Predicting Future Developer Behavior in the IDE Using Topic Models

Kostadin Damevski, Hui Chen, David C. Shepherd, Nicholas A. Kraft, *Senior Member, IEEE*, and Lori Pollock

Abstract—While early software command recommender systems drew negative user reaction, recent studies show that users of unusually complex applications will accept and utilize command recommendations. Given this new interest, more than a decade after first attempts, both the recommendation generation (backend) and the user experience (frontend) should be revisited. In this work, we focus on recommendation generation. One shortcoming of existing command recommenders is that algorithms focus primarily on mirroring the short-term past,—i.e., assuming that a developer who is currently debugging will continue to debug endlessly. We propose an approach to improve on the state of the art by modeling future task context to make better recommendations to developers. That is, the approach can predict that a developer who is currently debugging may continue to debug *OR* may edit their program. To predict future development commands, we applied Temporal Latent Dirichlet Allocation, a topic model used primarily for natural language, to software development interaction data (i.e., command streams). We evaluated this approach on two large interaction datasets for two different IDEs, Microsoft Visual Studio and ABB Robot Studio. Our evaluation shows that this is a promising approach for both predicting future IDE commands and producing empirically-interpretable observations.

Index Terms—Command recommendation systems, IDE interaction data

1 Introduction

Since Microsoft's digital assistant Clippy, embedded within Microsoft Office for Windows, first uttered the words, "It looks like you're writing a letter...", command recommender systems have drawn the ire of the public. Based on a misunderstanding of research which suggested that users interact with their computers as they would humans [1], Clippy patronized a generation of users, leading not only to its removal from Office, but also to a near-complete absence of command recommendation systems from desktop software.

Much has changed since Clippy's demise in 2004. Desktop software has become increasingly complex, with applications such as AutoCAD, a 3D modeling system, growing to have thousands of commands [2]. Accordingly, users can be overwhelmed by the volume of available commands and thus have become more receptive to recommender systems and their suggestions, as demonstrated, for example, by a study in which over 700 AutoCAD users downloaded and utilized a command recommender system for over 30 days [3].

Similar to 3D drawing systems, Integrated Development Environments (IDEs), which are often extensible via plug-ins, have manifest command overload for many years. Developers who use these IDEs commonly exercise only a fraction of their capabilities [4], [5], while novices often feel overwhelmed by the breadth of commands [6]. Attempts have been made to reduce this complexity,—e.g., by filtering commands not relevant to the current context [7], [8] or suggesting commands to help increase developer fluency [9]—but no system has yet demonstrated promise for use in practice, partly due to the ineffectiveness of their underlying recommendation engines.

Towards increasing the use of command recommendations in an IDE, this paper presents a new modeling approach for recommendation generation. The technique leverages ideas from natural language topic modeling that can effectively model *future* task context, to predict the next set of tasks that a developer will perform in the IDE. While the state of the art software engineering recommendation systems rely on *past* task context, assuming that the future mirrors the short-term past, we model the future task context explicitly, enabling better quality recommendation generation for IDEs. For instance, our system can predict that, after searching code via grep, a developer is likely to explore the identified code via structural navigation rather than to simply continue searching. Our technique is practical in that it both addresses the noisy nature of IDE interactions [10], which are the basis for the prediction, and captures their streaming (time-based) nature, which has rarely been modeled in the past [11], [12], [13].

- *K. Damevski is with the Department of Computer Science, Virginia Commonwealth University, Richmond, VA 23284. E-mail: damevski@acm.org.*
- *H. Chen is with the Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY 11210. E-mail: huichen@ieee.org.*
- *D. C. Shepherd and N. A. Kraft are with ABB Corporate Research, Raleigh, NC 27606. E-mail: {david.shepherd, nicholas.a.kraft}@us.abb.com.*
- *L. Pollock is with the Department of Computer and Information Sciences, University of Delaware, Newark, DE 19350. E-mail: pollock@udel.edu.*

Manuscript received 13 Dec. 2016; revised 2 Aug. 2017; accepted 27 Aug. 2017. Date of publication 1 Sept. 2017; date of current version 9 Nov. 2018. (Corresponding author: Kostadin Damevski.)
Recommended for acceptance by S. Kim.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2748134

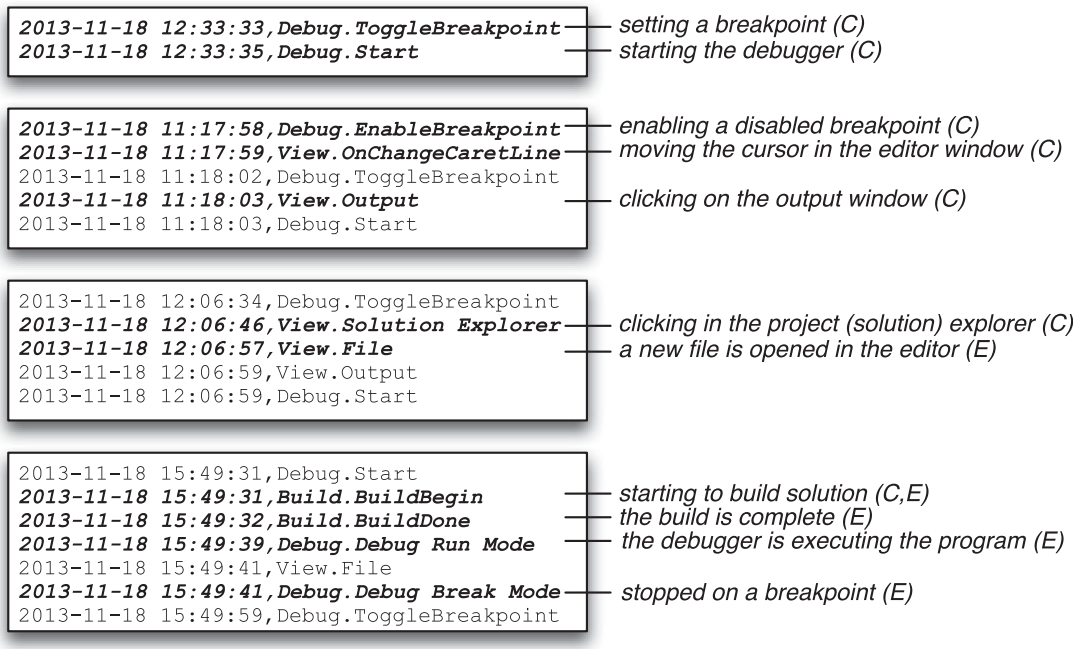

Fig. 1. Four interaction log snippets, showing a developer setting a breakpoint and starting the debugger. In interactions exhibiting this behavior, a variety of additional IDE *commands (C)* and *events (E)* are often interspersed with the direct commands that toggle a breakpoint and start the debugger.

A secondary contribution of this paper is that the constructed model is interpretable, allowing for analysis by researchers to determine the relationships between extracted high-level IDE user behaviors. Such analysis can be performed to examine individual tendencies, or as a holistic examination of IDE-developer interactions, which can guide understanding of how developers work in the field, contributing to empirical research in software engineering by validating findings from laboratory studies.

Specifically, we describe how to adapt and apply a popular probabilistic topic model, Latent Dirichlet Allocation (LDA) to IDE interaction data, which is in itself a novel (and effective) contribution. We also demonstrate why this model is appropriate for extracting high-level behaviors from low-level interaction datasets. We leverage a variant of LDA that takes time into account, called Temporal LDA, and describe a technique to train this model using historical interaction data and use it for online development task prediction during a developer's interaction with the IDE. Results on two large interaction datasets, one of nearly 200 Microsoft Visual Studio users and another of more than 25,000 ABB Robot Studio users, show that the prediction is accurate and interpretable, yielding insights on transitions between high-level behaviors of developers in the field.

The rest of the paper is organized as follows. We begin by describing in detail the features of the IDE interaction data, available at scale, which serves as input to our approach in Section 2. Next, in Section 3, we describe what makes topic models appropriate for interaction data, while Section 4 discusses the related work. In Section 5, we describe the details of Temporal LDA, while Section 6 shows the preliminary results in applying this model. Section 7 discusses the settings for some of the important parameters of our approach. Finally, Section 8 presents the conclusions and future work of this paper.

2 IDE Interaction Data Analysis Challenges

Interaction data captured in the field, during the daily work of professional developers, exhibits distinguishing characteristics that are common across different IDEs: (1) numerous kinds of interactions and a very large set of reasonable interaction sequences (or paths) exist; and (2) interactions are captured exhaustively and not extensively pre-filtered, thus the data reflects most clicks, key-presses and events in the IDE. The collected IDE interaction traces, in turn, exhibit the following characteristics, relative to other kinds of user interaction data (e.g., generic web page click-through datasets), which pose the main challenges in data analyses:

- *comprise of both events and commands*–events that reflect the state of the tool are captured in the same trace as commands that reflect user actions.
- *exponentially distributed*[1]–some events or commands dominate the trace (e.g., cursor movement commands), while most other commands occur relatively infrequently.
- *noisy*–the traces include spurious commands (or clicks), or unrelated events, that may not be important to the behavior of interest.
- *partially ordered*–the order of the events or commands is important in certain analyses, but not important in others, as often there are many different interaction paths in accomplishing a high-level task.

To illustrate the data characteristics and associated analysis challenges, Fig. 1 depicts real interaction traces from Microsoft Visual Studio, focusing on a single developer as she is starting to debug, by first setting a breakpoint followed by starting execution in the debugger. While each

1. Strictly speaking, the geometric distribution, which is the discrete counterpart of the exponential distribution is more accurate.

individual snippet shows this behavior, we observe that other commands and events are interspersed between the two commands for setting a breakpoint and debugging, and that sometimes even the order of these two commands may not matter (i.e., a breakpoint can be set after the debugger has been started and achieve the same goal).

3 Interaction Data Analysis via Topic Modeling

To our knowledge, topic models, which are commonly used to model natural language text, have not been applied to modeling interaction data, particularly IDE interaction data. The use of topic models is motivated by their capability to reduce dimensionality, which could be useful to raise the level of abstraction in IDE interaction data from low-level interaction messages to higher level developer behaviors. However, the possibility to apply these models relies on how difficult it is to adapt the modeling technique from natural language text to IDE interaction data. Here, we examine how in certain important ways, IDE interaction logs indeed mimic natural language text, which inspired our investigation into this modeling technique for command recommendation generation.

3.1 IDE Interaction Data and Natural Language Text

Both natural languages and interaction traces possess local regularity and repetition. In IDE interaction data, the number of interactions per unit of time, for a typical developer, during their daily work can be large. We differentiate the *number of interactions*, which is the count of messages in the interaction trace over a period of time, from the *number of interaction types*, which is the number of unique messages in the log. In our dataset, we commonly observe Visual Studio developers with 5,000 interactions per day. In addition, the number of interaction types that can be observed in a typical IDE is also large. We have observed approximately 1,200 interaction types in Visual Studio and nearly 4,000 in Robot Studio.

When we examine a smaller unit of the log, such as an hour of one developer's work, we find that the number of interaction types is small, consisting of usually highly regular and repetitive patterns. This is expected, as within a small period of time, a developer is likely focusing on a specific task and interacting with a small subset of the development environment which consists of relatively few interactions. In natural language text, similar words tend to occur within a paragraph or document, a notion sometimes referred to as "naturalness" [14]. We posit that this regular behavior and naming relations between the interaction types within smaller units of IDE usage time mimics the naturalness of text writing, and suggests that models used for analyzing natural language text might apply to IDE interaction data.

IDE interaction data exhibits naming relations such as synonymy and polysemy. A common goal of interaction trace logging is to capture as many different interactions as possible, that is, to be exhaustive (see Section 2). Thus, the trace often contains multiple messages that share meaning in a specific behavioral context. For instance, in Visual Studio's interaction log, opening the Find window using a shortcut produces a slightly different message compared to opening the same window by using the dropdown menu. Another example is shown in Fig. 1, where both ToggleBreakpoint and EnableBreakpoint have the same meaning in the same context. This is similar to the notion of synonymy in natural languages, where several words have the same meaning in a given context.

Similarly, IDE commands carry a different meaning depending on the task that the developer is performing. For example, an error in building the project after pulling code from the repository has a different meaning than encountering a build error after editing the code base. This characteristic is akin to polysemy in natural language, where one word has several different meanings based on its context. Thus, the natural language concepts of polysemy and synonymy are similar to these observed patterns in the log, where there could be several words that have one meaning and one word can have different meanings depending on context. All of these observations motivate the consideration of models used for natural language to be applied to IDE interaction data analysis.

3.2 Models, Documents, and Words

One goal of IDE interaction data analysis is to discover the developer's high level task behavior such as structured navigation, by discovering and interpreting sequences of lower level interaction events and commands [12]. A common approach to extracting and interpreting high level information from low level information in natural language is to use topic modeling to reduce the dimensionality of the natural language text. The most common techniques for topic modeling in natural language are Latent Dirichlet Allocation and Latent Semantic Indexing (LSI). In this paper, we leverage the regularity and naming relations of IDE interaction data described earlier to examine their applicability of topic modeling for reducing the dimensionality of interaction data.

LDA is a mixed membership generative model [15] that was independently developed for text [16] and for genetics [17], respectively. Mixed membership models imply that a *population* has multiple *subpopulations*, and an individual in a sample is assigned to *many* subpopulations. For text, the population can be a corpus of documents, and the subpopulations are indexed by using *topics*. For genetics, the populations can be corpora of genomes, and the subpopulations are indexed by using genotypes. Besides text and genetic data, LDA has also been applied to images, social networks, music, purchase histories, and source code to discover internal structures.

To apply LDA to IDE interaction data, we need notions of a *document* and a *word*. The interaction log messages, as a whole, can directly be used as the words. The notion of a document in IDE interaction data analysis can be formulated as any window of interaction events and commands executed over a contiguous time interval. By applying LDA to an IDE interaction window, we do not take into account the order of interactions, but instead study the words in the interaction data as a bag of words. We believe this is a reasonable approach given the amount of noise present in interaction datasets [10], and that many small scale tasks can be accomplished by varying command orders (as observed in Fig. 1). We discuss the potential impacts of a

bag of words approach and our choices for the various parameters in creating windows from the interaction log and in building the LDA model in Section 7.

4 Related Work

Today, large-scale interaction datasets are continuously gathered for numerous software engineering tools, most notably several modern IDEs[2] [18], [19], [20]. Analyzing IDE interaction data has presented opportunities for examining a broad range of software processes, including the examination of developers' preferred IDE views [21], commonly used (or under-used) refactoring commands [22], edit styles [23], and feature location [5]. Apart from use in empirical research, the most common goal of processing captured interaction data has been to guide improvements to the software development environment, prioritizing new features or suggesting improvements to existing features [24], [25].

Analyzing interaction traces has also been popular in other domains, such as generic business processes [26] and web click data [27], though complex software interaction traces differ from these in important ways (as discussed in Section 2). General software execution logs, which only loosely correspond to user interactions, have been the target of approaches to raise their abstraction level in order to improve their interpretability [28]. Iqbal et al. present a model that can detect breakpoints (i.e., transitions between tasks) [29], opportune moments to deliver notifications or recommendations to a user of a complex software application. The proposed breakpoint detection model is created using supervised learning, and requires a user annotated dataset of breakpoints specific to a particular application, while this paper's technique uses unsupervised learning.

There are several notable approaches that use IDE interaction data to build recommendation systems in software engineering [30]. Mylyn surfaces source code artifacts and IDE views that are relevant to a specific task, based on the developer's past interaction data accumulated for that specific task (i.e., its task context) [31]. Murphy-Hill et al. built an IDE command recommendation system based on several algorithms, including collaborative filtering and most frequently used commands [9]. While none of the recommendation algorithms used a per-developer task context, several of them relied on task context aggregated across the population of all developers, for example, by modeling how developers use specific new commands as frequent sequential patterns.

Our paper introduces two novel contributions beyond these previous efforts: 1) an exploration of the notion of using natural language inspired, probabilistic models for IDE interaction data; and 2) an approach for predicting the future development command context in software development. These contributions could benefit both empirical research in software engineering by aiding understanding of developer behavior in the field, without the observational bias of lab studies (i.e., Hawthorne effect), as well as software engineering recommendation systems by improving their models of task context.

2. The Eclipse Usage Data Collection (UDC) dataset is an older, well-known example.

5 Temporal LDA Topic Modeling and Prediction

We begin by first describing the specifics of Latent Dirichlet Allocation of IDE interaction data, followed by a definition of its extension for time-based topic modeling, Temporal LDA. Both models are initially created using historic IDE interaction data, collected either for a single developer or globally across all developers. Once constructed, the Temporal LDA model can be used for prediction and updated online, during the developer's interaction with the environment.

5.1 LDA Model of IDE Interaction Data

To build the initial LDA model, we decompose past developer interaction with an IDE into a set of interaction *sessions*, delimited by a period of inactivity of at least 5 minutes. We choose this interval with the goal of ensuring that, most of the time, a development task (e.g., structured navigation, debugging) does not span two sessions,[3] which we validate empirically by sampling and examining interaction traces.

To produce a reactive predictive model that can predict developer tasks in regular time intervals, rather than sporadically, on an activity break, we further divide the sessions into a succession of fixed-size *windows*, where each window is a sequence of m commands and events. Using shorter windows, rather than the sessions, also fosters better temporal locality in the model. We train both the initial LDA model with windows as documents as well as use windows for the Temporal LDA model and prediction. The choice of the parameter m, the number of interactions to include in a window, is important to the reactivity and accuracy of the model and is discussed in Section 7.

So, in applying LDA to interaction traces, a *window* of interactions corresponds to a *document*, an *interaction message* corresponds to a *word*, and a developer *intention* corresponds to a *topic*. In the following description, we use the interaction data specific terms (message, window, topic), when describing the LDA model.

An interaction message, denoted as m, is the basic unit of discrete data, while a vocabulary is the set of unique messages in the dataset, which we denote as $\mathcal{V}$. The number of messages in the vocabulary or the vocabulary size is $V = \mathcal{V}$. A interaction window is a sequence of N messages denoted as $\mathrm{m} = (m_1, m_2, \ldots, m_N)$ where m_n is the mth message in the sequence. A corpus is a set of M windows, denoted as $\mathcal{D} = \{\mathrm{m}_1, \mathrm{m}_2, \ldots, \mathrm{m}_M\} = \mathrm{m}_{1:M}$ where $M = |\mathcal{D}|$.

A topic, denoted as β, is a probability distribution over a fixed vocabulary. Specifically, if we assume K topics are associated with the corpus, the topics are $\mathcal{B} = \{\beta_1, \beta_2, \ldots, \beta_K\}$. The K topics are thus defined by their Probability Mass Functions (PMFs), e.g., the ith topic β_i is defined by its PMF as $P(m = m_j) = \beta_{i,j}$ where $m_j \in \mathcal{V}$ and $\beta_{i,j}$ is a probability, and thus $\beta_{i,j} \geq 0$ and $\sum_{j=1}^{V} \beta_{i,j} = 1$. Note that each interaction window exhibits all of these K topics; however, with different proportions, i.e., window w exhibits a topic indexed by k, where $1 \leq k \leq K$, with proportion $\theta_{w,k}$.

Given a set of observed windows from the IDE interaction data, we can train an LDA model to discover a set of

3. The converse that a session spans multiple tasks, is normal and expected.

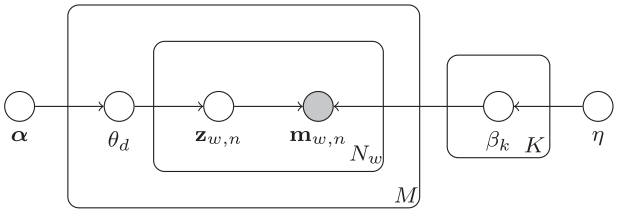

Fig. 2. Directed graphical model representation of LDA. Nodes represent random variables; edge denote possible dependences between random variables ($x \rightarrow y$ means x might be dependent on y); observed variables are shaded; unshaded nodes represent hidden (or latent) variables; a rectangle is a plate representing a replicated structure.

topics, which are *latent* (or hidden) variables. The LDA model used in this paper can be summarized by the graph model shown in Fig. 2. As input, LDA requires the set of messages in all of the windows in the corpus, represented by the shaded variable $\mathbf{m}_{w,n}$ and its surrouding plates denoting repeated structures for windows and messages. The variables (hyperparameters) shown on the right and left side of Fig. 2, η and α, can also be specified, affecting the distribution of the topics in the corpus (α) and the distribution of messages in each topic (η). Modifying these variables defines how sparse or dense the LDA model will become, with respect to its window to topic and topic to message relationships.

Note that we use a variant of LDA that features one symmetric and one asymmetric Dirichlet distribution for the priors parameterized by η and α. Prior work demonstrates that *symmetric* prior on topics and *asymmetric* prior on topic proportions produces better results than other variants [32]. In LDA, the priors are chosen to follow the Dirichlet distribution for computational efficiency and mathematical convenience since Dirichlet is a conjugate prior to the Multinomial distribution, which is used to model the document word (or message window) counts (i.e., the "bag of words" representation).

5.2 Temporal LDA

IDE interactions, as they are occuring during a developer's daily work, comprise of a stream (or sequence) of sessions (i.e., documents). Temporal LDA is an approach that extends LDA to model document streams, which has previously been proposed for predicting the topics distribution of a new tweet given a succession of historical tweets [33].

Temporal LDA learns a transform matrix that captures the transition of LDA topics between two consecutive windows of developer interactions. We conceptualize the Temporal LDA model in Fig. 3, where $\theta_t^{(u)} = \{\theta_{t,i}^{(u)} : 1 \leq i \leq K\}$ is the topic portions of developer u's interaction sequence at the window indexed by time t and $\mathbf{T} = (T_{ij} : 1 \leq i \leq K, 1 \leq j \leq K)$ is the transform matrix.

Note that T_{ij}, an element in the transform matrix, is *not* a probability value. Instead, T_{ij} is the weight of $\theta_{t,i}^{(u)}$'s contribution to $\theta_{t+1,j}^{(u)}$, which indicates how strongly topic j exhibited in an interaction sequence window at time $t+1$ is influenced by topic i exhibited in an interaction sequence window at time t; this influence can be positive or negative. Therefore, we should *not* consider Temporal LDA as a probabilistic model even though the topic proportions of a document at time t, i.e., $\theta_t^{(u)}$ depends on the topic proportions at

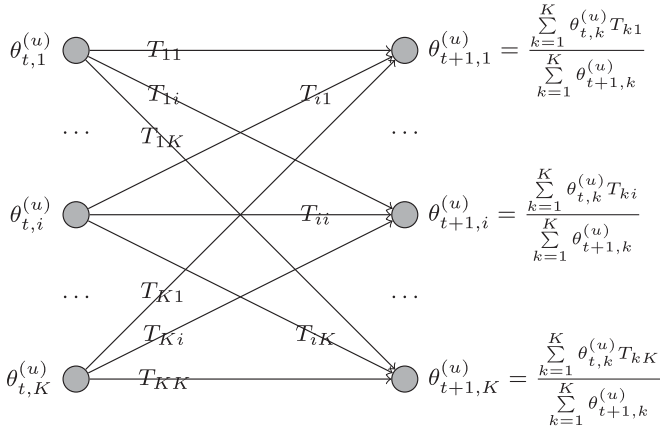

Fig. 3. The Temporal LDA model captures topic-to-topic transitions in consecutive windows of interactions.

time $t+1$, i.e., $\theta_{t+1}^{(u)}$, where both $\theta_t^{(u)}$ and $\theta_{t+1}^{(u)}$ are probability distributions of topics.

An essential task in building Temporal LDA is to determine the transform matrix $\mathbf{T}$ for a set of developers $\boldsymbol{\mathcal{U}}$. To this end, we resort to the determining the solution of the following linear equation system,

$$\boldsymbol{\Theta}_1 \mathbf{T} = \boldsymbol{\Theta}_2, \tag{1}$$

where $\boldsymbol{\Theta}_1$ and $\boldsymbol{\Theta}_2$ are L by K matrices, where $L \leq M-1$ and M is the number of interaction sequence windows in the corpus. This indicates that some interaction sequence windows *cannot* be used to obtain the transform matrix, as a result of the following constraint, row i in $\boldsymbol{\Theta}_1$ and that in $\boldsymbol{\Theta}_2$, $1 \leq i \leq L$ must satisfy the following condition,

$$\boldsymbol{\Theta}_{1,i} = (\theta_{t_i,j}^{u_1} : 1 \leq j \leq K, u_1 = u_2 = u \in \boldsymbol{\mathcal{U}}) \tag{2}$$

$$\boldsymbol{\Theta}_{2,i} = (\theta_{t_i+1,j}^{u_2} : 1 \leq j \leq K, u_2 = u_1 = u \in \boldsymbol{\mathcal{U}}). \tag{3}$$

This constraint states that both $\boldsymbol{\Theta}_{1,i}$ and $\boldsymbol{\Theta}_{2,i}$ must come from two interaction sequence windows of the same developer in two consecutive windows t_i and t_i+1 for any developer $u = u_1 = u_2 \in \boldsymbol{\mathcal{U}}$.

Once $\mathbf{T}$ is obtained, we predict the topic distribution of an interaction sequence at window $t+1$, i.e., θ_{t+1} given the topic distribution of an interaction sequence at window t, i.e., θ_t as follows:

$$\theta_{t+1} = \theta_t \mathbf{T}. \tag{4}$$

Updating the Temporal LDA model over time, in the course of a developer's interaction with the IDE, can be performed in several ways, depending on the stability of the learned model. For instance, if we observe that initial data to train the LDA model is not representative, or assume that the model can grow stale, we may choose to periodically retrain the entire model, paying a significant performance cost in the process. To make it practical, the rebuilding of the model can be performed offline, e.g., overnight. Metrics like perplexity and predictive likelihood can be used to measure the quality of an LDA model with respect to newly arriving data. On the other hand, if we observe that the initial LDA model is of sufficient quality, we can focus on only updating the transform matrix $\mathbf{T}$, tailoring it further to each developer's tendencies. In this case, the transform matrix can be

incrementally updated online using the original transform matrix in a way that reduces computational cost [33].

5.3 Summary

We summarize applying the Temporal LDA model to interaction data as follows. Using entire, unsplit log messages as words, and windows of m interactions in temporally contiguous sequences as documents, we train an LDA model using historical interaction data. The trained LDA model and the (same or subset of) historical data is used to create a transform matrix $\mathbf{T}$, the underlying data structure of the Temporal LDA model, used for making predictions of future interaction topics when given the current window of interactions. Trained in this way, the Temporal LDA model can be used as part of the IDE, to improve how recommendations are generated online, during a developer's use of the environment. The model can be updated at various frequencies and with different subsets of the interaction datasets produced, depending on assumptions of its quality, computational cost, and the desire to tailor it to an individual developer or, more broadly, to all developers.

6 Evaluation

To evaluate our approach of Temporal LDA for IDE interaction data analysis, towards predicting future developer task context using an interpretable model, we designed our evaluation with the intent of answering the following three research questions:

- How accurate is Temporal LDA in predicting future IDE interactions?
- How effective is Temporal LDA in recommending IDE commands?
- Is Temporal LDA capable of producing models of developer behavior that can be interpreted by those familiar with the vocabulary of interaction log messages?

The first research question concerns the ability to predict future behaviors (i.e., topics) exhibited by developers in the IDE, while the second focuses on predicting an individual commands of interest (e.g., `ExtractClass` refactoring), which could be recommended to a developer. The third research question examines, via a case study, whether this type of a model could be interpreted by developers, assuming that they are familiar with the interaction log messages.

6.1 Evaluation Data Sources and Procedure

For evaluation, we use developers' interaction traces for Microsoft Visual Studio and ABB Robot Studio. Visual Studio is a well known general purpose IDE, while Robot Studio is a popular IDE intended for robotics development that supports both simulation and physical robot programming and uses a programming language called RAPID. Both datasets are large and representative.

The Robot Studio data represents 25,724 developers over a maximum of 3 months of activity, or a total of 76,866 developer work hours. The Visual Studio data represents 196 developers at ABB, Inc. over a period of up to 12 months, or a total of 32,811 developer work hours.[4] Individual developers were identified by their machine identifiers. Otherwise, the data collection was completely anonymous and demographic information was unavailable for either dataset. Fig. 4 shows a density plot of developer interaction sessions for each of the two datasets, using a session break threshold of 5 minutes of inactivity, as described in Section 5. The density plot shows the presence of numerous longer sessions that allow for evaluation of the prediction accuracy based on session replay. As the Robot Studio dataset represents a narrow time slice and because interaction logs are not collected in real time and can be cached for long periods on user machines, there are numerous users that make a small contribution to our Robot Studio dataset. Though their contribution to the dataset is small, these users are not from different populations (e.g., trial users versus licensed users).

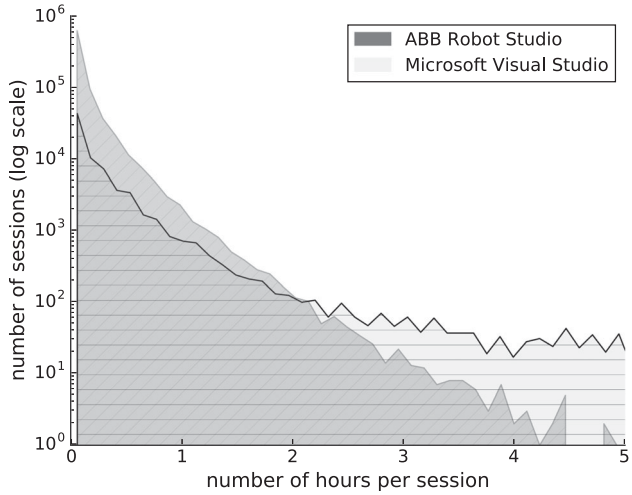

Fig. 4. Density plot of interaction session times (in hours) in the evaluation datasets for ABB Robot Studio and Microsoft Visual Studio.

We formed training and test data sets by dividing the data, such that the training data was used to train the Temporal LDA model and the test set was used for evaluation. We removed sessions that have only one window, due to the constraint in equations (2) and (3), which states that we can only use sessions that have at least 2 consecutive windows for training the model. For evaluation, we used sessions with at least 20 windows, forming a test set by reserving the last 20 windows from each session longer than this threshold.

Table 1 depicts a summary of the training and test data. For instance, after filtering out short sessions, the training and test datasets contain 145 developers' interactions with Microsoft Visual Studio. These datasets are made of 1,275 unique Visual Studio interactions, and when we set the window size at 25 messages, there are 19,937 sessions in the data set, resulting in a total of 210,688 windows for the training set, while the test data set has 3,433 sessions and 68,660 windows (as shown in the top row of Table 1).

6.2 Prediction Accuracy

As indicated in equation (4), Temporal LDA predicts the topic distribution exhibited in the next sequence of IDE interactions by using the topic distribution of the previous IDE interactions of the same developer. To evaluate the accuracy of this prediction, we identify windows from a previously recorded interaction sequence as the ground

4. Available at: http://abb-iss.github.io/DeveloperInteractionLogs

TABLE 1
Summary of Training and Test Data, After Filtering of Short Sessions

IDE	$\|\mathcal{U}\|$	V	Training Data N_d	Training Data N_s	Training Data M	Test Data N_{hs}	Test Data M_h
VS	145	1,275	25	19,937	210,688	3,433	68,660
	145		50	13,908	103,877	1,116	22,320
	145		100	8,722	48,819	241	4,820
	142		150	6,178	29,336	84	1,680
	142		200	4,583	19,730	32	640
RS	25,239	5,174	25	357,428	1,901,854	6,688	133,760
	25,239	5,174	50	341,785	794,168	1,628	32,560
	16,042	3,445	100	136,907	266,152	356	7,120
	11,389	2,731	150	70,939	75,891	167	3,340
	8,487	2,339	200	42,173	10,804	115	2,300

notations: *VS—Microsoft Visual Studio; RS—ABB Robot Studio; $|\mathcal{U}|$—number of developers (or users); V—number of unique interactions; N_d—number of interactions in each window (window size); N_s—number of sessions; M—number of windows; N_{hs}—number of sessions that have test windows; M_h—number of windows in the test data set.*

truth and compare our predicted topics to the ones exhibited by that developer's interactions in the ground truth.

There are multiple ways one could compare the predicted topics to the ones exhibited by the developer. For evaluating the potential use of topic modeling in recommendation systems, an important characteristic is that the single most influential predicted topic (i.e., the one with the highest probability) is accurate, because recommendations are likely to be made for that specific topic. Thus, we compute the accuracy of matching the most influential topic between the prediction and the ground truth. We label this metric *precision at 1* to indicate that the highest ranked topic (i.e., the one with the highest probability) is the primary focus of evaluation.

In addition, to measure the accuracy of the overall predicted distribution of topics, which gives a more comprehensive basis of evaluating the accuracy of the prediction, we also compute cosine similarity between the distribution of predicted and observed topics for the ground truth windows of interactions. We show both of these metrics for both IDEs and at several window sizes in Table 2. This table shows, for instance, that for $2/3$ of the time, (*precision at 1* = 0.67) for Visual Studio at a window size of 100 messages, Temporal LDA correctly identified the most influential topic

TABLE 2
Prediction Accuracy of Temporal LDA Model (Topics = 50). Both Metrics Range from 0 (Least Accurate) to 1 (Most Accurate)

IDE	N_d	Avg. Precision at 1	Avg. Cosine Similarity
VS	25	0.51	0.74
	50	0.58	0.79
	100	0.67	0.85
	150	0.74	0.89
	200	0.83	0.91
RS	25	0.89	0.94
	50	0.94	0.97
	100	0.99	0.99
	150	0.96	~ 1.00
	200	0.97	~ 1.00

TABLE 3
Percentage Improvement (in Number of Test Set Windows) of Temporal LDA Over Mirroring the Topics from the Last Window (Topics = 50)

IDE	N_d	Improvement in Precision at 1	Improvement in Cosine Similarity
VS	25	5.4%	68.0%
	50	4.3%	65.2%
	100	3.0%	62.0%
	150	2.8%	59.1%
	200	2.3%	54.6%
RS	25	2.8%	27.7%
	50	0.2%	27.3%
	100	0.1%	16.4%
	150	0.1%	10.8%
	200	1.2%	18.7%

out of 50 possible choices. As a baseline, consider the accuracy of always predicting the most probable LDA topic in the training set (i.e., ZeroR classifier) of 0.0993 or 9.93 percent. We also observe that the prediction accuracy improves with increasing window size, and Robot Studio is somewhat more predictable than Visual Studio at the same window size increments.

Next, we compare the prediction accuracy of our approach to simply mirroring the topics in the previous window, which effectively reflects the state of the art where it is assumed that the developer continues to perform exactly the same action as in the previous window. This baseline can check whether the **T** matrix has a positive impact on the prediction accuracy, as state of the art recommendation systems often use the short term past as a representation of the short term future. As before, we use precision at 1 and cosine similarity as the metrics. We computed the relative percent of windows where Temporal LDA provides an improvement over mirroring, i.e., $(wins_{TemporalLDA} - wins_{Mirroring})/M_h$, where a win is awarded for a window where a specific technique provides a better prediction (according to the metric) and M_h is the total number of windows in the evaluation set. The results are listed in Table 3. The first row of this table shows that, relative to topic mirroring, Temporal LDA improved the precision at 1 prediction by 5.4 percent and improved cosine similarity in 68 percent of test windows of length 25 in the Visual Studio dataset. Overall, these results show improvements in the number of predicted windows, relative to topic mirroring, for all window sizes and both IDEs, and for both of the evaluation metrics. We also observe that the utility of Temporal LDA, relative to mirroring, is smaller for Robot Studio, as interactions with this IDE seemed to be much more repetitive.

6.3 K-Tail Evaluation of Recommendation Accuracy

One of the key goals of our technique is to recommend previously unused IDE commands to individual developers, which are relevant to their current development task. The IDE command recommendation system proposed by Murphy-Hill et al. [9] introduced a set of algorithms, ranging from straightforward to sophisticated (e.g., combining sequential pattern mining with collaborative filtering) for

TABLE 4
% of Correct Message Recommendations in k-Tail Evaluation

IDE	N_d	% of correct recommendations
VS	25	38.2
	50	31.1
	100	23.1
	150	25.3
	200	21.1
RS	25	66.6
	50	75.3
	100	82.2
	150	74.6
	200	70.3

command recommendations. The effectiveness of the algorithms (i.e., the recommendation accuracy) was assessed using the *k-tail evaluation* strategy, initially proposed by Matejka et al. [2]. The intuition behind this evaluation strategy is to create a gold set using a set of commands that were newly discovered (or never before observed to be used) by a developer in the logged interaction data. The recommendation system's accuracy is evaluated by its ability to detect these newly discovered commands.

More specifically, in the interaction trace captured from a specific developer, after a start-up period of 2 windows, we evaluate the algorithm's ability to predict commands that were previously unseen in that user's trace. Only those newly discovered commands that occur more than once in the trace are used, filtering out spurious command uses. In order to translate from topics, which Temporal LDA predicts, to commands in the evaluation dataset we consider the top j highest probability commands in the strongest predicted topic. As a reasonable value for j we use the number of different commands that occurred in the window prior to the one predicted.

Using this evaluation setup, we present the percentage of newly discovered commands that were correctly predicted by Temporal LDA in our two datasets in Table 4. Across different window sizes Visual Studio has recommendation accuracy of between 38.2 percent and 21.1 percent, generally decreasing with larger window sizes. We expected smaller window sizes to perform better, since their narrower scope is more likely to focus in on the command of interest. On the other hand, Robot Studio accuracy was significantly higher, between 66.6 percent and 82.2 percent, with the highest accuracy peak at window size 100. Peaking at larger window sizes is likely due to the repetitiveness of Robot Studio's interaction data, which allows larger windows to capture a single longer developer behavior, but only up to a point.

The recommendation accuracy that we observed for both Robot Studio and Visual Studio was higher than the algorithms in Murphy-Hill's paper ($< 30\%$). However, these accuracies should not be compared directly without considering that: (1) we predict topics, which is easier than predicting commands, as there are fewer potential topics than commands; (2) we use fixed-time windows while Murphy-Hill et al. consider an evaluation approach that is time-insensitive; (3) we use different IDEs (Murphy-Hill uses Eclipse) which affect the prediction significantly judging by our two datasets. Another difference is that Murphy-Hill's proposed algorithms are directed towards recommendation of a single command, rather than accurate prediction of the overall behavior of the developer, and therefore could in fact be coupled with our approach instead of contrasted against it. For instance, command popularity, based on the usage of a command by similar developers, can easily be integrated with the predicted probability distribution of commands in a topic from our approach. Also, our approach could be used to predict when (not which) to introduce a command to a developer, enhancing Murphy-Hill's work along a different dimension.

6.4 Interpretability of Developer Behavior

The interpretability of the Temporal LDA model indicates how easy it is for researchers, and others, to understand: (1) the extracted topics and (2) the transition tendencies between pairs of topics. In order to illustrate the interpretability of the model, we must select reasonable parameter choices in building it, e.g., a window size of 50 interactions and 50 LDA topics for the Visual Studio dataset. In particular, the latter parameter (i.e., number of topics) must be carefully chosen to maximize interpretability, balancing between too few topics, where many behaviors are grouped into a single topic, and too many topics, where the behaviors become extremely fine grained. By examining the results for a few different numbers of topics in the Visual Studio dataset, we found that 50 topics provided a good balance in raising the level of abstraction, so that higher-level behaviors come to light, but without confounding numerous behaviors to a single topic.

Interpreting the resulting topics requires deep knowledge of the constituent messages, and the different possible contexts of their occurrence, much as knowledge of natural language words and their typical context is necessary in interpreting topics extracted from text. Often, interaction log messages are too brief to be clearly understood by developers that are not familiar with them or the product. We present the slice of topics related to debugging behavior in Visual Studio in Table 5. Debugging provides an adequate case study because it consists of numerous relevant messages (> 50) in our Visual Studio dataset, relatable to most developers. The extracted Temporal LDA model (in table 5) exhibits typical behaviors such as starting to debug, stepping through code, etc. We observe that for this dataset, each of these behaviors is logically assigned to its own topic. We also observe that events that always accompany some of these commands, e.g., Debug.Debug Break Mode. Debug.Debug Run Mode, which are confounding to some other approaches, are correctly included in several topics.

To further examine the interpretability of the model, we present a more in-depth analysis of the transformation matrix inferred by Temporal LDA in Fig. 5 focusing on the debugging topics introduced in Table 5. We find several interesting observations when focusing on the more extreme values in this matrix, highlighted in red. For instance, by observing the high value on the matrix diagonal at cell (1,1), we notice that when stepping over some code with the debugger, developers are most likely to keep repeating this action. Most of the debugging topics exhibit

TABLE 5
A Listing of the LDA Topics and Words that Concern Debugging, Extracted for the Visual Studio IDE Using Our Dataset (n = 50 Topics)

Topic 1	Stepping *over* with the debugger. (1.Debug.Debug Break Mode, 2.Debug.Debug Run Mode, 3.Debug.StepOver)	
Topic 10	Performing a (automatic) build and starting to debug. (1.Build.BuildDone, 2.Build.BuildBegin, 3.Debug.Enter Design Mode, 4.Debug.Start, 5.Debug.Debug Run Mode)	
Topic 17	Stepping *into* with the debugger. (1.Debug.StepInto, 2.Debug.Debug Break Mode, 3.Debug.Debug Run Mode)	
Topic 18	Starting to debug. (1.Debug.Debug Break Mode, 2.Debug.Start, 3.Debug.Debug Run Mode)	
Topic 28	Clicking on the call stack window. (1.View.Call Stack)	
Topic 33	Toggling a breakpoint (1.Debug.ToggleBreakpoint)	
Topic 42	Continuing to debug the project. (1.Debug.StartDebugTarget, 2.Debug.Start, 3.Debug.Debug Run Mode, 4.Debug.Debug Break Mode)	
Topic 49	Stopping the debugger. (1.Debug.Enter Design Mode, 2.Debug.StopDebugging)	

Glossary of Interactions: Debug.Debug Break Mode – the debugger has stopped at a breakpoint; Debug.Debug Run Mode – the debugger is running the code; Debug.StepOver, Debug.StepInto – stepping over (or into) a line of code with the debugger; Build.BuildDone, Build.BuildBegin – event signaling that building the project has started or completed; Debug.Enter Design Mode – event when the debugging session ends either because execution completed or because the debugger was stopped by the user; Debug.Start, Debug.StopDebugging – commands that start or end the debugging session; View.CallStack – clicking on the call stack window; Debug.ToggleBreakpoint – insert or remove a breakpoint from a line of code; Debug.StartDebugTarget – event that occurs when continuing to run a previously executing debugger;

We provide a description of each topic, an ordered listing of the constituent words (interactions), and a visualization of the distribution of word influences (only words with probability of 0.1 or greater are shown).

repetitive tendencies, observed by high values in the matrix diagonal, with the stepping topics as the most repetitive. This was also observed in a former study using the same Visual Studio dataset [11]. Apart from repeating, Topic 33, which represents toggling a breakpoint, strongly transitions into Topic 18 (starting to debug) and Topic 1 (stepping over with the debugger). This makes sense as developers likely set breakpoints before starting to debug, or set breakpoints in the middle of a debugging session, while stepping over the code, as we observed earlier in the log snippet in Fig. 1.

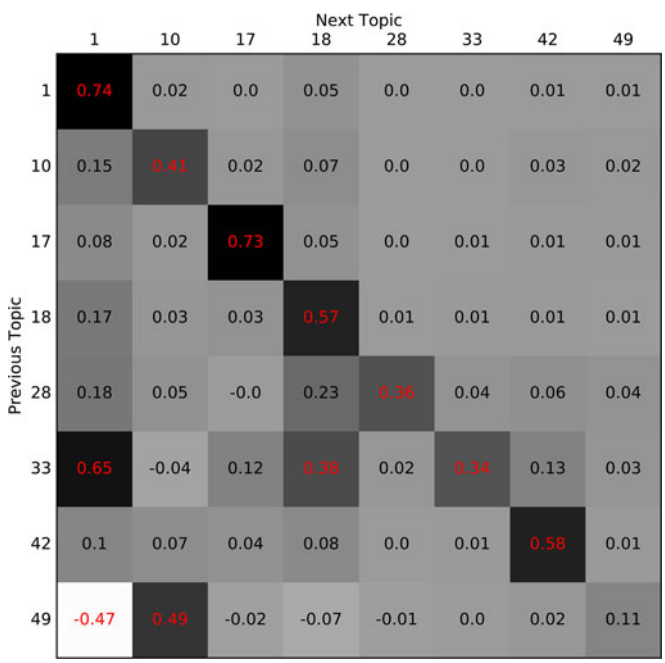

Fig. 5. Topic to topic transitions based on Temporal LDA model. Extreme values are highlighted in red.

We also observe high values from Topic 49 (stop debugging) to Topic 10, which rebuilds the code, presumably after a change, and restarts the debugger, and extremely low (negative) transition to topic 1 (stepping over). Note that, in this discussion, for simplicity, we omitted the transition of these topics to and from the remaining topics, which, sums to 1 in each row of the matrix, so it is proportional.

Robot Studio also contains debugging interactions, but its debugging capability of RAPID programs is not as rich as in Visual Studio. A Temporal LDA model formed with the same parameters (window size of 50 messages and 50 topics) provided two topics that were strongly debugging related. One of thee topic dominantly exhibited the RapidStepIn interaction message, while another topic exhibited the RapidStepOver message coupled with the ActiveWindowChanged message, which occurs when a new file is opened in the editor window. Both topics had strong repetitive behavior, observable in their transitions in the transformation matrix, with values of 0.97 and 0.98. A third topic, which encoded the behavior of starting and stopping a rapid program, showed significant transitions towards the two debugging topics, with strength of 0.17 towards the stepping in debugging topic, and 0.03 towards the stepping over topic, relative to its repetitive behavior strength of 0.72.

7 Parameter Analysis

In the previous sections, we observed that the model can be interpretable and has reasonable prediction accuracy, at specific choices of parameters for converting the interaction stream into documents that can be used in the LDA model and at specific choices of parameters of the LDA and Temporal LDA models. In this section, we describe the effect of different parameter values to the quality of the prediction,

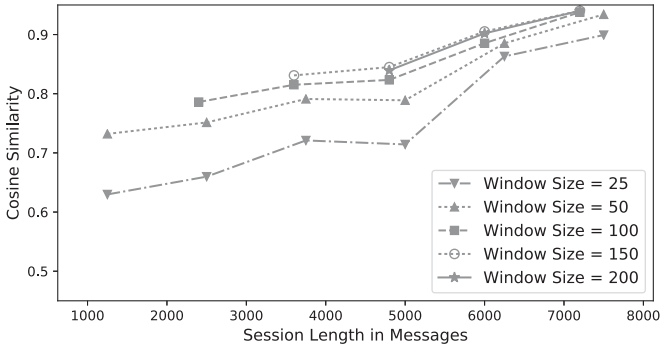

Fig. 6. Cosine similarity between predicted and observed topic distributions for different session lengths in messages for Visual Studio.

focusing separately on the parameters used to process the data and the parameters used to construct the model.

7.1 Effect of Interaction Data Processing

The key parameters in processing the interaction data are: (1) the number of the commands in each window, and, less so, (2) the session break criteria. The interaction data is initially broken into sessions, based on some time-based criterion, and then further subdivided into equal-length windows, which form documents to create the LDA model. Windows in the online interaction stream of a developer are used to create predictions and update the model.

The session break serves as a sanity threshold that will avoid placing interactions that were perfomed hours apart into the same window. Therefore, a value that corresponds to a time interval where we can reasonably assume that the developer has stopped working is appropriate. Previous work has used values in the range of 1 to 5 minutes [11], [21].

We examined how the prediction accuracy is affected by longer sessions, showing the result for Visual Studio in Fig. 6. Linear regression analysis of this data found that with high confidence ($> 99\%$) the prediction accuracy, measured as cosine similarity between the predicted and ground truth topics, improves with longer sessions, reflecting the property of the interaction data that more stability and predictability occurs in longer sequences.

The size of the window affects two aspects of Temporal LDA: the temporal locality of the LDA model and the responsiveness of the prediction. For temporal locality, we mean that smaller windows are likely to isolate messages occuring close temporally to each other, forming LDA topics that reflect this assumption. Larger windows would result in broader topics. As for the responsiveness of the prediction, it is clear that smaller windows would enable us to produce faster, but short-term predictions, while larger windows would produce less frequent but more far-reaching predictions. Naturally, the latter predictions would also be more error prone. Fig. 6, as well as Table 2, show the effect of window size on the prediction accuracy. We observe that the average accuracy improves with larger windows, likely because larger windows have a smoothing effect over abrupt harder-to-predict commands or events in the interaction trace.

7.2 Effect of Model Parameters

Key parameters for the Temporal LDA model are the number of LDA topics and values for the two hyperparameters constraining the generative process of the LDA model, $\boldsymbol{\alpha}$ and η.

While the number of topics can affect the granularity and interpretability of the model, it should not affect the quality of the prediction, if isolated from these other concerns. Wallach et al. [32] studied the quality of LDA when the number of topics is overestimated, finding that the model was relatively resilient, producing accurate inferences, and maintaining relatively correct relationships between the constituent words and topics.

The two parameters $\boldsymbol{\alpha}$ and η affect the tendency of developer interactions to be influenced by fewer or greater numbers of topics (increasing or decreasing the sparsity of the topics in developer interaction sequences) and the tendency of topics to be influenced by fewer or greater numbers of messages (increasing or decreasing the sparsity of the messages in topics). In this work, we chose an asymmetric Dirichlet $Dir(\boldsymbol{\alpha})$ to draw topic proportions for a developer interaction sequence, where $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_K)$ is a vector with K elements where K is the number of topics. If we have prior knowledge of which topics should be weighted more than others, we can boost those topics by setting a larger value of their corresponding elements in $\boldsymbol{\alpha}$, which will likely lead to a better LDA and Temporal LDA model, and in turn better predictions. However, if we lack solid prior knowledge on how developers' intentions are exhibited in their interaction messages with the IDEs, we can treat $\boldsymbol{\alpha}$ as a latent variable and learn it from the training data.

We performed an experiment where we trained a set of LDA models on the Visual Studio dataset, and examined the distribution of $\boldsymbol{\alpha}$ values learned by these models. We found that, for example, at window size 50 and 10 trainings, α_1 is determined as 0.025 ± 0.017 while α_4 0.214 ± 0.218. The difference in the range of these values provides evidence that prior knowledge of the topics could indeed improve the model by providing proper $\boldsymbol{\alpha}$. Also, in general, the asymmetric Dirichlet prior for $\boldsymbol{\alpha}$ is an appropriate choice for this model, rather than the symmetric one.

Another parameter of the model is the size and composition of the training data to construct both the LDA model and the Temporal LDA matrix $\mathbf{T}$. One could examine, for instance, if training the model using only the dataset of a single developer may yield more accurate, or more reliable, predictions. Another possibility is to use different settings in training LDA from training Temporal LDA, with the goal of producing a highly localized LDA model with a small window size, while maintaining larger window sizes for prediction in order to maximize its time period of utility. We consider these studies of training data as future work of this paper, along with the examination of larger and more heterogeneous datasets that include other popular IDEs.

8 Conclusion and Future Work

This paper presents a novel approach to dimensionality reduction and *future* task context prediction for IDE interaction datasets using a time-sensitive variant of Latent Dirichlet Allocation. The presented Temporal LDA algorithm extracts interpretable topics from noisy interaction datasets, and predicts the topic of the next set of interactions that will be performed by the developer with high accuracy. The

evaluation indicates that applying natural language modeling techniques to interaction trace datasets in software engineering is a promising approach to analyzing such data for recommendation systems.

The future work of this paper is multipronged, consisting of using larger and more heterogeneous datasets with Temporal LDA, exploring different training set configurations for the Temporal LDA model, and investigating the applicability of other natural language modeling techniques to IDE interaction data.

References

[1] B. Reeves and C. Nass, "The media equation: How people respond to computers, television, and new media like real people and places," Cambridge University Press, New York, NY, USA, 1996.

[2] J. Matejka, W. Li, T. Grossman, and G. Fitzmaurice, "Communitycommands: command recommendations for software applications," in *Proc. 22nd Annu. ACM Symp. User Interface Softw. Technol.*, 2009, pp. 193–202.

[3] T. Grossman, W. Li, J. Matejka, and G. Fitzmaurice, "Deploying communitycommands: A software command recommender system case study," in *Proc. 28th AAAI Conf. Artificial Intell.*, 2014, pp. 2922–2929.

[4] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Elipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, Jul.–Aug. 2006.

[5] K. Damevski, D. Shepherd, and L. Pollock, "A field study of how developers locate features in source code," *Empirical Softw. Eng.*, vol. 21, no. 2, pp. 724–747, 2015.

[6] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The BlueJ system and its pedagogy," *Comput. Sci. Edu.*, vol. 13, no. 4, pp. 249–268, 2003.

[7] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman, "Bespoke tools: Adapted to the concepts developers know," in *Proc. 10th Joint Meeting Foundations Softw. Eng. New Emerging Results Track*, 2015, pp. 878–881.

[8] A. Chis, O. Nierstrasz, and T. Gırba, "Towards moldable development tools," in *Proc. 6th Workshop Int. Evaluation Usability Programm. Languages Tools*, 2015, pp. 25–26.

[9] E. Murphy-Hill, R. Jiresal, and G. C. Murphy, "Improving software developers' fluency by recommending development environment commands," in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations Softw. Eng.*, 2012, pp. 42:1–42:11.

[10] Z. Soh, T. Drioul, P. A. Rappe, F. Khomh, Y. G. Gueheneuc, and N. Habra, "Noises in interaction traces data and their impact on previous research studies," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct 2015, pp. 1–10.

[11] K. Damevski, D. Shepherd, J. Schneider, and L. Pollock, "Mining sequences of developer interactions in visual studio for usage smells," *IEEE Trans. Softw. Eng.*, vol. 43, no. 4, pp. 359–371, Apr. 2017.

[12] G. Khodabandelou, C. Hug, R. Deneckère, and C. Salinesi, "Unsupervised discovery of intentional process models from event logs," in *Proc. 11th Working Conf. Mining Software Repositories*, 2014, pp. 282–291. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597101

[13] K. Damevski, H. Chen, D. Shepherd, and L. Pollock, "Interactive exploration of developer interaction traces using a hidden Markov model," in *Proc. 13th Int. Workshop Mining Softw. Repositories*, 2016, pp. 126–136.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847.

[15] E. Erosheva, S. Fienberg, and J. Lafferty, "Mixed-membership models of scientific publications," in *Proc. Nat. Academy Sci.*, 2004, vol. 101, no. 1, pp. 5220–5227.

[16] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Jan. 2003.

[17] J. K. Pritchard, M. Stephens, and P. Donnelly, "Inference of population structure using multilocus genotype data," *Genetics*, vol. 155, no. 2, pp. 945–959, 2000. [Online]. Available: http://www.genetics.org/content/155/2/945

[18] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proc. 3rd ACM SIGPLAN Workshop Evaluation Usability Programm. Languages Tools*, 2011, pp. 25–30.

[19] S. Amann, S. Proksch, and S. Nadi, "Feedbag: An interaction tracker for visual studio," in *Proc. 24th Int. Conf. Program Comprehension*, 2016, pp. 1–3.

[20] R. Minelli and M. Lanza, "Dflow–towards the understanding of the workflow of developers," in *Proc. 6th Seminar Series Advanced Techn. Tools Softw. Evolution*, 2013.

[21] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer: An investigation of how developers spend their time," in *Proc. 2015 IEEE 23rd Int. Conf. Program Comprehension*, 2015, pp. 25–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820289

[22] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 5–18, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6112738

[23] A. T. T. Ying and M. P. Robillard, "The influence of the task on programmer behaviour," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 31–40.

[24] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 803–813.

[25] M. Vakilian and R. E. Johnson, "Alternate refactoring paths reveal usability problems," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1106–1116.

[26] W. Van Der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Berlin, Germany: Springer Science & Business Media, 2011.

[27] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: Discovery and applications of usage patterns from web data," *ACM SIGKDD Explorations Newslett.*, vol. 1, no. 2, pp. 12–23, 2000.

[28] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann, "Abstracting execution logs to execution events for enterprise applications (short paper)," in *Proc. 8th Int. Conf. Quality Softw.*, 2008, pp. 181–186. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2008.50

[29] S. T. Iqbal and B. P. Bailey, "Understanding and developing models for detecting and differentiating breakpoints during interactive tasks," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2007, pp. 697–706. [Online]. Available: http://doi.acm.org/10.1145/1240624.1240732

[30] W. Maalej, T. Fritz, and R. Robbes, "Collecting and processing interaction data for recommendation systems," in *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer, 2014, pp. 173–197.

[31] M. Kersten and G. C. Murphy, "Mylar: A degree-of-interest model for IDEs," in *Proc. 4th Int. Conf. Aspect-Oriented Softw. Develop.*, 2005, pp. 159–168.

[32] H. M. Wallach, D. M. Mimno, and A. McCallum, "Rethinking LDA: Why priors matter," in *Porc. Int. Conf. Advances Neural Inf. Process. Syst.*2009, pp. 1973–1981. [Online]. Available: http://papers.nips.cc/paper/3854-rethinking-lda-why-priors-matter.pdf

[33] Y. Wang, E. Agichtein, and M. Benzi, "TM-LDA: Efficient online modeling of latent topic transitions in social media," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 123–131. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339552

Kostadin Damevski received the PhD degree in computer science from the University of Utah in Salt Lake City. He is an assistant professor in the Deparment of Computer Science, Virginia Commonwealth University. Prior to that he was a faculty member in the Department of Computer Science, Virginia State University and a postdoctoral research assistant in the Scientific Computing and Imaging institute at the University of Utah. His research focuses on information retrieval techniques and recommendation systems for software maintenance.

Hui Chen received the PhD degree in computer science from the University of Memphis in Memphis, Tennessee. He is an assistant professor in the Department of Computer and Information Science, Brooklyn College of the City University of New York. Before that, he was a computer science faculty member with Virginia State University. He engaged in geophysical research and worked as a software developer in industry. His research in computer science has been primarily on mobility management in wireless networks, caching for wireless systems, coverage problem of wireless sensor networks, accountable systems and networks, as well as understanding developers' interactions with Integrated Developing Environments. He served on various computer science and computer communications conference technical program committees and as reviewers for journals.

David C. Shepherd is a senior principal scientist with ABB Corporate Research where he leads a group focused on improving developer productivity and increasing software quality. His background, including becoming employee number nine at a successful software tools spinoff and working extensively on popular open source projects, has focused his research on bridging the gap between academic ideas and viable industrial tools. His main research interests to date have centered on software tools that improve developers search and navigation behavior.

Nicholas A. Kraft received the PhD degree in computer science from Clemson University in 2007. He is a software researcher at ABB Corporate Research in Raleigh, North Carolina. Previously, he was an associate professor in the Department of Computer Science, The University of Alabama. His research interests include software evolution, with an emphasis on techniques and tools to support developers in understanding evolving software and to support managers in understanding software evolution processes. His research has been funded by grants from the NSF, DARPA, and ED. He currently serves on the editorial board of the *IEEE Software* and on the steering committee of the IEEE International Conference on Software Maintenance and Evolution (ICSME). He is a senior member of the ACM and the IEEE.

Lori Pollock is alumni distinguished professor in Computer and Information Sciences, the University of Delaware and ACM distinguished scientist. Her research focuses on software artifact analyses for easing software maintenance, testing, and developing energy-efficient software, code optimization, and computer science education. She leads a team to integrate CS into K-12 through teacher professional development in the CS10K national efforts. She was awarded the ACM SIGSOFT Influential Educator award 2016 and University of Delaware's Excellence in Teaching Award, E.A. Trabant Award for Women's Equity in 2004. She serves on the Executive Board of the Computing Research of Women in Computing (CRA-W).