Context-Aware Integrated Development Environment Command Recommender Systems

Marko Gasparic, Tural Gurbanov, Francesco Ricci
Free University of Bozen-Bolzano
Piazza Domenicani, 3, 39100 Bolzano, Italy

***Abstract*—Integrated development environments (IDEs) are complex applications that integrate multiple tools for creating and manipulating software project artifacts. To improve users' knowledge and the effectiveness of usage of the available functionality, the inclusion of recommender systems into IDEs has been proposed. We present a novel IDE command recommendation algorithm that, by taking into account the contexts in which a developer works and in which different commands are usually executed, is able to provide relevant recommendations. We performed an empirical comparison of the proposed algorithm with state-of-the-art IDE command recommenders on a real-world data set. The algorithms were evaluated in terms of precision, recall, F1, k-tail, and with a new evaluation metric that is specifically measuring the usefulness of contextual recommendations. The experiments revealed that in terms of the contextual relevance and usefulness of recommendations the proposed algorithm outperforms existing algorithms.**

I. Introduction

Software development tools can affect the efficiency and the quality of software construction [1]. Integrated development environments (IDEs) are popular applications that serve the needs of a large and diverse user population, by bringing together multiple tools to create and manipulate software project artifacts. A generic IDE user is not supposed to access the full provided functionality if that is not useful to accomplish the task at hand. Nevertheless, it is important to try to increase the breadth of the used functionality, especially of the less skilled users, since the lack of knowledge may prevent the exploitation of useful functions [2]. In fact, in our previous work, we conducted a user study and performed a number of interviews which confirmed that even professional software developers are willing to learn new IDE functionality and would like to use an application that could help them to achieve this goal [3][4].

To improve IDE users' knowledge and usage of the available functionality, command recommender systems (RSs) have been proposed [5]. Commands are shortcuts or menu buttons that execute a certain function, and they can be recommended as: *opportunistic* suggestions, which refer to recommendations that are relevant in the specific situation when the recommendation is presented; and *global* suggestions, which refer to recommendations that are based on the long-term command usage history and are supposed to be relevant in general for the future activity of the user. In our work, we focus on *global* IDE command recommendations, which should take into account the typical contexts in which a particular developer works and are useful in these contexts. We note that the actual delivery of the recommendations, i.e., presentation and timing, are out of scope of this paper, but we refer an interested reader to some of our previous publications, in particular: [4] and [6].

We model the context with contextual factors that describe the environment in which an IDE user executed a command, the temporal perspective of the execution, what was the user's activity before and during the execution, and what project artifact she interacted with [7]. The exploitation of contextual information to generate *global* command recommendations is novel, since existing algorithms, which are either based on command popularity, collaborative filtering, or command discovery patterns [5], do not take into account any contextual information. Even CoDis [8], which is a recent algorithm that combines the information about the co-occurrences of commands in the same session with the command discovery patterns, cannot be considered as context-aware, when it generates *global* recommendations. We conjecture that without taking into account contextual information, the usefulness of the recommendations to the recipient's work is coincidental.

Imagine an illustrative scenario, which we also observed in the data set used in the evaluation. By the end of the first week of usage of Eclipse IDE (see http://www.eclipse.org), a novice programmer executed 13 distinct commands, such as Paste, Build All, and Navigate Back. CoDis recommends to this user: Save, Save As, Run, Quick Fix, and Open Browser. Conversely, recommending the most popular commands, the user would learn: Save, Run, Undo, Copy, and Content Assist (also called "autocomplete"). Naturally, these commands are very useful for any IDE user, and especially for that novice, who is using Eclipse for editing simple Java code. However, these are the most basic functions provided by any IDE and the user will surely learn them quickly, also by herself. On the other hand, the context-aware RS, based on the algorithm proposed in this paper, which we named CNTX, would suggest simple, but less obvious commands, namely: Save All Files, Run Last, Reset Perspective, Go To Line Start, and Show Marketplace Wizard. Moreover, after some more time, when the novice user has already executed 41 distinct commands, our algorithm would also recommend two simple commands: Collapse All and Go To Text Start, and three advanced commands: Execute, Open Run Configurations, and Incremental Find. But, the other two algorithms would still be focused on simple commands, such as Redo, Go To Previous Word, Select Next Word, Select Previous Word, and Show File Properties.

Hence, as this example illustrates, we believe that there

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research - New Ideas

is a great potential in context-aware command recommender systems, and this paper discusses their main advantages and also their limitations.

In the rest of the paper, we first present the basic characteristics of existing algorithms for generating *global* IDE command recommendations (Sec. II). Then we present the proposed CNTX algorithm (Sec. III) and the offline evaluation method (Sec. IV). The results (Sec. V) show that by using context, the recommender is able to identify and suggest commands that are less likely to be discovered without a RS and are more useful for the working context than the commands recommended by existing algorithms. At the end, we draw the conclusions of our research results and outline the plans for the future work (Sec. VI).

The main contributions of this paper are: a novel context-based algorithm for recommending IDE commands, a detailed discussion of the algorithm implementation and problems resolution, a novel context-based metric for estimating the potential usefulness of the recommended commands, and an offline evaluation of the existing algorithms.

II. Related Work

Murphy-Hill et al. [5] studied eight command recommendation algorithms, namely: Most Popular, which recommends most frequently executed commands; Most Widely Used, which recommends commands that are used by the largest number of users; Item-based Collaborative Filtering (CF) and User-based CF; which recommend commands that are most similar to those already used by the user, or used by similar users; and Advanced Discovery, Most Popular Discovery, Item-based CF with Discovery, and User-based CF with Discovery, which are based on sequential pattern mining that is combined with basic collaborative filtering algorithms or popularity-based algorithms. To evaluate these eight algorithms, Murphy-Hill et al. performed offline and online evaluations.

In the offline evaluation, they applied the k-tail evaluation method suggested by Li et al. [9], which measures the capability of an algorithm to predict commands that the user eventually used. Advanced Discovery algorithm achieved the highest score, which was noticeably higher than the scores of Most Popular, User-based CF, and Item-based CF, and relatively similar to the scores of other algorithms. In the special k-tail evaluation, where a command was treated as being discovered only if it was used *multiple times* or in *multiple sessions*, the highest score was achieved by User-based CF with Discovery.

In the online evaluation, Murphy-Hill et al. recruited four experts and nine novices. The participants were asked to rate the usefulness and novelty of recommendations. The results show that for the novices, User-based CF, Most Popular, and Item-based CF with Discovery generated the largest proportion of useful and novel recommendations. For the experts, the Item-based CF with Discovery algorithm was the only algorithm that performed relatively well.

Recently, Zolaktaf and Murphy [8] proposed CoDis, which is an algorithm based on the analysis of command discovery patterns and co-occurrence of command executions in work-sessions. In an offline study, CoDis outperformed all other algorithms, in terms of the k-tail metric, when the number of observed top-N recommendations is larger than 2; for $N \in \{1, 2\}$, User-based CF with Discovery outperformed CoDis.

It is worth noting that the aforementioned algorithms were designed to accurately predict which commands an IDE user will discover and start using autonomously. Nevertheless, we argue in this paper that it is also important for a recommendation algorithm to identify commands which a recommendation recipient can and will use during the work, but are not likely to be discovered without the help of a RS. We think that in such a case an IDE command RS is especially valuable. Hence, we conjecture that by employing information about the contexts in which a developer works and in which different commands are usually executed, it is possible to provide such a type of recommendations.

III. CNTX: Context-Based IDE Command Recommendation Generation Algorithm

We apply a probabilistic model to generate personalized, context-aware, novel, and useful command recommendations. The recommendation score of a command a for a user u is defined to be $P(a|u)$, which is the probability of observing the usage of a, assuming that u knows a. Commands with the highest recommendation score should be recommended first (top-N). We assume, as in [10], that the RS users are only interested to receive recommendations of commands that they are not aware of, i.e., have never been executed by the user.

The input data for the recommendation algorithm was collected by logging the IDE interactions of first year bachelor students at the Free University of Bozen-Bolzano, during the first ten weeks of the Introduction to Programming course. The data collection was completely anonymous. The data set contains 199,220 command execution records. Each record is a tuple $< u, a, t, c >$, where t is the timestamp and c is the context in which u executed a. Overall, we detected 113 different user identifiers and 219 different commands. Each context is a set of values for different contextual factors. We used the context model proposed by Gasparic et al. [7], which consists of eleven contextual factors, namely: *type*, *length*, and *complexity of the artifact under development*, *current* and *previous activity*, *time of the day*, *day of the week*, *IDE instance*, *active perspective*, *opened user interface elements*, and *user interface element with focus*. An example of the interaction history log is presented in Tab. I; the columns following "timestamp" refer to the contextual factors.

A user u can be described by a set of contexts C_u that were detected when she executed commands. The probability $P(a|u)$, that a can be executed by u, if she knows a, is estimated as $P(a|C_u)$, which is the probability to observe the execution of a in the population of users that know and use a, given a set of contexts in which u worked. Assuming that the contexts are alternative, we define $P(a|u)$ as the average

TABLE I
USER-IDE INTERACTION DATA LOG EXAMPLE.

command_id	user_id	timestamp	artifact_type	...	ui_with_focus
undo	A	1493241234	.html	...	editor
collapseAll	A	1493590321	.java	...	editor
lineStart	B	1493628126	.java	...	console
...	...	...	...	...	...
undo	X	1493708129	.java	...	expressions

value of the probabilities $P(a|c)$ to observe the execution of a, given the u's context c. Hence, the scoring function is defined as follows:

$$score(u, a) = P(a|u) = P(a|C_u) = \frac{1}{|C_u|} \sum_{c \in C_u} P(a|c)$$

To estimate $P(a|c)$, for each command a, we train a regression model [11] (for each command a) on a set of contexts extracted from the IDE-interaction history logs of the users who are using a. The regression model is trained to predict a boolean variable indicating whether a will be executed when a specific context c is detected. The score calculated by the regression model for the context c is considered as the probability $P(a|c)$ to observe a given c.

To train a regression model, categorical variables should be transformed into continuous [12]. Since the values of our contextual factors are nominal, i.e., cannot be ordered, we transform each value of a contextual factor into a binary variable. For example, if the contextual factor *type of the artifact under development* can have ".html" and ".java" values (see "artifact_type" column in Tab. I), two binary variables are created: "artifact_type::.html" and "artifact_type::.java"; and the values of "artifact_type::.html" are equal to 1 for the rows where "artifact_type" contains ".html" and 0 otherwise (see "artifact_type::.html" column in Tab. II).

Moreover, since regression models trained with few observations are unreliable, we limited the set of recommendable commands to those for which we have at least m observations. We tested different parameter values and finally set m to 5 since this value yields the best results in the offline evaluation reported in this paper.

Furthermore, in order to improve the accuracy of the regression model and to compress the transformed data set, which contains a large number of potentially correlated columns, we performed "dimensionality reduction" [11]. We applied Latent Semantic Indexing (LSI) [13], which uses Singular Value Decomposition to identify a linear subspace of the full *features'* space that captures the largest part of the variance in the collection. If we consider the generated binary variables as *features*, we could have applied LSI directly, however, since the number of rows, i.e., observations of the command executions, is large, we trained LSI with a random sample of observations from the interaction history log. This, without losing the quality, ensures that the calculations can be performed in minutes, and not in hours or days. By setting the number of *features* to 35 and the maximum size of the sample to 45,000 command execution records, LSI explains 99% of the training data variance, even though the initial data set contains more than 1,000 *features*. We used this configuration also in the offline evaluation.

The class imbalance problem was the last issue to solve, before we could train the regression model. The number of executions of any command a is much lower than the number of executions of non-a commands, i.e., commands other than a. Consequently, any trained model learned on the full data set would be biased to predict that command a is never executed, regardless of the context. To solve this problem, we adopted "random undersampling" [14]: for each command regression model, we included in the training data all the observations of the command a execution and an equal number of randomly picked observations of executions of the other commands.

Finally, for each command, the training data, represented with the selected latent *features*, was used to train a regression model that predicts $P(a|c)$. In our experiments, we considered three linear regression models: Ridge, AdaBoost with a linear loss function, and AdaBoost with an exponential loss function [15]. The Ridge regressor is based on the linear least squares loss function and uses L^2-norm regularization to avoid model's overfitting. An AdaBoost regressor is a meta-estimator that begins by fitting a regressor on the original data set and then fits additional copies of the regressor on the same data set, with adjusted weights assigned to instances, which are larger for instances where the regressor is wrong. In that way, subsequent regressors focus more on difficult cases. The Ridge model has been used as a component of the AdaBoost models.

To identify the most suitable regression model for a command, we conducted a five-fold random cross-validation test, over a set of candidate models. The model with the highest average score, according to the R^2 metric [11] was selected. The R^2 metric, also called coefficient of determination, is a regression score function that measures how well future samples are likely to be predicted by the model.

IV. OFFLINE EVALUATION METHOD

We conducted an offline experiment to evaluate RS algorithms. With an offline experiment, one can efficiently compare a wide range of algorithms and estimate their quality, by using some proxy metrics, before the RS is deployed to real users. Typically, these types of experiments are simple and inexpensive, if compared to user studies and online evaluations, as they require no interaction with real users. However, offline experiments can only answer a narrow set of research questions, since the data sets collected before the deployment of the RS cannot be used for measuring the RS's effect on user behavior in a real-world setting [16].

In our empirical evaluation, we have compared CNTX with the algorithms listed in Sec. II. We used the data set described in Sec. III. Since the participating students were enrolled in an introductory course, they were expected to have none or very basic programming knowledge. Consequently, the assigned tasks were focused on Java syntax. To allow a fair comparison and meaningful analysis of the evolution of the metrics over the weeks, we limited the testing set to 43 students who used

TABLE II
BINARIZED INTERACTION DATA LOG EXAMPLE.

command_id	user_id	timestamp	artifact_type::.html	artifact_type::.java	...	ui_with_focus::editor	ui_with_focus::console	...	ui_with_focus::expressions
undo	A	1493241234	1	0	...	1	0	...	0
collapseAll	A	1493590321	0	1	...	1	0	...	0
lineStart	B	1493628126	0	1	...	0	1	...	0
...	...	...	...	...	...	...	...	...	...
undo	X	1493708129	0	1	...	0	0	...	1

the IDE for at least five weeks. For each user and with each algorithm, we generated the top-5 recommendations, for each different week of usage, by using the data observed in the past. Previous analyses benchmarked the proposed algorithms on a larger number of recommendations. However, in a real-world setting only the very first recommended commands are usually browsed by the users.

We must observe that IDE command usage history logs show only the actions that have been performed by the users, and it is unclear why certain actions—in particular, executions of specific commands—have not been performed. For instance, when a command is never observed, is this because the user is not aware of its existence or has she deliberately decided not to use it? Secondly, IDE command usage history logs contain commands that have been discovered by the users without the help of a RS, and traditional offline evaluation metrics show how well the recommender prediction model identifies commands that the user eventually used, but they do not fully show whether the recommender identifies new and useful commands. For instance, the k-tail evaluation method [9] is measuring the accuracy of the recommendation algorithm in suggesting commands autonomously discovered by the user. That is why we also introduce here a novel evaluation metric that measures the usefulness of the recommendations by considering the contexts in which users will actually execute the suggested commands.

A. Usage Prediction

In our experiments, to measure the accuracy of the algorithms' predictions, we considered $precision$, $recall$, and $F1$ metrics [16]. These are popular RSs evaluation metrics. Moreover, we also adopted the k-tail evaluation method, to better relate our analysis to previous studies.

To perform the k-tail evaluation and to compute $precision$, $recall$, and $F1$, the data must be split into training and testing sets. The recommendations are generated by training the predictive model on the observations from the training set. Standard evaluation is based on the assumption that the commands included in the testing set, which have been used by the users, are good recommendations: higher similarity between the recommendations and the commands in the testing set indicates higher quality of the algorithm.

$Precision$ measures how many items in the list of recommendations are relevant, while $recall$ shows how many of all the relevant items are included in the list of recommendations. Since there is usually a trade-off between $precision$ and $recall$, it is also common to measure $F1$, which averages (harmonically) the two metrics:

$$precision = \frac{|TP|}{|TP| + |FP|} \qquad recall = \frac{|TP|}{|TP| + |FN|}$$

$$F1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

where TP represents the set of correct recommendations, often called "true positives", FP represents the set of incorrect recommendations, often called "false positives", and FN represents the set of correct items that were not recommended, often called "false negatives". The highest possible score of $F1$ is 1. It is obtained if the algorithm recommends exactly all the relevant items.

To calculate $precision$, $recall$, and $F1$ and to observe their evolution over time, we split the command usage logs in ten weeks. We generated a set of recommendations every week, for the first 9 weeks. The data collected until Wednesday 6am, which is between the weekly assignment submission deadlines and laboratory exercises, was used as the training set. The data collected in the following one week was used as the testing set, where the set of relevant recommendations is composed of the commands not already present in the training set.

Conversely, in k-tail testing procedure, the training set consists of the logs of the command executions that occurred before the user started using the last k commands, while the testing set consists of the commands that were executed for the first time after that moment. The testing set is used to calculate $hit\text{-}ratio$ (HR):

$$HR = \frac{1}{|U|} \sum_{u \in U} hit_u$$

where U is the set of all users u and hit_u is a binary variable equal to 1 when the recommended commands include at least one of the last k commands and equal to 0 otherwise. In our experiment, k is equal to 1, as in [5] and [8], which means that the testing set contains only the last discovered command.

B. Contextual Relevance

In this paper, we propose to relate the relevance of a command recommendation to the contexts in which the command has been executed. In fact, we know that the contexts in which the command is executed are the contexts in which the command is useful. In other words, in ideal conditions, the most relevant command for a given context is the one that has been observed most often in this context. But, since contexts are presented in a high dimensional space, it is very unlikely

to observe for a target user u exactly the same contexts that are observed in the training set. Hence, we use a notion of similarity between the target contexts of the user and the contexts in which the command has been executed, to identify the most relevant recommendations. In particular, the relevance of a command a to a user u can be defined as the average similarity between the contexts in which the command was used—by the users that know and use it—and the contexts in which the recommendation recipient worked during the period that is included in the testing set. The more similar the two contexts are, the more relevant a is to u. The contextual relevance is calculated as follows:

$$rel(a,u) = \frac{1}{|C(u)|} \sum_{c_u \in C(u)} \frac{\sum_{c_a \in C(a)} sim(c_u, c_a)}{|C(a)|}$$

where $C(a)$ is a set of contexts in which a has been executed, $C(u)$ is a set of contexts in which u was executing commands, $sim(c_u, c_a)$ is the similarity between two contexts, and c_a and c_u are a context where the command a was executed and the target context of user u, respectively. We use "cosine similarity" since we are interested in the correlation between features of the different contexts [17].

The context-aware usefulness metric $AR@N$ calculates the average relevance of the top-N recommendations for each user $u \in U$. If $Rec@N_u$ is the set of top-N recommended commands, then $AR@N$ can be defined as follows:

$$AR@N = \frac{1}{|U|} \sum_{u \in U} \frac{\sum_{a_r \in Rec@N_u} rel(a_r, u)}{N}$$

Higher values indicate that more useful commands are recommended by the algorithm. Moreover, $AR@N$ penalizes the algorithms that generate less than N recommendations.

To calculate the weekly contextual relevance of the top-5 commands for each user, i.e., $AR@5$, we used the interaction history logs in the same way as for the calculation of $precision$, $recall$, and $F1$. But, instead of matching recommended commands to commands in the testing set, we match the observed contexts of the users, which were detected during the week that followed the recommendation generation milestone, with the contexts in which the recommended commands can be effectively used.

We note that the source code of $AR@N$ and CNTX is available at https://gitlab.inf.unibz.it/tural-gurbanov/ide_rs.

V. Results

The algorithms evaluated in our study generate very different recommendations, as can be seen in Fig. 1. Here, as an example, the word cloud created from the recommendations produced by the CNTX and the Most Widely Used algorithms are shown. It can be seen that the commands recommended and their frequencies are dissimilar.

The results of the k-tail evaluation (Tab. III) show that Advanced Discovery has the highest $hit\text{-}ratio$, as in [5]. Apart from that, our results are in a disagreement with the previous studies. For instance, Most Popular has higher $hit\text{-}ratio$ than Most Widely Used, and User-based CF with Discovery and

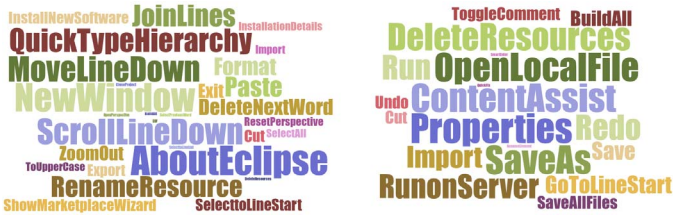

Fig. 1. Recommendation cloud of CNTX (left) and Most Widely Used (right) algorithms in the fifth week of the study.

TABLE III
Results of k-tail evaluation.

Algorithm	$Hit\text{-}ratio$	Algorithm	$Hit\text{-}ratio$
Most Popular	9.3%	Most Widely Used	4.7%
Item CF	11.6%	User CF	4.7%
Advanced Discovery	18.6%	Most Popular Discovery	14%
Item CF + Discovery	7%	User CF + Discovery	4.7%
CoDis	4.7%	CNTX	2.3%

CoDis have a very low $hit\text{-}ratio$. Interestingly, the lowest $hit\text{-}ratio$ was achieved by our algorithm: 2.3% means that only 1 out of 43 students autonomously discovered and executed a command that would have been recommended to them by the context-aware algoirthm. Hence, according to the results of the k-tail evaluation, we expect that the proposed algorithm will recommend commands that the users would find really novel.

The average $recall$, $precision$, $F1$, and $AR@5$, of the considered algorithms during nine weeks, are shown in Fig. 2. These results show that $recall$ grows over time. The main reason for this is that there are less and less commands in the testing set every week. For the same reason, $precision$ tends to decrease over time. The only exceptions are offered by the algorithms based on discovery patterns, which do not provide any recommendations in the first weeks, thus, their $precision$ is initially 0 and later starts increasing. In Fig. 2, it can be seen that the ranking of the algorithms according to $F1$ and $precision$ is basically identical. Similarly to what we observed and discussed in the k-tail evaluation, we can conclude that CNTX algorithm would recommend only a small set of the commands that the students would discover autonomously during the next week. The only algorithm that would provide more novel recommendations is User-based CF.

As the number of the commands that can be recommended decreases, also the number of the recommendable commands with the contexts similar to the users' contexts decreases over time. Consequently, the $AR@5$ scores tend to decrease with time. Nevertheless, considering this metric, the proposed CNTX algorithm outperforms other algorithms in every week. Furthermore, we can see that the popularity-based algorithms perform worse than how they performed with respect to $F1$. But the ranking of the other algorithms remains almost the same as for $F1$. Still, it seems that the popularity-based algorithms perform very well at the beginning, which means that recommending popular commands to novices is reasonable. This result is also in agreement with the results of the online

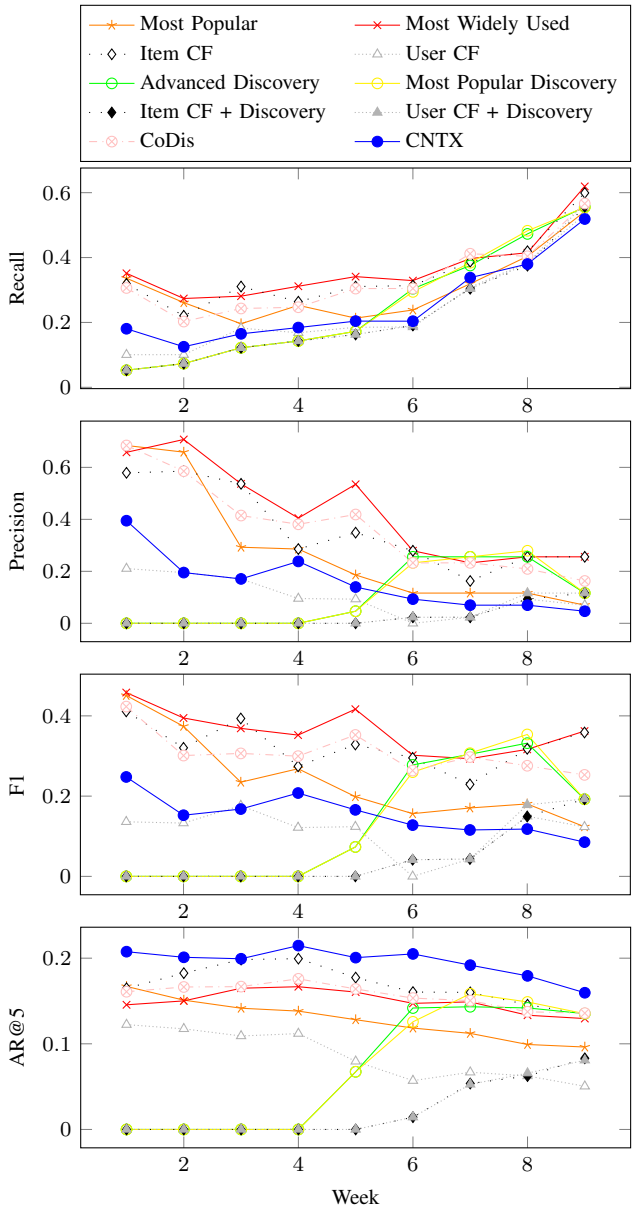

Fig. 2. Recall, precision, F1, and AR@5 values of the algorithms per week.

evaluation performed by Murphy-Hill et al. [5]. However, after the number of known commands by the users increases, personalized algorithms tend to outperform popularity-based algorithms, while context-aware algorithms have an even greater potential.

VI. Conclusions and Future Work

In this paper, we have presented a novel IDE command recommendation algorithm that provides recommendations by taking into account the contexts in which a particular software developer works and the contexts in which different commands are usually executed. The algorithm aims at suggesting the commands that a recommendation recipient can and will use during the work, but which are not likely to be discovered without the help of the RS. Since traditional metrics, such as *precision*, *recall*, $F1$, and *hit-ratio*, only measure the accuracy of predicting the commands autonomously discovered by the users, we also introduced a novel evaluation metric that measures the relevance of the recommendations in the contexts of the users. We have compared the proposed algorithm with a set of state-of-the-art algorithms, on a real-world data set. The experiments revealed that in terms of contextual relevance and recommendation usefulness, the proposed algorithm outperforms existing algorithms, while the traditional metrics score it lower. The explanation of this is that the proposed algorithm recommends more novel commands to the user, which are not likely to be discovered without the help of the recommender.

In the future, we plan to conduct an online experiment, on a smaller set of algorithms, to evaluate the RS's effect on the user behavior in a real-world setting. Moreover, we will consider designing and testing hybrid algorithms, with the aim to obtain the benefits of the popularity, collaborative filtering, and context-based algorithms, while decreasing their drawbacks.

References

[1] IEEE Comp. Soc., P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society Press, 2014.

[2] T. Grossman, G. Fitzmaurice, and R. Attar, "A survey of software learnability: Metrics, methodologies and guidelines," in *Conference on Human Factors in Computing Systems*, 2009.

[3] M. Gasparic, A. Janes, F. Ricci, and M. Zanellati, "GUI design for IDE command recommendations," in *International Conference on Intelligent User Interfaces*, 2017.

[4] M. Gasparic, A. Janes, F. Ricci, G. C. Murphy, and T. Gurbanov, "A graphical user interface for presenting integrated development environment command recommendations: Design, evaluation, and implementation," *Inf. Softw. Tech.*, 2017.

[5] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands," in *International Symposium on the Foundations of Software Engineering*, 2012.

[6] M. Gasparic and F. Ricci, "Should context-aware IDE command recommendations always be presented in-context or not?" in *Workshop on Awareness Interfaces and Interactions*, 2017.

[7] M. Gasparic, G. C. Murphy, and F. Ricci, "A context model for IDE-based recommendation systems," *J. Syst. Softw.*, vol. 128, pp. 200–219, 2017.

[8] S. Zolaktaf and G. C. Murphy, "What to learn next: Recommending commands in a feature-rich environment," in *International Conference on Machine Learning and Applications*, 2015.

[9] W. Li, J. Matejka, T. Grossman, J. A. Konstan, and G. Fitzmaurice, "Design and evaluation of a command recommendation system for software applications," *ACM Trans. Comput.-Hum. Interaction*, vol. 18, pp. 6:1–6:35, 2011.

[10] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *International Conference on Data Mining*, 2008.

[11] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer, 2014.

[12] S. Koranne, *Handbook of Open Source Tools*. Springer, 2010.

[13] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Soc. Inf. Sci.*, vol. 41, pp. 391–407, 1990.

[14] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. on Knowl. and Data Eng.*, vol. 21, pp. 1263–1284, 2009.

[15] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.

[16] A. Gunawardana and G. Shani, "Evaluating recommender systems," in *Recommender Systems Handbook*. Springer, 2015.

[17] X. Amatriain and J. M. Pujol, "Data mining methods for recommender systems," in *Recommender Systems Handbook*. Springer, 2015.