score, s_i, for each command c_i which is not in the active user's active list:

$$s_i = \text{average}(M_{ik}, \forall c_k \in L)$$

5. Generate Top 10 List

The last step is to sort the unused commands by their similarity scores s_i, and to provide the top ten commands in the user's recommendation list.

Domain-Specific Rules

The above techniques work without any specific knowledge about the application. In an initial pilot study, this was shown to lead to some poor recommendations which could have been avoided. Thus, we created two types of rules to inject some basic domain knowledge into the system.

Upgrades (A ⇏ B)

An upgrade is a situation where if you use command A there is no need for you to use command B. For example, if an AutoCAD user uses *MouseWheelPan* we would not recommend the *Pan* command, since it is a less efficient mechanism to activate the same function.

Equivalencies (A ⇎ B)

We consider two commands to be "equivalent" when it makes sense for a user to use one of the two commands, but not both. For example in AutoCAD there is the *HATCH* and *BHATCH* commands. *BHATCH* is from earlier versions of the product, but it does the same thing.

We spent approximately 2 hours with a domain expert to come up with 21 specific rules. Four of these rules were *Upgrades* and 17 of the rules were *Equivalencies*.

OFFLINE ALGORITHM EVALUATION

Here, we present an automated method to evaluate the recommender algorithms using our existing offline data. Although offline evaluation cannot replace online evaluation, it is a necessary and important step to tune the algorithms and verify our design decisions before the recommender system is deployed to real users.

The development of the algorithm was a challenging task since we required a metric that would indicate if a recommended command, which had never been observed, would be useful to a user. To do so we developed a new *k-tail evaluation* where we use the first part of a user's command history as a training set, and the rest of the history as a testing set. We choose to use the most recently used commands as the testing set as opposed to a random hold out to more closely map to our real usage situation.

Consider a user u_i with a series of commands S. k-tail evaluation divides this command sequence into a training sequence S_{train} and a testing sequence S_{test}, so that there are k unique commands in S_{train} which are not in S_{test}. For example, the command sequence in Figure 7 is a 2-tail series since there are two commands, SOLIDEDIT and 3D-ROTATE, which have never appeared in the training set.

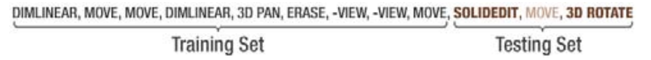

Figure 7. k-Tail evaluation of a command sequence.

To evaluate an algorithm, we find the average number of commands which are in both a user i's recommendation list R_i, and their testing set $S_{test,i}$. We define the evaluation result of k-tail as hit_k:

$$hit_k = \frac{\sum_{i=1}^{n} |R_i \cap ST_i|}{n}$$

where n is the size of community.

Algorithms

In addition to testing our user-based and item-based collaborative filtering algorithms, we also implemented and evaluated Linton's algorithm [24, 25]. The algorithm suggests the top commands, as averaged across the total community, which a user doesn't use.

Offline Results

All three algorithms were evaluated using the k-tail method and the offline CIP data. We only included users for which we had observed at least 2000 commands (4033 total users). The command sequence of each CIP user is divided into a training set and a k-tail. Figure 8 shows that when k=1, the item-based algorithm predicts the next new command correctly for 850 users, about 240 more than Linton's.

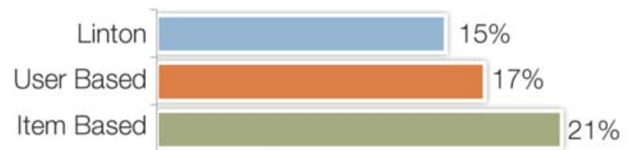

Figure 8. Offline results showing the percentage of times the next new command was predicted in a list of 10 by each algorithm.

ONLINE ALGORITHM EVALUATION

While our offline evaluation showed promise for our new techniques, the results may not be fully indicative of how our algorithms would work in practice. As such, we conducted an online "live" study with real users. We collected

data for a set of real users, generated personalized recommendations for each of them, and had them evaluate their recommendations in a web-based survey.

Participants

We recruited 36 users (25 male, 11 female) of AutoCAD 2009 to participate in the study. To be considered for the study users were required to use AutoCAD a minimum of 20 hours per week. Participants were aged 21 to 55 and worked in varying fields including architecture, civil planning, and mechanical design, across North America.

Setup

To capture the participants' command usages, we gave each participant a custom plug-in, which would give us access to their full CIP data from the time it was installed. Participants were asked to continue using AutoCAD as they normally would. Command data was recorded from each user for approximately 10 weeks. After collecting each user's data, the recommendations were generated.

During the course of the study we stopped receiving data from 12 of the participants (they changed computers, lost their job, company inserted a new firewall, etc.) leaving us with 24 viable participants. Three of these were used in a preliminary pilot study. We sent out 21 surveys, with 4 participants not responding, leaving us with 17 users completing the study.

Generating Recommendations

We used a within-participant design. That is, each participant was sent recommendations from each of the three algorithms. To do this, we generated a top 8 list for each of the three algorithms. We then took the union of these three lists, and randomized the order. Since the algorithms could produce common commands, the final lists could range in size, and in the study, ranged from 17 to 25 items.

Each user was sent a survey with questions about each of the commands in their customized recommendation list. For each recommended command participants were given a short description of the functionality (for example "*XLINE: creates an infinite line*"). Users were asked to familiarize themselves with the command as much as possible before answering any questions about it.

Participants were asked to rate the commands on the following 2 statements, using a 5-point Likert scale from *strongly disagree* to *strongly agree*:

Q1. I was familiar with this command.
Q2. I will use this command.

In an initial pilot study, we found that users sometimes claimed to use a command frequently, when we could tell from their data that they did not. This was often due to two different commands sounding similar. As such, in this study, we made it clear to the participants that they had not used any of the commands.

Results

Novelty and Usefulness

Recall our main design considerations for the recommender system was for it to produce useful and novel recommendations. We used responses to Q1 to measure novelty, and responses to Q2 to measure usefulness. Repeated measure analysis of variance showed a significant difference in average usefulness for technique ($F_{2,32} = 13.340$, $p < .0001$). The ratings were 2.82 for *Linton*, 3.18 for *User-Based*, and 3.44 for *Item-Based*. There was a significant difference between *Linton* and *Item-Based* ($p = .0001$) and *User-Based* and *Item-Based* ($p = .038$). The effect on technique on novelty ratings did not reach significance.

As discussed in the design considerations section we are interested in the quality of the individual recommendations (Figure 2), particularly those falling into the "good" or "poor" categories. As such we do not only want to look at usefulness ratings, but rather judge the quality of the lists which the algorithms provide by assessing the number of good and poor recommendations which they produce.

Good and Poor Recommendations

First, we consider good recommendations to be those where the user was not previously familiar with the command, but after seeing the suggestion, will use it. This corresponds to a response of *strongly disagree*, *somewhat disagree*, or *neither agree nor disagree* to Q1, and a response of *somewhat agree*, or *strongly agree* to Q2. We define the percentage of good recommendations as the average number of recommendations produced which were good.

Repeated measure analysis of variance showed a main effect for the algorithm ($F_{2,32} = 12.301$, $p < .0001$) on percentage of good recommendations. The overall percentages of good recommendations were 14.7% for *Linton*, 27.2% for *User-Based*, and 30.9% for *Item-Based*. Pairwise comparison using Bonferroni adjustment showed that both *User-Based* ($p = .006$) and *Item-Based* ($p = .001$) were significantly different from Linton, but the difference between *Item-Based* and *User-Based* was not significant (Figure 9).

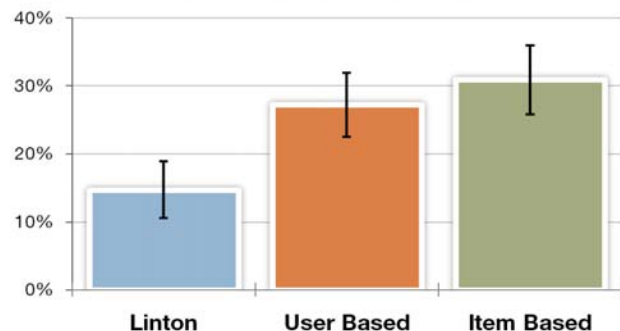

Figure 9. Percentage of "good" suggestions by technique.

We defined poor recommendations as those where regardless of previous familiarity, the user would not use the command, corresponding to a response of *strongly disagree*, or *somewhat disagree* to Q2.

Repeated measure analysis of variance showed a main effect for the algorithm ($F_{2,32} = 11.486$, $p < .0001$). The overall percentages of poor recommendations were 41.9% for *Linton*, 32.4% for *User-Based*, and 22.1% for *Item-Based*. Pairwise comparison showed that *Item-Based* was signifi-

cantly different from both *User-Based* and *Linton* ($p < .05$), but *User-Based* was not significantly different from *Linton* (Figure 10).

Overall, these results are very encouraging. Compared to Linton's algorithm, the item-based algorithm increased the number of good commands a user would receive by 110.2%, while reducing the number of poor commands by 47.3%, and in both cases the difference was significant. The user-based algorithm also showed promise, increasing good commands by 85.7%, and decreasing poor commands by 22.6%, although these differences were not significant.

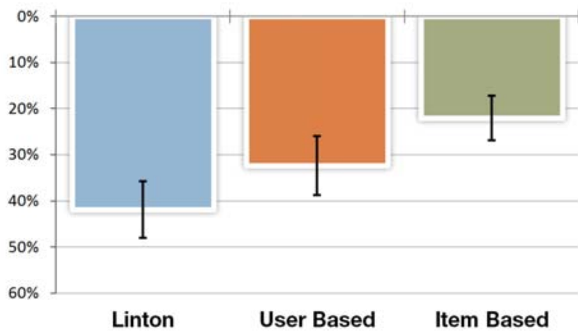

Figure 10. Percentage of "poor" suggestions.

Survey

Participants were asked to rate 6 design properties of a command recommendation system (Figure 11). The two features considered most important were *Makes Useful Recommendations* and *Easy To Dismiss*.

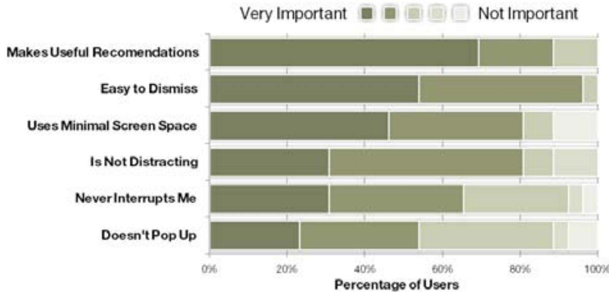

Figure 11. Subjective importance ratings for properties of a command recommender system.

We also asked the participants to estimate how many total commands they use. By looking at their command data for the period of the study we are able to compare their estimated with actual command counts (Figure 12). All but one of the participants underestimated the number of commands they used, and in most cases the amount was vastly underestimated. On average our participants use more than 300% as many commands as they estimated. Part of this may have been due to users having scripts which, when executed, would call a series of commands.

In general the participants both greatly underestimated how many commands they were using while at the same time greatly overestimating what percentage of the program's functionality they were using. These results show that users may not appreciate how many commands they need to use, and how much is still available for them to learn. This provides strong evidence for the existence of the awareness problem [15], even among professional users.

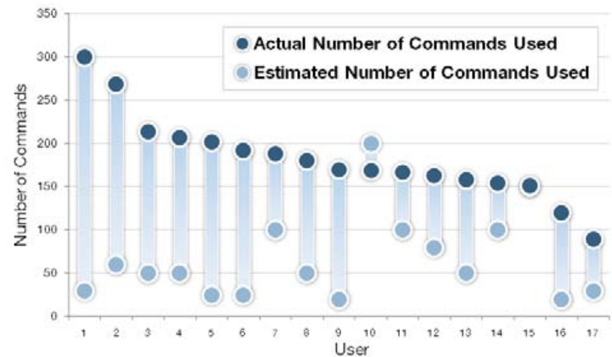

Figure 12. Actual vs. Estimated commands used.

However, we also asked users what *percentage* of all AutoCAD commands they thought they used. Taking this percentage, and the number of command they estimated they used, we can calculate how many commands they thought are in AutoCAD (Figure 13). All participants thought the number of commands in the program was much smaller than it truly is, with the average estimated size being 367, or approximately 1/5th the actual total command count.

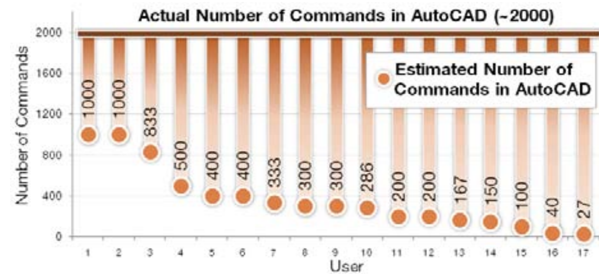

Figure 13. Actual vs. Estimated number of commands in the entire program.

Participant Interview

We went on a workplace site visit to interview two of the participants that had completed the study. The first participant is a manager who trains new employees and serves as a gatekeeper for learning new features. The second participant reports to the manager. We asked both participants to comment on every item in their personalized recommendation list of commands from the survey. In a few cases we found that a recommended command was known to the user but not used as it had a known limitation for their needs and an alternative workflow was used. We also found two cases where an upgrade rule could have been defined to prevent a poor recommendation.

In one case, the second participant was very interested in a recommended command (*CANNOSCALE*) and identified it as something that would be very critical for future workflows. However, since the current team project conventions had already been defined, using this new command would need to be deferred until the team transitioned to the new annotation workflow in a future project.

The manager was very focused on improving the team's efficiency. While she expects the team to already be very proficient in using the AutoCAD software, she stated that it would be valuable to collect CIP data for her company and individual team to detect training opportunities on underutilized commands. She also would like the ability to create a "blocklist" of commands to block from the recommendations lists or flag with a warning that using these commands may interfere with the group's standard procedure.

Through our interview we recognized that the recommendation list could also give team members an excuse to talk to one another in a structured way about workflow and to compare and contrast approaches. In one case, the manager heard a command mentioned (*XLINE*), and said that a colleague on her team would find it useful, and wrote it down. In general, we got the sense that individuals work in significantly different ways, yet their workflows are not shared or known to other team members. Thus, CommunityCommands could serve as a mechanism to expose alternative workflows from teammates and the larger community.

PRESENTATION UI

Our study has shown that we successfully developed a new command recommendation algorithm that can significantly improve the recommendations that a user will receive. This, in itself, is an important contribution. However, we still need to consider how to present recommendations to the user.

We designed a prototype of the CommunityCommands UI (Figure 14). The interface is implemented with a WPF control inside of an AutoCAD palette using the ObjectARX plug-in architecture. The interface contains a collection of buttons and a list containing the most highly ranked recommendations. Clicking on a command button will activate the command and hovering will bring up a tooltip.

A toggle at the top of the panel (Figure 15) controls the nature of the recommendations. By looking at the user's entire history we can generate recommendations which are appropriate for the overall type of work the user does. We call this "Long Term" recommendation. If we look at only those commands used recently (say in the last 15 minutes) we can generate "Short Term" recommendations which should be more appropriate for the type of the work the user is doing at the moment.

The top panel also contains buttons to access a command "notebook", view the command suggestion history, and filter the recommendations based on category.

The command notebook contains all of the commands the user has ever used and a place to store relevant tips or tricks about the command. These notebook entries are presented beneath the tooltip information when hovering over a command button. Besides seeing personal notes, users could also see notes from a manager or co-workers. Using this mechanism, managers could put company policies and best practices information in their notebooks, and that information would be visible to their entire teams.

With the history mechanism the user can see what commands have been suggested in the past. Since after the command has been used once it will no longer appear in the suggestion list, this is a way to revisit previous suggestions.

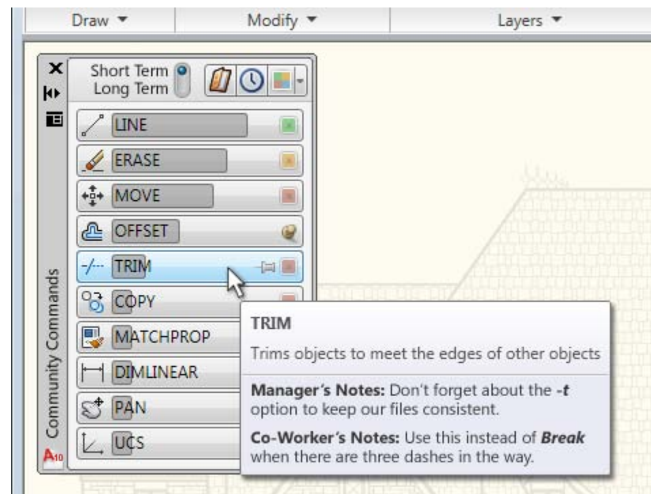

Figure 14. CommunityCommands UI.

We have included a command category selector for when a user would like to focus their learning on a particular area of the program. For AutoCAD this list contains items such as File I/O, Rendering, and 3D Modeling. By default commands for all areas are shown. The category of the command is reflected in the color of the close box or push pin.

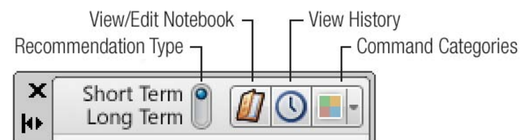

Figure 15. Overall system UI elements.

The bar on the individual command buttons (Figure 16) represents the relevance of the command to the user's current workflow. While doing "long term" recommendations the most relevant overall items will always be at the top of the list. While in "short term" mode, the most relevant commands will be at the top of list, and the length of the bar will indicate how closely related the command is to the commands the user is currently using.

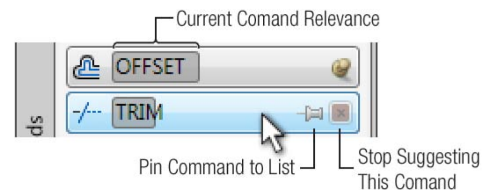

Figure 16. Individual suggestion UI elements.

To support the design goal of minimal cost for poor suggestions, each command button contains a close button to remove the command from the list, and prevent it from being suggested in the future. To keep a command the pin can be clicked. Clicking on the pin again unpins the item and reveals the close box.

To minimize the screen space used, yet remain visible for ambient awareness, the palette can be docked in the unused space beside the command line in the bottom right corner of the application (Figure 17).

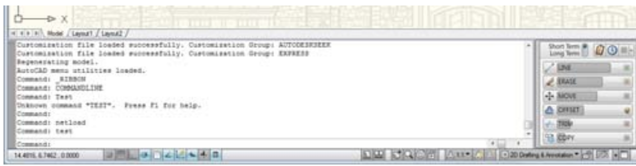

Figure 17. CommunityCommands UI docked in the unused space beside the command line interface.

Initial User Feedback

We demonstrated a working prototype of the CommunityCommands UI palette to both of the users described above during our on-site survey, and it was very well received. Both participants stated that a constantly updating, task sensitive recommendation list would be beneficial. Also, they agreed that being able to specify recommendations on subtopics (e.g., modeling, annotation, etc.) would allow them to focus on specific areas of improvement. Reviewing past recommendations or pinning current recommendations (as a reminder) was considered important as it would provide flexible opportunity to explore new commands when the user had spare time. We suggested the UI palette be placed beside the command line interface and this was well received, as this space was wasted, even on their 21 inch monitor.

DISCUSSION & FUTURE WORK

Our study has shown that the item-based recommendation algorithm works well over previous research approaches. However, there are a number of additional issues to consider. It may be important to have some of the recommendations on the list be unexpected or weird – the assumption being that you want to recommend items the user would not naturally progress to and instead expose a new or rare cluster of functionality. While both of these are useful options, it must be balanced with the number of recommendations a user is willing to browse at any given time. A second issue to consider is the addition of new commands to the software application – this typically happens with new releases. Software vendors may want to “push” commands into the recommendation list since essentially these commands are experiencing a “cold start” as no users have used the commands yet. Entry points could be determined by product designers or by using the beta-customer testing usage patterns that typically precede a software release.

The recommendation algorithm could be more adaptive by looking at the adoption rate of commands being suggested. This specific information could be fed back into the recommender where we could overweight the commands that are more highly adopted. In addition, as often found in other recommender based systems, we could allow users to provide explicit feedback on the quality of the individual recommendations and feed this into the algorithm.

A limitation of the current design is that once a command is used once, it is never recommended again. Here we could modify the algorithm to reintroduce the command after looking at how frequently and long ago it was used. The user interface could also allow users to dismiss a recommended command for the short term or forever.

Future research could investigate recommending higher level tasks to the user. These tasks would contain a collection of commands and sequences of workflows. Similarly, future improvements could inspect short sequences of commands and recommend a single advanced command that could replace the sequence or even an alternative workflow strategy. Also, instead of recommending a command by presenting the command name, we could present images of the effect the command does onto application data. Lastly, a longitudinal study using the CommunityCommands UI and recommendation algorithm would be useful to measure long-term command adoption patterns.

CONCLUSION

With CommunityCommands we have adapted modern recommender collaborative filtering algorithms together with rule-based domain knowledge to address the learning problem in complex software applications. To test the algorithms offline we developed the k-tail evaluation system and then conducted a comprehensive user study by generating personalized recommendations for a group of real users. Results showed a 2.1 times improvement in the number of good recommendations over previous research. The ambient user interface was designed to present the recommendations to the user, while satisfying our outlined design principles, and considering the lessons learned from visiting our participants.

ACKNOWLEDGEMENTS

The authors thank Joe Konstan for his valuable advice, Chris Willets for collecting the CIP data, and Ramtin Attar for his help with AutoCAD.

REFERENCES

1. Abran, A., Khelifi, A., Suryn, W. and Seffah, A. Usability Meanings and Interpretations in ISO standards. *Software Quality Journal*. 11(4): 325-338.
2. Adomavicius, G., Tuzhilin, A. (2005) Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Transactions on Knowledge and Data Engineering* 17(6): 734-749.
3. Apple Computers iTunes (http://www.apple.com/itunes) Active March 2009.
4. Benyon, D. (1993) Adaptive Systems: a Solution to Usability Problems. *Journal of User Modelling and User-Adapted Interaction*, 3(1): 65-87.
5. Claypool, M., Le, P., Waseda, M., Brown, D. (2001) Implicit interest indicators, in *ACM IUI*.p.33-40.
6. Cutrell, E., Czerwinski, M., and Horvitz, E. (2001) Notification, Disruption, and Memory: Effects of Messaging Interruptions on Memory and Performance. *ACM CHI*. p.263-269.

7. Czerwinski, M., Cutrell, E., and Horvitz, E. (2000) Instant messaging: Effects of relevance and time. *People and Computers XIV: Proceedings of HCI.* p.71-76.
8. Davison, B., and Hirsh, H. (1998) Predicting Sequences of User Actions. *the AAAI/ICML Workshop on Predicting the Future: AI Approaches to Time-Series Analysis.* AAAI. p.5-12.
9. Dent, L., Boticario, J., McDermott, J., Mitchell, T., and Zabowski, D. (1992) A Personal Learning Apprentice. *AAAI.* p.96-103.
10. Farzan, R., and Brusilovsky, P.. (2006) Social Navigation Support in a Course Recommendation System. *Proceedings of Hypermedia and Adaptive Web-Based Systems.* p.91-100.
11. Fischer, G. (2001). User Modeling in Human–Computer Interaction. *UMUAI.* 11(1-2): p.65-86.
12. Fogarty, J., Hudson, S. E., Atkeson, C. G., Avrahami, D., Forlizzi, J., Kiesler, S., Lee, J. C., and Yang, J. (2005). Predicting human interruptibility with sensors. *ACM TOCHI.* 12(1): p.119-146.
13. Gajos, K. Z., Everitt, K., Tan, D.S., Czerwinski, M., and Weld D.S. (2008) Predictability and Accuracy in Adaptive User Interfaces. *ACM CHI.* p.1271-1274.
14. Grossman, T., Fitzmaurice, G., and Attar, R. (2009) A Survey of Software Learnability: Metrics, Methodologies and Guidelines. *ACM CHI.*
15. Hermens, L.A., and Schlimmer, J.C. (1994) A Machine-Learning Apprentice for the Completion of Repetitive Forms. *IEEE Expert* 9(1): p.28-33.
16. Hill, W., Stead, L., Rosenstein, M., and Furnas, G. (1995) Recommending And Evaluating Choices in a Virtual Community of Use. *ACM CHI.* 194 - 201.
17. Horvitz, E. (1999) Principles of Mixed-Initiative User Interfaces. *ACM CHI.* p.159-166.
18. Hsi, I. and Potts, C. (2000). Studying the Evolution and Enhancement of Software Features. *IEEE SM*. 143-151.
19. Hsu, Mei-Hua. (2008) A Personalized English Learning Recommender System for ESL Students. *Expert Systems with Applications*, 34(1): 683-688.
20. Igarashi, E., and Hughes, J.F. (2001) A Suggestive Interface for 3D Drawing. *UIST.* p.173-181.
21. Jones, Karen Spärck (1972) A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 60(5): p.493-502.
22. Knabe, K. (1995). Apple guide: a case study in user-aided design of online help. *ACM CHI.* p.286-287.
23. Linden, G., Smith, B., and York, J. (2003) Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing*, 7(1): p.76-80.
24. Linton, F., and Schaefer, H. (2000) Recommender Systems for Learning: Building User and Expert Models through Long-Term Observation of Application Use. *UMUAI.* 10(2-3): p.181-208.
25. Linton, F., Joy, D., Schaefer, H., and Charron, A.. (2000) OWL: A Recommender System for Organization-Wide Learning. *Educational Technology and Society*, 3(1): 62-76.
26. Liu, J., Wong, C.K., and Hui, K.K. (2003) An Adaptive User Interface Based On Personalized Learning. *Intelligent Systems*, 18(2): 52-57.
27. McCrickard, S.D., Czerwinski, M., and Bartram, L. (2003) Introduction: design and evaluation of notification user interfaces. *Int. J. Human-Computer Studies* 8(5): p.509-514.
28. McGrenere, J., Baecker, R. M., and Booth, K. S. (2007) A Field Evaluation of An Adaptable Two-Interface Design for Feature-Rich Software. *TOCHI.* 14(1): #3
29. McNee, S.M., Kapoor, N., and Konstan, J.A. (2007) Don't look stupid: avoiding pitfalls when recommending research papers. *CSCW.* p.171-180.
30. Miller B. N., Albert, I., Lam, S. K., Konstan, J. A., Riedl, J. (2003) MovieLens unplugged: experiences with an occasionally connected recommender system, *ACM IUI*. p.263-266.
31. Mitchell, J., and Shneiderman, B. (1989). Dynamic versus static menus: an exploratory comparison. *SIGCHI Bull.* 20(4): p.33-37
32. Owen, D. (1986), Answers first, then questions. In: D. A. Norman and S. W. Draper (eds.), *User-Centered System Design, New Perspectives on Human-Computer Interaction*. p.361-375
33. Paymans, T.F., Lindenberg, J. and Neerincx, M. (2004). Usability trade-offs for adaptive user interfaces: ease of use and learnability, *ACM IUI*. p.301-303
34. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., and Riedl, J. (1994) GroupLens: An Open Architecture for Collaborative Filtering of Netnews. *CSCW.* p.175-186.
35. Salton G., McGill M.J., (1983), Introduction to Modern Information Retrieval'. McGraw-Hill, New York
36. Sarwar, B., George, K., Konstan, J., and Riedl, J. (2000) Analysis of Recommendation Algorithms for E-Commerce. *ACM Electronic Commerce.* p.158 - 167.
37. Shardanand, U., and Maes, P. (1995) Social Information Filtering: Algorithms for Automating Word of Mouth". *ACM CHI*. p.210 - 217.
38. Shen, J., Li, L., Dietterich, T.G., and Herlocker J.L. (2006) A Hybrid Learning System for Recognizing User Tasks from Desktop Activities and Email Messages. *IUI.* p.86-92
39. Shneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. *Computer.* 16(8): 57-69.
40. Shneiderman, B., (1997) Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley Longman Publishing Co., Inc.
41. Witten I., Cleary, J., and Greenberg, S. (1984). On frequency-based menu-splitting algorithms. *Intl. Journal of Man-Machine Studies* 21(2): 135-148.
42. Xiao, J., Stasko, J. and Catrambone, R. (2004). An Empirical Study of the Effect of Agent Competence on User Performance and Perception. *Joint Conference on Autonomous Agents and Multiagent Systems*. p.178-185.