Highlights

Recognizing Developer Activity Based on Joint Modeling of Code and Command Interactions

Agnieszka Ciborowska, Kostadin Damevski

- novel unsupervised modeling of developer activity (e.g., debugging, testing) based on IDE interaction data.
- simulation-based evaluation indicates that using predicted developer activities has a positive impact on the quality of in-IDE command and code recommendations.
- results indicating that the combination of developer IDE commands and code accesses as input, rather than each of these data streams separately, yields models that are better at predicting human activity.

Recognizing Developer Activity Based on Joint Modeling of Code and Command Interactions

Agnieszka Ciborowska[a], Kostadin Damevski[a]

[a]*Department of Computer Science, Virginia Commonwealth University, Richmond, Virginia, USA*

Abstract

Recognizing the current activity of a software developer (e.g., debugging, reading, editing) can improve the effectiveness of recommendation systems that aim to reduce the cognitive load of information lookup during software development. Current recommendation systems based on developer activity detection focus primarily on a single dimension of developer behavior, e.g., both observing and recommending IDE commands or source code classes. In addition, the current state of the art techniques require that the number and type of activities exhibited by developers is pre-specified and that labeled interaction data is provided as input. In this paper, we propose the use of an approach that eschews these requirements, leveraging an unsupervised statistical model that uses both IDE commands and source code accesses to discern latent developer activities. Our approach outperforms baseline supervised and unsupervised learning techniques on simulation-based evaluation of source code and command recommendations for most of the configurations we examined. We also show that our technique benefits from observing both commands and code accesses when identifying developer activity.

Keywords: developer activities, interaction data, probabilistic hierarchical modeling, Hidden Markov Model, recommendation system

1. Introduction

As software continues to grow in size and complexity, software developers need to recall numerous and heterogeneous fragments of information to effectively perform their daily work. Such information is often scattered across internal (e.g., program elements, documentation, tests) and external resources (e.g., Q&A forums, tutorials, blog posts). Locating, understanding, and remembering relevant artifacts, like project classes and methods, prior issue reports, and API documentation, imposes a significant cognitive burden on developers [1, 2, 3]. Recommendation systems that aid software developers by suggesting relevant artifacts are a natural solution to this problem, but have yet to become prevalent or gain wide acceptance in the software developer community.

Recommendation systems in software engineering are distinct from their general-purpose counterparts in that they are highly task dependent [4, 5, 6]. The task of the developer, e.g., fixing a specific bug or implementing a specific feature, is a key component of recommendation context, relating to each other groups of program elements, commands, documentation, etc. that are necessary for task completion or can improve efficiency [7, 8]. At a finer granularity, the activity of a developer, e.g., debugging, running tests, navigating code, etc., provides further important context to better target recommendations [9, 10, 11].

The state of the art in automated developer activity detection has focused on a single dimension of the interaction data generated by a typical software developer. For example, detecting activities from source code accesses over time and recommending program elements [10] or detecting latent activities from a stream of Integrated Development Environment (IDE) commands and events and recommending the same [12, 13]. While using a single dimension of interaction data is effective, there are opportunities in leveraging multiple dimensions together. In this paper, we propose a technique for *joint modeling of source code accesses and IDE commands (and events) that uses both data dimensions to improve the quality of activity detection and the activity-aware recommendation of source code and IDE commands.*

Researchers have performed several studies to determine a de facto set of activities used in software maintenance [10, 11, 14, 15]. However, the mapping between pre-determined activities by researchers and those exhibited by a set of developers in the field is likely to be imperfect, as developer work styles have been observed to have strong personal variations [10, 11]. A key strength of the approach described in this paper is that it does not require a pre-determined, fixed set of activities, instead, allowing for the number of activities to be inferred based on the developers' interactions. Another advantage of the proposed approach is that it is unsupervised and therefore does not require labeled data, which is hard to obtain at scale and from a sufficiently diverse set of developers. A supervised model would have a tendency to overfit the behavior of the individuals or groups present in the labeled training data, exemplified by the fact that such models consistently produce best results when trained and tested solely on one individual's interaction data [10]. To summarize, the contributions of this paper are:

- technique that combines IDE commands/events and source code accesses to build statistical representations of developer activity;
- unsupervised modeling approach that does not require as input the number or pre-determined types of developer activities;
- simulation-based evaluation of the effectiveness of the approach when used as a part of a recommendation system.

The organization of this paper is as follows. Section 2 outlines the related work for the problem domain, while Section 3 and introduces a set of features representing IDE commands and source code accesses over time. In Section 4 we define the statistical model and its hyperparameters. Section 5 describes the research questions. Section 6 provides evaluation results for the source code and IDE command recommendation, while Section 7 shows evaluation results for activity type recognition. Section 8 discusses the threats to validity and, finally, in Section 9 we conclude the paper and list our plans for future work.

2. Related Work

The related work can be grouped into two categories: 1) methods for extracting higher level developers' intent from their interactions; and 2) recommendation systems that leverage developer interactions as context. We discuss the state of the art research in each category in turn.

Inferring latent developer tasks and activities from their interactions is an active area of research in software engineering, contributing to empirically understanding development in the field and leading towards improvements in design of IDEs and other tools. Maalej et al. investigated how tasks can be extracted from developer interactions with the source code in order to compute similarity between pairs of tasks [16]. Bao et al. collect interaction data at the operating system level in order to capture development data from all relevant applications (e.g., IDE, Web browser). They also propose a supervised learning approach, using Conditional Random Fields, for splitting the interactions into salient development activities [17]. Damevski et al. proposed an approach for interactively constructing Hidden Markov Models that capture development activities [18], which was later extended to Gadler et al. in order to be further automated [19]. Roehm and Maalej also used HMMs for inferring developer activites, but relied on pre-aggregated data encoding only seven different observed actions by the developers [20]. Recently, Chattopadhyay et al. performed field study to analyze how developers manage tasks and context in an industrial environment. The results revel that developers decompose their work into multiple subtasks of different properties, such as e.g., timespan of the subtask or the probability of switching to a new subtask [21]. In this paper, we extend the previous approaches with an unsupervised method that leverages modeling multivariate data without explicitly specifying the number of latent activities.

While research on recommendation systems that use developer interaction data has a long lineage, there are substantial differences among the various approaches in what is recommended (e.g., IDE commands, source code edits, source code clicks and key presses) and the corresponding input data used. Among the most influential early attempts is the work of Kersten and Murphy on Mylyn, which bases its recommendations on manual task feedback from the developer and a degree of interest model of source code accesses over time [22]. Another distinct set of recommendation systems have been proposed for suggesting underused commands in the IDE based on developer activity in order to improve their command fluency. An initial approach was described by Murphy-Hill et al. [12], with additional work by Zolkataf et al. [23], and Damevski et al. [13], while analogous work was performed by Li et al. in the context of recommending commands for Adobe Photoshop. [24]. Code completion within the IDE is also target for recommendation system. Robbes and Lanza proposed a recommendation system for in-IDE code completion that uses program edit histories recorded over time [25].

Only few recent recommendation systems use higher level abstractions of developer task and activity to move beyond straightforward and single purpose recommendation algorithms and models. Damevski et al. applied a temporal variant of Latent Dirichlet Allocation to the problem of activity detection based on IDE commands [13]. The approach was also targeted towards prediction and recommendation generation of IDE commands. Kevic et al. recently proposed an approach for detecting activities similar to the ones we aim to recognize in this paper, but used only source code activities and a supervised learning approach that requires training data [10]. They observed improved performance of models trained per developer to those that were trained across different developers. The approach of this paper enhances these prior attempts with a unsupervised technique that combines source code accesses with IDE commands and events.

3. Developer Activities Analysis

While working on a task, developers engage in different types of activities, such as, reading and editing code, debugging, testing, and interacting with version control systems [26, 27, 14, 15]. Many developers rely on Integrated Development Environments, and, therefore, each of these activities can be reified by a set of micro-interactions occurring between a developer and the IDE. A recording of these micro-interaction as a continuous stream forms an IDE interaction dataset, which can help understand development behavior in the field and improve the quality of support tools [28, 29, 20, 27].

Detecting activity types, as they occur, using IDE interaction data can be challenging due to several reasons, the paramount of which are data scale and granularity. Every interaction represents an atomic action performed by a developer, whereas an activity is a high-level reflection of developer's latent intent and it is built upon a mix of different micro-interactions. Moreover, there is no direct translation between activity types and low-level interactions as they may refer to more than one activity, e.g., using the IDE's structural navigation utilities may occur when developer is either editing or reading code. Another difficulty is the time– and order–dependency occurring between different activities, as, e.g., likelihood of switching from code reading to code editing is higher than to test execution [30]. As noted by Meyer et al. [11], developers tend to quickly switch between different activities, corresponding to abrupt changes in the types and frequencies of interactions in the stream, which can be challenging to interpret for an activity recognition system.

In this section we describe the characteristics of the IDE interaction data we use as a case study for our technique for automatically determining developer activity, the preprocessing steps we perform on this data, and the set of features that we extract to represent patterns of interest in the data.

3.1. Enriched Event Streams Dataset

In this study, we use the Enriched Event Streams dataset[31], consisting of over 11 million IDE commands and events (developer micro-interactions) collected in the Visual Studio IDE by the interaction tracker FeedBaG++ [32]. The dataset contains time-ordered logs of 15K development hours by 81 developers. On average, each developer contributed over 136K (median 54K) interactions corresponding to 185 (median 48) hours of work.

Interactions in the Enriched Event Streams dataset are represented via a set of 18 different events that represent the specific action that was performed by the developer or the current state of the IDE, e.g., executing a command is represented by a CommandEvent, editing code is stored in a EditEvent, using debugger is registered as DebuggerEvent and opening or setting focus on an editor is recorded as WindowEvent. Every event type has a set of attributes storing context information, such as, among others, number of changed characters (EditEvent), command identifier (CommandEvent) or type of navigation (NavigationEvent). In addition, code-related events, i.e. CompletionEvent and EditEvent, enclose a source code snapshot in the Simplified Syntax Tree format – a model able to efficiently represent code structure and type information.

3.2. Dataset Preprocessing

Developer interactions within the IDE are represented as a time-ordered stream. In order to prepare this streaming dataset for building an activity type detection model and its subsequent use in recommendation systems, we perform window-based segmentation of the stream. As the Enriched Event Stream dataset consists of different types of events, we opt to use the occurrence of specific events as delimiters of the interaction segments, instead of using fixed size segments based on event counts or time.

To select an event to use as a sampling point, first, we identify the different event types in the dataset that are associated with code elements, such as classes or methods. We are specifically interested in those events that most closely represent the beginning of an interaction between a developer and a code element. We focused on two possible candidates, NavigationEvent and WindowEvent, which we empirically examined by observing FeedBaG++ interaction stream while interacting with the IDE. We observed inconsistencies in triggering and collecting NavigationEvent, i.e. mouse clicks in the editor window did not always trigger a NavigationEvent. On the other hand, Win-

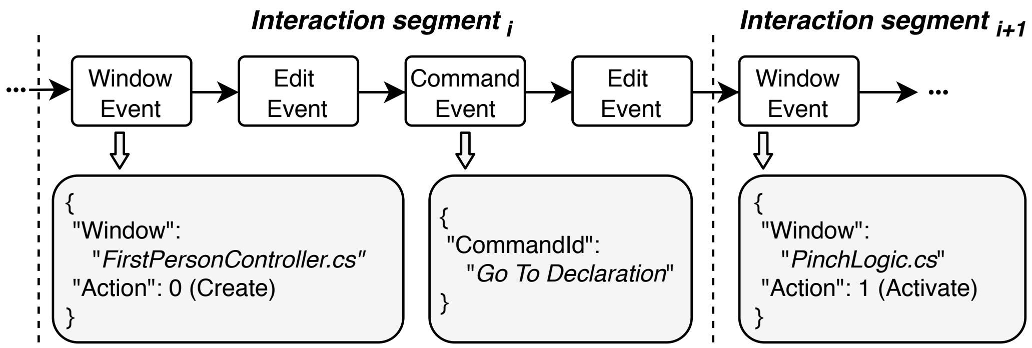

Interaction segment $_i$
Interaction segment $_{i+1}$
Window Event
Edit Event
Command Event
Edit Event
Window Event
{ "Window": "FirstPersonController.cs" "Action": 0 (Create) }
{ "CommandId": "Go To Declaration" }
{ "Window": "PinchLogic.cs" "Action": 1 (Activate) }

Figure 1: Stream of developer micro-interactions forming interaction segment.

dowEvent behaved as expected, being recorded every time a view, including a code editor, is clicked on or instantiated.

Presence of a WindowEvent in a stream indicates that the developer started interacting with a file by opening a new file in the code editor or activating an already opened tab. All events occurring after the occurrence of a single WindowEvent belong to a single *interaction segment* as illustrated in Fig. 1. Note that, in our dataset, there are no events indicating that a developer has left the IDE and is inactive. Therefore, to ensure that long time inactivity periods do not spuriously extend an interaction segment's duration, we apply a 20 minute inactivity threshold, a period after which we deem that is unlikely for a developer to be active without exhibiting any interactions [11].

3.3. Developer Activity Features

The next step is to translate each interaction segment into features that succinctly represent developer behavior with respect to the activity they are engaged in. We focus on three different groups of features that represent interactions with the code and commands in the IDE: 1) the relationship between consecutively visited classes, 2) the intensity of development activity, and 3) the characteristics of the IDE commands invoked by developer. The set of features we select in each of these groups is shown in Table 1.

We represent the relationship between visited classes using three measures: Coupling Between Objects (CBO), Common Elements Accessed (CEA) and Lexical Similarity. The first metric is commonly used to measure mutual class dependency[33]. It quantifies the relationship between two classes A and

Table 1: Features used to represent developer activity.

Group	**Feature**	**Formulation**
Code Metrics	Coupling Between Objects (CBO)	$\lvert E_{A\rightarrow B}\rvert + \lvert E_{B\rightarrow A}\rvert$
	Common Elements Accessed (CEA)	$\sum_{i=1}^{n} \lvert E_{A\rightarrow C_i} \cap E_{B\rightarrow C_i}\rvert$
	Lexical Similarity	$\frac{\lvert L_A \cap L_B\rvert}{\lvert L_A \cup L_B\rvert}$
Development Intensity	Edit Intensity	$\frac{\lvert \Delta_{chars}\rvert}{t_{segment}}$
	Activity Intensity	$\frac{\lvert \Delta_{commands}\rvert}{t_{segment}}$
IDE Commands	Command Types	$sPCA(\{commands\}_s)_{10}$
	Difference in Command Bigrams Distributions	$\sum_{i=1}^{n} \frac{(b_{i,s} - b_{i,s-1})^2}{(b_{i,s} + b_{i,s-1})}$

Notations: $E_{A\rightarrow B}$ – members of class B accessed by class A; L_A – tokens extracted from class A's fully qualified name; Δ_{chars} – changed characters during the interaction; $\Delta_{commands}$ – number of IDE commands during the interaction; $t_{segment}$ – time of the interaction segment in seconds; $b_{i,s}$ – tf-idf score of bigrams of type i in session s; $PCA(\{commands_s\})_{10}$ – PCA with 10 components on commands invoked in session s.

B by counting the members of class A accessed by class B and vice versa. Variations of the CBO metric were used in previous research to differentiate change task activities and to characterize navigation behavior [10, 34]. As a metric complementary to CBO, we use the CEA metric in order to capture common, external class references by counting methods and fields accessed by both classes that do not belong to either of them, but belong to some other class. Note that computing the CBO and CEA measures is imprecise in the Enriched Event Streams dataset when there are missing occurrences of EditEvent and CompletionEvent, which carry the source code for a specific class. If these types of events are not collected for a specific class, we are unable to factor that class' influence in CBO and CEA. The last class relationship metric is Lexical Similarity [10, 35], which is computed as the normalized count of shared tokens between fully qualified class names. The class names are split on punctuation and camel case to produce tokens.

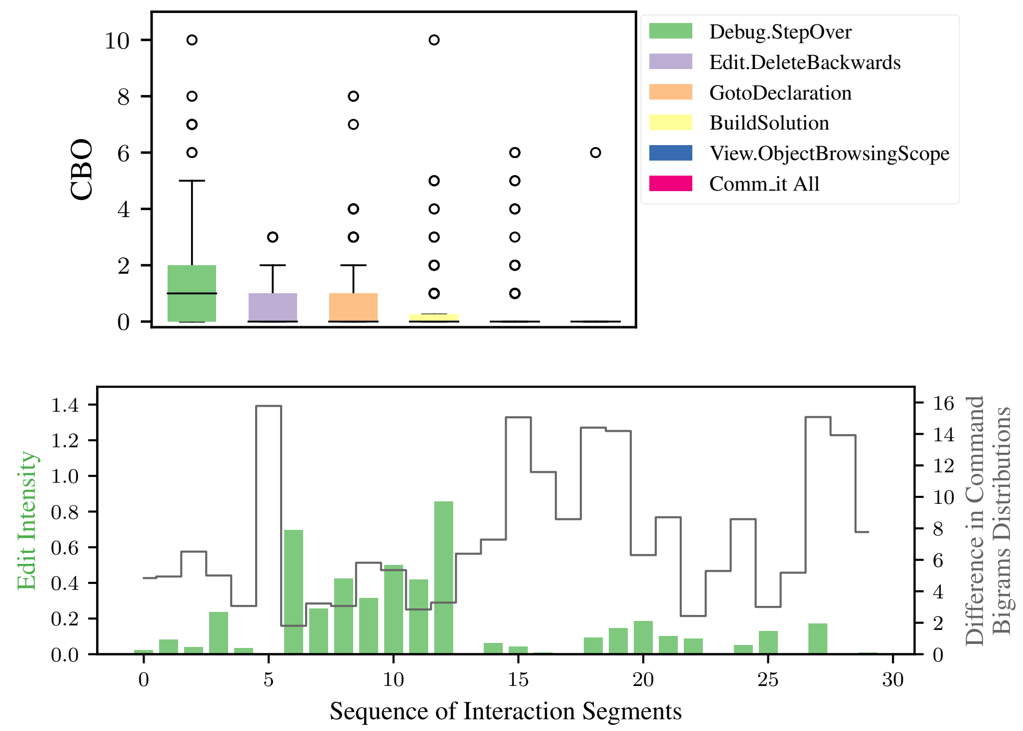


Figure 2: Features used to capture the properties of an interaction segment. The top plot shows values of CBO metric grouped by selected commands. The bottom plot illustrates edit intensity (bars) and the difference in command bigram distribution (lines) over the sequence of interaction segments collected from one developer.

The second set of measures accounts for developer's interaction intensity while working in an IDE. Low interaction intensity may correspond to tasks with a high cognitive load, e.g., program comprehension during a debug session, while high interaction intensity can reflect more mechanical tasks performed by developer, e.g., stepping through debugger [29]. Following similar quantities proposed in Kevic et al.[10], we introduce edit intensity, corresponding to the number of changed characters within the class, and activity intensity, related to the number of invoked IDE commands. Both features are normalized by the time of the interaction segment, measured in seconds.

To measure the characteristics of commands used during an interaction segment, we examine the command identifiers, labels corresponding to the observed action, which are included as a part of the context of a CommandEvent. As observed in other similar datasets [13], the recorded commands are noisy, with cases of several labels denoting certain singular events. The commands also follow a power law distribution, with a few very common commands and many others that occur sparingly. To account for aforementioned issues, we propose to encode the stream of commands invoked during an interaction segment as a vector of counts over most often occurring command identifiers in the dataset. First, we identify the top–100 most frequently occurring commands invoked by at least 10 developers. Next, for every interaction segment we count the number of times a command from the top–100 was invoked during the interaction segment to create a vector of command counts. Finally, to reduce the dimensions of the data, we apply sparse Principal Component Analysis (sPCA)[36] with 10 components.

To get an accurate measurement of change in a developer's command usage, we examine the distributions of commands during an interaction segment. We use command identifiers from consecutive CommandEvents within an interaction segment to create bigrams, e.g., `File.SaveAll` $\rightarrow$ `BuildSolution`. In order to account for common vs. uncommon bigrams we employ tf-idf encoding, whereby each interaction segment corresponds to a document, each bigram corresponds to a word, and the corpus is the set of all observed interaction segments. This procedure allows us to quantify interaction segments by their tf-idf weighted bigram distributions. Finally, we compute the chi-square distance between the current and the prior interaction's bigram distributions to quantify the change in a developer's command usage.

3.4. Characteristics of Developer Activity Features

In Figure 2, we briefly visualize and express some of the features selected to represent the interaction segments. The top part of the figure shows the distribution of CBO values for interaction segments differentiated by containing one of 6 common IDE commands in the dataset. Specifically, `View.ObjectBrowsingScope` corresponds to clicking on the Object Browser view in Visual Studio, `Edit.DeleteBackwards` corresponds to deleting a character with the backspace key, while `Comm_it All` denotes committing code to the repository via the IDE, `Debug.StepOver` denotes stepping over with the interactive debugger, `BuildSolution` invokes the VisualStudio compiler, and `GoToDeclaration` is a command that navigates from the use to the declaration of a class. Overall, we observe higher values of CBO for interaction segments containing commands related to editing, debugging and navigating. This observation confirms the expectation that higher values for a metric quantifying structural relation between code elements, i.e., CBO, correspond to occurrences of IDE commands that leverage the structure of the code. A related observation from this Figure is that commands corresponding to opening different classes or building the solution have relatively smaller values for CBO, indicating that developers only occasionally visit structurally connected classes while using the Object Browser view or building the project.

The relationship between Edit Intensity and the Difference in Command Bigrams Distributions, on a sequence of time-ordered interaction segments extracted from the Enriched Event Streams dataset for one of the developers, is illustrated in the bottom part of Figure 2. Focusing on the Difference in Command Bigrams Distributions, we notice several big spikes – clearly distinguishable moments in time with significant change in the command bigrams distribution, for instance, between the 4th and 5th interaction segment. High values for this feature indicate that the pattern of invoked commands changed significantly between the two interaction segments. After the spike, we observe a relatively flat region in the next few interaction segments, which implies that the invoked commands are similar in this period of time. Interestingly, between 6th and 12th interaction segments, immediately after the previously described spike in the command bigrams difference, we note a significant increase in the Edit Intensity. The observed change in values of both features over the course of these several time steps suggests that the developer switched her activity type at the 5th interaction segment and started a new activity requiring intensive editing of source code. We posit that the relationships between the features observed over the sequence of interaction segments is likely to correspond to specific activities, however, these relationships are not simple, where high values of one feature and low in another correspond directly to a specific activity. Rather, we believe that the overall trend is important as activities are often affected by prior developer interactions, as can be observed in the above example. To incorporate such knowledge from the history of performed activities, we propose to use a Hidden Markov Model that carries forth the influence from the prior states when deciding about the current developer activity.

4. Developer Activity Model

Two of the main dimensions of developer interaction data are source code accesses over time and IDE commands or events over time. The two dimensions are obviously related, as they both represent developers' actions as they are completing a task, thus, we require a model that is able to leverage both dimensions in order to capture the latent high-level activities of a single developer, during her daily work. We also prefer for the model to be unsupervised, so that it does not require ground truth developer activity labels, which is hard to obtain at scale and with enough population diversity. As a related important requirement, the model should not require a predefined and fixed set of developer activities.

Hidden Markov Models (HMMs) are a class of unsupervised modeling techniques that are well adept at capturing higher level behaviors from a sequence of events. HMMs have been shown to be useful for a variety of tasks, most notably in speech recognition and signal processing [37]. HMMs consists of two parallel random processes, one expressing the observable sequence of states, while the other revealing the hidden or latent sequence. For the purpose of developer activity detection, the observable states of an HMM correspond to IDE commands and code interaction sequences, while the hidden states correspond to developer activities. While HMMs accomplish many of the requirements we set forth, they fall short on a few important points. Specifically, HMMs typically rely on only one set of observable states and we have several features that we want to model jointly. HMMs also require prior knowledge of the set of hidden states, a definition we explicitly want to avoid a priori.

A recently proposed HMM variant is the Hierarchical Dirichlet Process Hidden Markov Model, or HDP-HMM [38], provides the desired functionality. It models multivariate data, so we can use multiple features, inferring a full covariance matrix of their relationships from the data. Such representation allows for additional flexibility in the observed data, since it does not require that each feature is independent from each other (like e.g., a naive Bayes classifier). Also, the model does not require a fixed number of hidden states, instead introducing an upper bound for the number of states max_K. The initial value of max_K is minimized through merging similar and removing redundant states during model training. HDP-HMM has already been found to be useful for a variety of purposes, including music synthesis [39] and modeling of gene expressions [40].

The "sticky" variant of HDP-HMM, which we use in this paper, further adds consideration for HMMs inclination to abrupt and too frequent transition between states, avoiding this by including an additional prior into the model [41]. The *sticky* parameter, denoted as hmm_κ, sets extra prior mass to the transition probabilities, thus adding inertia for the model to remain in the current state.

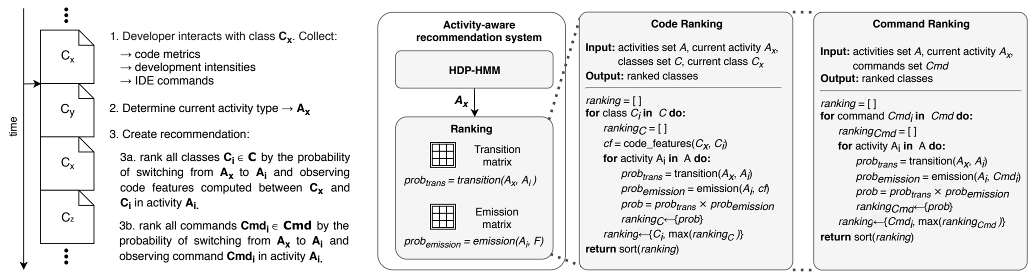
In our study, we use the HDP-HMM implementation proposed by Hughes et al.[42] that scales to big data application through parallelization of local inference steps across groups (or developers). The key parameters of the hidden part of the model, except of hmm_κ, are the symmetric Dirichlet priors: $start\_\alpha$, $trans\_\alpha$ and γ. The first two parameters are positive priors over the starting state probabilities and the state transition probabilities respectively, whereas γ is a concentration parameter of the top-level Dirichlet Process that corresponds to the latent base vector of probabilities over an infinite set of activities. As a model of the observable sequence we use the multivariate Gaussian distribution with a full covariance matrix. This distribution is parameterized through Wishart prior parameters, number of degrees of freedom (ν), scale factor of the expected covariance matrix (sF), and the Gaussian prior of the means (κ), which controls the inverse variance. We discuss choosing values for this relatively large set of parameters in Section 6.2.

5. Research Questions

Previous research has showed that with known activities the identification of relevant program elements for recommendation improves significantly [10]. Our work extends prior work via unsupervised models that provide both code and command recommendations from a single model. To understand the effectiveness of our model towards activity-aware recommendation systems, we investigate the following questions:

RQ1: Can the HDP-HMM activity modeling based on commands and code interactions aid in activity-based recommendation of (a) code elements, and (b) IDE commands?

To answer this RQ, we design an evaluation scenario that replays developer interactions as a simulation in order to evaluate whether a recommendation system could have assisted the developer, at a specific point in time, in finding relevant program elements and IDE commands with respect to detected developer activity. We measure the model performance according to a set of


time
C_x
C_y
C_x
C_z
Activity-aware recommendation system
HDP-HMM
A_x
Ranking
Transition matrix
$prob_{trans} = transition(A_x, A_i)$
Emission matrix
$prob_{emission} = emission(A_i, F)$

1. Developer interacts with class $\mathbf{C_x}$. Collect:
   → code metrics
   → development intensities
   → IDE commands
2. Determine current activity type → $\mathbf{A_x}$
3. Create recommendation:
   3a. rank all classes $\mathbf{C_i} \in \mathbf{C}$ by the probability of switching from $\mathbf{A_x}$ to $\mathbf{A_i}$ and observing code features computed between $\mathbf{C_x}$ and $\mathbf{C_i}$ in activity $\mathbf{A_i}$.
   3b. rank all commands $\mathbf{Cmd_i} \in \mathbf{Cmd}$ by the probability of switching from $\mathbf{A_x}$ to $\mathbf{A_i}$ and observing command $\mathbf{Cmd_i}$ in activity $\mathbf{A_i}$.

Code Ranking

```
Input: activities set A, current activity A_x,
       classes set C, current class C_x
Output: ranked classes
----------------------------------------------
ranking = [ ]
for class C_i in C do:
    ranking_C = [ ]
    cf = code_features(C_x, C_i)
    for activity A_i in A do:
        prob_trans = transition(A_x, A_i)
        prob_emission = emission(A_i, cf)
        prob = prob_trans × prob_emission
        ranking_C ← {prob}
    ranking ← {C_i, max(ranking_C)}
return sort(ranking)
```

Command Ranking

```
Input: activities set A, current activity A_x,
       commands set Cmd
Output: ranked classes
----------------------------------------------
ranking = [ ]
for command Cmd_i in Cmd do:
    ranking_Cmd = [ ]
    for activity A_i in A do:
        prob_trans = transition(A_x, A_i)
        prob_emission = emission(A_i, Cmd_i)
        prob = prob_trans × prob_emission
        ranking_Cmd ← {prob}
    ranking ← {Cmd_i, max(ranking_Cmd)}
return sort(ranking)
```

Figure 3: Building code and command recommendation lists using the HDP-HMM model. The model is trained on a project with C classes, and discovered A latent developer activities.

recommendation system metrics: novelty, correctness and item-space coverage. A simulation-based evaluation approach has previously been used for evaluating several recommendation systems [43, 13].

RQ2: What is the influence of the combination of code and command features on the performance of our model when recognizing predetermined types of developer activities?

The goal of RQ2 is to evaluate the performance of the model directly towards activity prediction, using a manually labeled pre-fixed set of developer activities. Specifically, we are interested in the performance of our model with 4 different feature sets: only code-related features, only commands-related features, only intensity-related features and all features combined. The dataset used in the study does not explicitly identify developer activities, thus, we annotated a portion of the dataset to conduct this experiment.

Unlike RQ1, the evaluation of RQ2 requires that we compare activities predicted by the model against ground truth labels. This is not ideal, as our unsupervised model does not necessarily produce activities in number or type that matches what a human annotator would expect. Therefore, this evaluation requires that we force the HDP-HMM model into producing a pre-determined number of activities, mapping those activities to the chosen ground truth set. Even though the model is not optimal, it can be used to answer the RQ by comparing the performance achieved by the same model with different feature combinations.

6. Activity-Based Recommendation (RQ1)

To evaluate the quality of code and command recommendations produced by the activity-aware model, we use a simulation scenario that mimics a developer interacting with a recommendation system, as illustrated in Fig. 3. Based on the features collected for the current interaction segment, the model predicts the developer's latent activity and subsequently uses the characteristics of that activity to build a list of relevant IDE commands or source code classes.

6.1. Dataset

Following the description in Section 3, we extracted time-ordered interaction segments from the Enriched Event Streams dataset [31]. Overall, the developers were active for 1149 interaction segments on average (median = 449; std. dev. = 1763), with each interaction segment occurring at an average interval of 244 seconds (median = 16; std. dev. = 3929). To conduct evaluation of code and IDE command recommendation, we divided developers' interaction segments into three datasets: training dataset (50,049 interaction segments, 15 developers), validation dataset (8,682 interaction segments, 29 developers) and testing dataset (8,643 interaction segments, 26 developers). The process for creating the training dataset was to use the data from the 15 developers with the highest count of interaction segments, while the remaining developers' data was split into validation and testing sets.

6.2. Baselines and Metrics

The primary challenge towards a direct comparison between our work and that of, e.g., Kevic et al. [10] is high differences among interaction datasets. Kevic et al.'s dataset includes activities annotated by developers, in the course of their daily work, however, it's a proprietary dataset unavailable for reuse. Therefore, the performance of the HDP-HMM model is compared against the following ML baselines:

K-Means. An algorithm commonly used for clustering unlabeled data [44]. At a very high level, K-Means operates in a similar manner as HDP-HMM in that it performs unsupervised clustering of the data into clusters. However, while HDP-HMM takes into account transitions between different states, i.e., activities, K-Means does not. Using K-Means as a baseline allows us to compare to a popular unsupervised model that completely ignores the time-ordered nature of interaction data.

Table 2: Hyperparameter values used during experiments.

Model	Parameter	Value	Model	Parameter	Value
HDP-HMM	*max_K*	30	K-Means	*n_clusters*	30
	hmm_κ	50		*max_iterations*	500
	γ	0.5		*tolerance*	0.001
	start_α	1.0		*n_init*	5
	trans_α	1.0	RF	*n_estimators*	10
	sF	1^{-8}		*max_features*	10
	κ	100		*max_depth*	15
	ν	18		*min_samples_split*	5

Feature Descriptions: *n_clusters* – number of clusters; *max_iterations* – max number of iterations; *tolerance* – convergence tolerance; *n_init* – K-Means runs with diff. seeds; *n_estimators* – number of trees; *max_features* – number of features to use for splitting; *max_depth* – max depth of tree; *min_samples_split* – samples required to split a node.

Random Forest (RF). An ensemble of classifiers that has been applied in various domains since it exhibits universally good performance [45]. However, as RF is a supervised algorithm, it requires annotated data, which prevents or significantly limits possibilities for its application in many real world scenarios. In our evaluation, we employ RF for the sole purpose of contrasting the performance differences between supervised and unsupervised methods. Given that we operate in an unsupervised environment, we manually annotated part of the dataset to train RF (see Section 7).

To ensure effectiveness of all algorithms, we perform hyperparameter tuning using grid search. Given the time-ordered character of our data, which is essential for HDP-HMM model, we cannot employ cross-validation, and thus we apply hold-out validation. In the case of unsupervised models, HDP-HMM and K-Means, we use the training dataset during training phase and validation dataset to test the performance of the models. Due to the fact that RF requires labeled data, during training we use the first half of the annotated dataset. The final selected, optimal hyperparameter values for each model are shown in Table 2.

To evaluate the quality of code and command recommendations, we employ a set of metrics from a software recommendation system domain that quantify the value of the recommendations to a user:
K-tail evaluation strategy for novelty detection was initially introduced by Matejka et al. [46]. It allows to assess model ability to predict previously unseen (i.e., novel) items, e.g., relevant but previously unseen classes that a developer should examine or edit. We compute the K-tail value by dividing

Table 3: Activity-based class recommendation.

Model	**Rec. List Length**	**Item-space Coverage**	**Correctness**	**K-tail**
HDP-HMM	1	**0.541**	0.263	0.239
K-Means	1	0.397	0.135	0.134
RF_6	1	0.401	**0.287**	0.247
RF_10	1	0.402	0.284	**0.249**
HDP-HMM	3	**0.756**	**0.509**	**0.505**
K-Means	3	0.722	0.363	0.388
RF_6	3	0.673	0.493	0.478
RF_10	3	0.673	0.495	0.475
HDP-HMM	5	0.831	**0.636**	**0.657**
K-Means	5	**0.851**	0.420	0.430
RF_6	5	0.766	0.608	0.614
RF_10	5	0.769	0.615	0.616

the number of times the desired element (class or command) appears in the recommendation list to the total number of recommendations.
Correctness [4] quantifies the accuracy of the model computed as the number of times that desired element is present in the recommendation list divided by the total number of recommendations. The metric is similar to K-tail with an exception of not having the requirement that an element to be recommended has not been observed before by the developer.
Item-space coverage [4] is a measure that refers to the percentage of elements that can ever be recommended by the model. We calculate this measure dividing the number of distinct recommended elements by the overall number of elements (classes or commands).

6.3. Experiment Setup

The simulation of the code and command recommendation system follows a three-step approach as illustrated in the left side of Figure 3. First, we compute our set of features for the current interaction segment. Next, we use these feature values with each of the three activity detection models (HDP-HMM, RF, and K-means) to determine the current developer activity, A_x. Finally, specific for each model, we use the characteristics of the predicted activity to rank classes and IDE commands.

For step 3, since each of the studied models captures the properties of developer activities and its features using a different approach, we tailor a specific recommendation strategy that best utilizes each model's capabilities for generating recommendations. In the case of HDP-HMM, to produce a ranking of the classes, we compute the probability of observing the code features given the probability that developer transitioned to a new activity, as illustrated on the right side of Figure 3. Specifically, for code recommendation, given the current class C_x and activity type A_x, we use the following formula to compute probability for each class C_i in the project:

$$prob_{C_i} = transition(A_x, A_i) \times emission(A_i, CF(C_x, C_i))$$

where *transition* corresponds to the probability of switching from activity A_x to A_i as captured by transition matrix, *emission* represent the probability of activity A_i emitting certain features, and CF represents code features computed between class C_x and C_i. To recommend commands, for each window, we encode the number of occurrences of the top 100 commands in a vector, and apply PCA to obtain the same data representation as during training.

For the K-Means baseline, the recommendation generation strategy is based on computing distances between the code features or the command vector and a cluster centroid of the predicted activity. In the case of RF, we create a list of recommended classes or commands using features' importance weights computed for each activity type. To acquire feature weights for an activity type, we average the feature importance weights from the top 3 decision trees with the highest probability of correct prediction for a given activity type. We multiply these features' importance weights by the code and command features for the current interaction window in order to obtain a importance factor for each class and command respectively. Note that K-Means and RF do not capture transitions between activity types, thus we assume the predicted activity and the next activity are of the same type.

6.4. Results

Experimental results for the code recommendation with respect to three recommendation list lengths are presented in Table 3. We observe that the HDP-HMM model outperforms the baselines achieving the best results for Correctness and K-tail metrics for the recommendation list size of 3 and 5. In the case of recommendation list of size 1, the HDP-HMM model performs slightly worse than RF in term of Correctness and K-tail with a difference in performance of 0.024 and 0.01 respectively. On the same experiment, RF achieves Item-space coverage of about 40%, while HDP-HMM covers 54.1% of

Table 4: Activity-based IDE command recommendation.

Model	**Rec. List Length**	**Item-space Coverage**	**Correctness**
HDP-HMM	1	0.066	**0.192**
K-Means		**0.161**	0.032
RF_6		0.062	0.039
RF_10		0.039	0.040
HDP-HMM	3	0.118	**0.313**
K-Means		**0.214**	0.146
RF_6		0.093	0.147
RF_10		0.050	0.149
HDP-HMM	5	0.166	**0.320**
K-Means		**0.266**	0.171
RF_6		0.116	0.201
RF_10		0.074	0.167

classes in the project. As expected, the Item-space coverage metric for three activity-aware models increases with larger recommendation lists, with the HDP-HMM achieving highest values of 0.541 and 0.756 for recommendation lists of size 1 and 3. For recommendation list of size 5, K-Means gets slightly better Item-space coverage of 0.851 relative to the 0.831 noted for the HDP-HMM model.

We measure the model's ability to predict novel (previously unseen) classes in the source code using the K-tail metric. For the classes in our dataset, the HDP-HMM model has recommendation accuracy ranging between 0.239 and 0.657, increasing with the recommendation list length, outperforming the second best model (RF) for list sizes of 3 and 5 by 0.027 and 0.041. In the case of the one element recommendation, the RF achieves the best performance with K-tail value of 0.249, superseding HDP-HMM by 0.01.

The results for IDE command recommendation are presented in Table 4. To limit the dimensionality in representing commands, we trained the HDP-HMM model (and the baselines) with the top 100 most often occurring IDE commands, which affects the need for novelty evaluation with the K-tail metric. In other words, the commands represented by our model are rarely novel. Therefore, we omit this metric for command recommendation. The results in Table 4 show that the HDP-HMM model outperforms RF and K-Means across all recommendation list sizes on Correctness, with the value of this metric reaching 0.320 for a recommendation list size of 5 commands. In contrasting the Correctness between the three models, we note the highest difference for the list of size 1 with HDP-HMM achieving Correctness of 0.192, compared to less than 0.05 for RF and K-Means. The highest value of Item-Space Coverage is produced by K-Means, while HDP-HMM achieves the second best result in all three configurations.

We observe that the recommendations created by the HDP-HMM model are relevant and are better than our chosen baselines in most of our trials. Therefore, we conclude that the HDP-HMM model based on our features has the potential to aid developers by recommending both source code and IDE commands.

7. Detecting Activity Type (RQ2)

To study the performance of the model when using different combinations of features, we perform gold set evaluation, where we compare the model's predictions against a set of manually labeled activities.

7.1. Dataset

We randomly select data from 5 developers in the dataset to annotate with ground truth labels. For this purpose, we consulted literature [7, 14, 26, 47], choosing two different sets of activities, of 6 (Group A) and 10 (Group B) items respectively, as shown in Table 5. Both groupings are aimed to be individually complete, covering commonly known developer activities at two different levels of granularity. Using the two sets of labels, both of the authors separately annotated data of selected developers, consisting of the total number of 1613 interaction segments. To aid annotation, we used a listing of the code and command events in each interaction segment, including the name of the visited class, the 5 most common command bigrams and the 5 most common individual command identifiers. The agreement between authors' initial annotation for Group A and Group B reached reasonable Cohen's Kappa values of 0.81 and 0.61 respectively. Differences between the annotations were resolved via in-person discussion to produce a final annotated dataset. Note that the annotated dataset is unbalanced, e.g., 671 edit activities vs. 21 repo activities in Group A, and 275 read activities vs. 45 run activities in Group B.

Table 5: Activities selected for data annotation.

Label	Description	Group A(6)	Group B(10)
edit	editing code[26]	✓	✓
read	reading code[26]	✓	✓
break	developer stopped working[7]	✓	✓
repo	using version control system[26]	✓	✓
debug	using interactive debugger[26]	✓	
build	using build tools[14]	✓	
d&b	using build tools and interactive debugger[14]		✓
nav	navigating using IDE utilities[26]		✓
run	running application[26]		✓
nuget	dependency management[47]		✓
hi_int	multiple activities during a interaction segment[26]		✓
other	other activities[26]		✓

7.2. Baselines and Metrics

For baselines, we use Random Forest as a representative supervised learning approach and K-Means as a representative unsupervised learning approach, similar to RQ1. We use similar hyperparameters for all techniques as in RQ1, with the exception of the number of states (K in HDP-HMM and $n\_clusters$ in K-means), which we set to match the number of activities in the annotated set. In HDP-HMM, with the max_K parameter we can only specify an upper value threshold. However, as the number of states we want to achieve (6 and 10) is low, the generated number of states based on our dataset always reaches this threshold.

We employ the following metrics frequently used for classification results to assess models effectiveness:
F1 score[48, 49] measures the accuracy of classification by calculating a harmonic mean of precision and recall. The final score for all predicted activities is computed using two types of F1 averaging schemes: macro and weighted. In the first case, F1 score is calculated as a unweighted average of F1 scores for all activity labels, whereas the weighted scheme finds the average weighted by the activity label supports.
Matthews Correlation Coefficient (MCC)[50] is a balanced measure representing relationship between the predicted activities and the ground truth activities. MCC for the multi-class problem is computed as an average

Table 6: Activity prediction with different sets of features.

Features	**F1 score (macro)**			**F1 score (weighted)**			**MCC**		
	HDP-HMM	*K-Means*	*RF*	*HDP-HMM*	*K-Means*	*RF*	*HDP-HMM*	*K-Means*	*RF*
6 activity types									
Code	0.163	0.196	0.167	0.318	0.339	0.348	0.068	0.106	0.031
Commands	0.398	**0.255**	0.566	0.647	**0.459**	0.808	**0.553**	**0.223**	0.748
Intensities	0.249	0.096	0.356	0.500	0.223	0.631	0.259	0.036	0.483
All	**0.467**	0.243	**0.630**	**0.674**	0.406	**0.856**	0.518	0.175	**0.812**
10 activity types									
Code	0.152	0.182	0.115	0.268	0.325	0.269	0.125	0.157	0.043
Commands	0.235	0.157	0.302	0.415	0.297	0.570	**0.337**	0.126	0.490
Intensities	0.222	0.119	0.260	0.372	0.262	0.480	0.279	0.092	0.373
All	**0.266**	**0.207**	**0.332**	**0.422**	**0.348**	**0.607**	0.306	**0.191**	**0.525**

of MCCs of all predicted classes.

7.3. Experiment Setup

We evaluate the performance of activity detection with respect to two sets of predefined labels shown in Table 5. During the training phase, HDP-HMM and K-Means are trained using the unlabeled training dataset, whereas in the case of RF, we apply the first half of the annotated dataset. To evaluate the performance of the models, we use the second half of annotated dataset ensuring that the same test set is used across the three different models. Since for this experiment, the model's activity predictions need to be compared against a set of ground truth labels, for HDP-HMM and K-Means we are forced to set the number of latent states and the number of clusters to 6 and 10 (the number of labels), which does not yield the optimal model performance, as the models are not able to effectively infer or cluster all patterns occurring in the data. To compare predictions of both approaches against the ground truth labels, we use the Hungarian algorithm [51], which maps clusters or latent states predicted by the models to the activities in the annotated dataset.

7.4. Results

The results for activity type detection with respect to the two predetermined sets of 6 and 10 developer activities and different sets of features are presented in Table 6. Overall, the HDP-HMM model exhibits the second best predictive performance across the metrics for both activity sets, with

F1 macro scores of 0.467 for 6 activity types and 0.266 for 10 activity types. The HDP-HMM model results are superseded by the Random Forest (RF) results, with F1 macro scores of 0.630 and 0.332 for 6 and 10 activity types respectively. Given the fact that RF has an advantage of using annotated data during training, it is not surprising to observe better prediction performance when compared the unsupervised HDP-HMM model for the activity prediction task. Moreover, as the number of latent states for the HDP-HMM model is fixed due to the nature of the evaluation scenario, its performance is likely to be suboptimal. The second unsupervised model, K-Means, achieves the lowest activity prediction accuracy indicating that K-Means is unable to correctly distinguish between pre-determined activities.

To pinpoint the influence of modeling with combined command and code features, we compare the models' performance for activity prediction using four sets of features: command-related features, code-related features, intensity-related features, and the set of all the features. In the case of 10 activities, we notice an increase in predictive performance across all the metrics and models when modeling with the set of all features, with an exception of MCC value for the HDP-HMM model. Specifically, for the HDP-HMM model, we note an increase in F1 macro score of 0.013, 0.114 and 0.044 when compared to using only commands, code or intensity features respectively. In the case of 6 activities, we observe higher predictive performance for RF across all the metrics, whereas HDP-HMM exhibits improvements in both F1 metrics. Interestingly, K-Means trained with 6 clusters reaches the highest performance when only command-related features are used during training. This result is most likely the manifestation of some correlation between command features and activities, which is easier to identify in the absence of the code- and intensity-related features. However, the overall performance of K-Means is still significantly lower when compared to the performance of the remaining models.

In most cases and for most models, the combined feature set outperforms individual groups of features. More specifically, the HDP-HMM model benefits from both command and code features when performing activity prediction.

7.5. Model Explainability and Validity

To examine the explainability and validity of our HDP-HMM model of developer activities used in RQ2, we empirically examined the developer rationale encoded by the inferred model. Transition probabilities between 6 latent developers' activities (as in Table 5) inferred by the HDP-HMM model are presented in Figure 4. We did not visualize transitions with probabilities less than 0.10 to increase readability. The strength of each transition is depicted with three types of lines: dotted lines represent weak transition probability, regular lines correspond to medium probability and thick lines denote high probability. Overall, we notice that many of activities have the strongest inclination towards self-transitions, indicative of a known preference for the developers to remain performing the same activity from one interaction window to the next. Exceptions to this rule are the Break and Build activities, which can be rationally expected to lack continuity and promote transition to Read. Apart from a strong self-transition, the Read activity transitions into Edit and Debug, which, in turn, have the strongest transitions back to Read. Other developer studies have also shown that these activities occur frequently with developers rapidly alternating among these activities during a typical workday [26, 14]. We also observe that the Debug activity is well connected to several other activities, especially from Read and Edit. This may demonstrate a frequently observed pattern of developers editing code or reading and comprehending code while engaged in a session with the interactive debugger [52]. In addition, the strong transition from Build to Debug shows that the model is consistent with the ingrained behavior of modern IDEs to rebuild the software each time before starting a session with the interactive debugger [13].

8. Threats to Validity

We identify a number of threats to the evaluation's validity, both internal and external. Missing source codes is an internal threat that arises directly from the fact that code snapshots in the Enriched Event Stream dataset were only recorded in a context of a CompletionEvent or EditEvent, and those may not occur during every interaction segment. As a result, we are unable to compute code features for a part of the available interaction segments. To mitigate this threat, we extract all code snapshots for a developer and for every class with missing source code, we use the class's snapshot that was collected in a previous interaction. We deem that if there is no EditEvent or

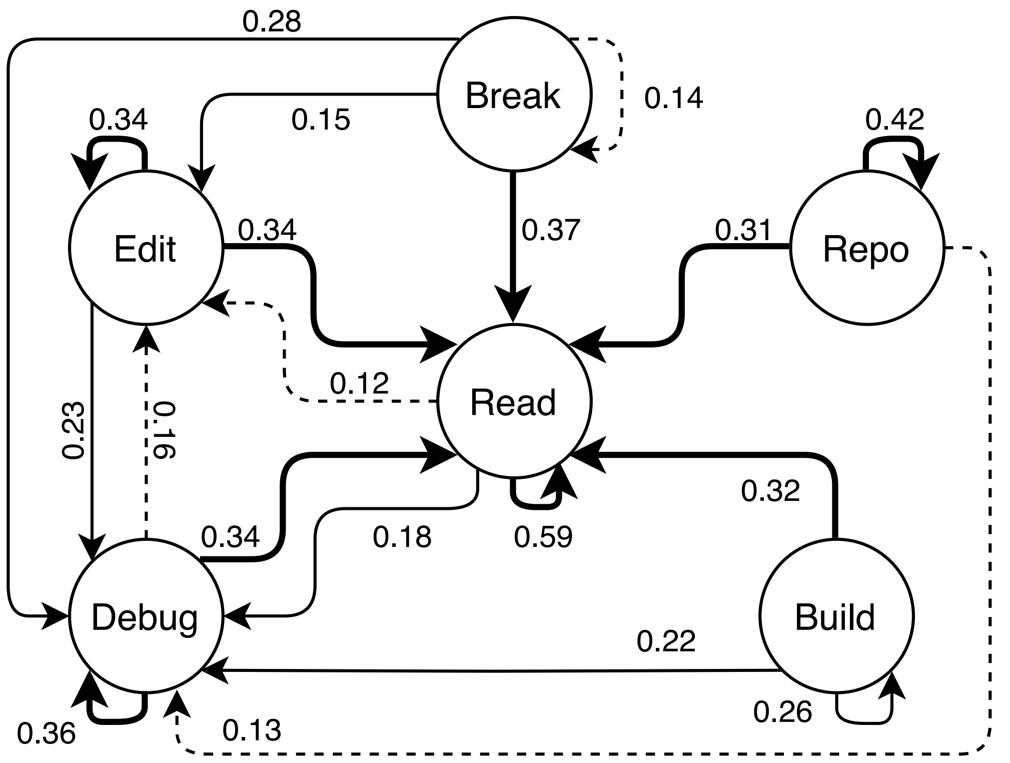


Figure 4: Transitions graph of the HDP-HMM model trained with 6 activities.

CompletionEvent recorded during a interaction segment, the source code is not changed, thus we are allowed to use the last known snapshot. Although this approach reduces the threat, it does not eliminate it completely.

Selecting WindowEvent as a sampling point poses another internal threat, as it influences data granularity. During a single interaction segment developer may perform different types of activities, as noted in Latoza et al. [15], while in our approach each interaction segment is identified as a singular activity that is dominant based on the values of the features. Additionally, another threat is caused by the data imbalance as some activities are more frequent than the other (e.g., editing happens much more often than building the application [14]). This threat is mitigated only for HDP-HMM, as it uses a multivariate probabilistic representation for each activity, which expresses activity transitions probabilistically, effectively representing a period of time between two data points in the interaction stream as a mixture of activities.

Annotating interaction data with predetermined set of activity types makes our approach susceptible to a construct threat as the selected activities may not fully reflect the characteristics of the data. To mitigate the threat, we select the two sets of activity types varying in granularity and scope, that were identified by researchers during both lab and field studies [26, 15, 17, 7, 14].

9. Conclusions and future work

This paper presents an application of an unsupervised statistical model to identify latent developer activities based on features representing both IDE commands and source code accesses over time. Using interaction segments extracted from the Enriched Event Streams dataset and represented by a set of command- and code-related features, we examined the model's effectiveness for code and command recommendation relative to popular baselines. We also investigated the effect of leveraging different types of features on the model's performance on a separate, simpler task - prediction of a predetermined set of activity types.

The results of the study indicate that: (1) statistical, unsupervised models are able to model software developer activities effectively for the purpose of generating recommendations without the aid of labeled data, (2) joint modeling using command and code features increases model accuracy, and (3) further support for prior findings that activity-based code recommendation can enhance recommendation quality. Our future work is in applying the model to a wider set of datasets and in examining active (feedback-driven) learning of developer interaction models.

References

[1] A. J. Ko, R. DeLine, G. Venolia, Information needs in collocated software development teams, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, 2007, pp. 344–353. doi:10.1109/ICSE.2007.45.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on Software Engineering 32 (12) (2006) 971–987. doi:10.1109/TSE.2006.116.

[3] M. P. Robillard, Y. B. Chhetri, Recommending reference API documentation, Empirical Software Engineering 20 (6) (2015) 1558–1586.

[4] M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann, Recommendation Systems in Software Engineering, Springer Science & Business, 2014.

[5] L. Zou, M. W. Godfrey, An industrial case study of Coman's automated task detection algorithm: What worked, what didn't, and why, in: Proceedings of the 28th International Conference on Software Maintenance, IEEE, 2012, pp. 6–14.

[6] I. D. Coman, A. Sillitti, Automated identification of tasks in development sessions, in: Proceedings of the 16th International Conference on Program Comprehension, IEEE, 2008, pp. 212–217.

[7] M. Gasparic, G. C. Murphy, F. Ricci, A context model for IDE-based recommendation systems, Journal of Systems and Software 128 (C) (2017) 200–219. doi:10.1016/j.jss.2016.09.012.

[8] A. T. T. Ying, M. P. Robillard, The influence of the task on programmer behaviour, in: Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, ICPC '11, IEEE Computer Society, 2011, pp. 31–40. doi:10.1109/ICPC.2011.35.

[9] R. Minelli, A. Mocci, M. Lanza, I know what you did last summer - an investigation of how developers spend their time, in: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC '15, IEEE Computer Society, USA, 2015, p. 25–35. doi:10.1109/ICPC.2015.12.
URL `https://doi.org/10.1109/ICPC.2015.12`

[10] K. Kevic, T. Fritz, Towards activity-aware tool support for change tasks, in: Proceedings of the International Conference on Software Maintenance and Evolution, 2017, pp. 171–182.

[11] A. N. Meyer, L. E. Barton, G. C. Murphy, T. Zimmermann, T. Fritz, The work life of developers: Activities, switches and perceived productivity, IEEE Transactions on Software Engineering 43 (12) (2017) 1178–1193.

[12] E. Murphy-Hill, R. Jiresal, G. C. Murphy, Improving software developers' fluency by recommending development environment commands, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM Press, 2012, pp. 42:1–42:11. doi:10.1145/2393596.2393645.

[13] K. Damevski, H. Chen, D. C. Shepherd, N. A. Kraft, L. Pollock, Predicting future developer behavior in the IDE using topic models, IEEE Transactions on Software Engineering (2018). doi:10.1109/TSE.2017.2748134.

[14] S. Amann, S. Proksch, S. Nadi, M. Mezini, A study of Visual Studio usage in practice, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Vol. 1, 2016, pp. 124–134.

[15] T. D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: A study of developer work habits, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, ACM, 2006, pp. 492–501.

[16] W. Maalej, M. Ellmann, R. Robbes, Using contexts similarity to predict relationships between tasks, Journal of Systems and Software 128 (2017) 267 – 284. doi:https://doi.org/10.1016/j.jss.2016.11.033.

[17] L. Bao, Z. Xing, X. Xia, D. Lo, A. E. Hassan, Inference of development activities from interaction with uninstrumented applications, Empirical Software Engineering 23 (3) (2018) 1313–1351. doi:10.1007/s10664-017-9547-8.

[18] K. Damevski, H. Chen, D. Shepherd, L. Pollock, Interactive exploration of developer interaction traces using a Hidden Markov Model, in: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, 2016, pp. 126–136.

[19] D. Gadler, M. Mairegger, A. Janes, B. Russo, Mining logs to model the use of a system, in: Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '17, IEEE Press, 2017, pp. 334–343. doi:10.1109/ESEM.2017.47.

[20] T. Roehm, W. Maalej, Automatically detecting developer activities and problems in software development work, in: Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 1261–1264.

[21] S. Chattopadhyay, N. Nelson, Y. Ramirez Gonzalez, A. Amelia Leon, R. Pandita, A. Sarma, Latent patterns in activities: A field study of how developers manage context, in: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE), 2019, pp. 373–383.

[22] M. Kersten, G. C. Murphy, Mylar: a degree-of-interest model for IDEs, in: Proceedings of the 4th international conference on Aspect-oriented software development, ACM, 2005, pp. 159–168.

[23] S. Zolaktaf, G. C. Murphy, What to learn next: Recommending commands in a feature-rich environment, in: Proceedings of the 14th International Conference on Machine Learning and Applications, 2015, pp. 1038–1044.

[24] W. Li, J. Matejka, T. Grossman, J. A. Konstan, G. Fitzmaurice, Design and evaluation of a command recommendation system for software applications, ACM Transactions on Computer-Human Interaction 18 (2) (2011) 6:1–6:35. doi:10.1145/1970378.1970380.

[25] R. Robbes, M. Lanza, Improving code completion with program history, Automated Software Engineering 17 (2) (2010) 181–212. doi:10.1007/s10515-010-0064-x.

[26] A. N. Meyer, T. Fritz, G. C. Murphy, T. Zimmermann, Software developers' perceptions of productivity, in: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, ACM, 2014, pp. 19–29. doi:10.1145/2635868.2635892.

[27] A. Yamamori, A. M. Hagward, T. Kobayashi, Can developers' interaction data improve change recommendation?, in: Proceedings of the 41st Annual Computer Software and Applications Conference, Vol. 1, 2017, pp. 128–137.

[28] W. Maalej, T. Fritz, R. Robbes, Collecting and Processing Interaction Data for Recommendation Systems, Springer Berlin Heidelberg, 2014, pp. 173–197.

[29] K. Damevski, D. C. Shepherd, J. Schneider, L. Pollock, Mining sequences of developer interactions in visual studio for usage smells, IEEE Transactions on Software Engineering 43 (4) (2017) 359–371.

[30] A. Ciborowska, N. A. Kraft, K. Damevski, Detecting and characterizing developer behavior following opportunistic reuse of code snippets from the Web, in: Proceedings of the 15th International Conference on Mining Software Repositories, ACM, 2018.

[31] S. Proksch, S. Amann, S. Nadi, Enriched Event Streams: A general dataset for empirical studies on in-IDE activities of software developers, in: Proceedings of the 15th Working Conference on Mining Software Repositories, MSR'18, 2018.

[32] S. Amann, S. Proksch, S. Nadi, FeedBaG: An interaction tracker for Visual Studio, in: Proceedings of the 24th International Conference on Program Comprehension, 2016, pp. 1–3. doi:10.1109/ICPC.2016.7503741.

[33] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[34] M. P. Robillard, G. C. Murphy, Automatically inferring concern code from program investigation activities, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, IEEE, 2003, pp. 225–234.

[35] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, C. Bräunlich, Developers' code context models for change tasks, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2014, pp. 7–18.

[36] K. Pearson, Liii. on lines and planes of closest fit to systems of points in space, The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 2 (11) (1901) 559–572.

[37] L. R. Rabiner, Readings in speech recognition, Morgan Kaufmann Publishers Inc., 1990, Ch. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, pp. 267–296.

[38] Y. W. Teh, M. I. Jordan, M. J. Beal, D. M. Blei, Sharing clusters among related groups: Hierarchical Dirichlet Processes, in: Advances in Neural Information Processing Systems, 2005, pp. 1385–1392.

[39] M. D. Hoffman, P. R. Cook, D. M. Blei, Data-driven recomposition using the Hierarchical Dirichlet Process Hidden Markov Model, in: Proceedings of the International Computer Music Conference, 2008.

[40] M. Beal, P. Krishnamurthy, Gene expression time course clustering with countably infinite Hidden Markov Models, in: Proceedings of the Conference on Uncertainty in Artificial Intelligence, 2006.

[41] E. B. Fox, E. B. Sudderth, M. I. Jordan, A. S. Willsky, An HDP-HMM for systems with state persistence, in: Proceedings of the 25th International Conference on Machine Learning, ACM, 2008, pp. 312–319.

[42] M. C. Hughes, W. T. Stephenson, E. Sudderth, Scalable adaptation of state complexity for nonparametric Hidden Markov Models, in: Advances in Neural Information Processing Systems, 2015, pp. 1198–1206.

[43] W. Li, J. Matejka, T. Grossman, J. A. Konstan, G. Fitzmaurice, Design and evaluation of a command recommendation system for software applications, ACM Transactions on Computer-Human Interaction (TOCHI) 18 (2) (2011) 6.

[44] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, S. Y. Philip, et al., Top 10 algorithms in data mining, Knowledge and Information Systems 14 (1) (2008) 1–37.

[45] M. Fernández-Delgado, E. Cernadas, S. Barro, D. Amorim, Do we need hundreds of classifiers to solve real world classification problems?, The Journal of Machine Learning Research 15 (1) (2014) 3133–3181.

[46] J. Matejka, W. Li, T. Grossman, G. Fitzmaurice, Community-commands: command recommendations for software applications, in: Proceedings of the 22nd annual ACM symposium on User interface software and technology, ACM, 2009, pp. 193–202.

[47] C. de Souza, D. Redmiles, An empirical study of software developers' management of dependencies and changes, in: Proceedings of the 30th International Conference on Software Engineering, IEEE, 2008, pp. 241–250.

[48] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, in: Proceedings of the 38th IEEE International Conference on Software Engineering, IEEE, 2016, pp. 297–308.

[49] X. Xia, D. Lo, E. Shihab, X. Wang, X. Yang, Elblocker: Predicting blocking bugs with ensemble imbalance learning, Information and Software Technology 61 (2015) 93–106.

[50] M. Shepperd, D. Bowes, T. Hall, Researcher bias: The use of machine learning in software defect prediction, IEEE Transactions on Software Engineering 40 (6) (2014) 603–616.

[51] H. W. Kuhn, The hungarian method for the assignment problem, Naval Research Logistics Quarterly 2 (1-2) (1955) 83–97.

[52] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, A. Horvath, To fix or to learn? how production bias affects developers' information foraging during debugging, in: Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), 2015, pp. 11–20.