Modeling Hierarchical Usage Context for Software Exceptions based on Interaction Data

Hui Chen, Kostadin Damevski, David Shepherd · Nicholas A. Kraft

Received: date / Accepted: date

Abstract Traces of user interactions with a software system, captured in production, are commonly used as an input source for user experience testing. In this paper, we present an alternative use, introducing a novel approach of modeling user interaction traces enriched with another type of data gathered in production - software fault reports consisting of software exceptions and stack traces. The model described in this paper aims to improve developers’ comprehension of the circumstances surrounding a specific software exception and can highlight specific user behaviors that lead to a high frequency of software faults.

Modeling the combination of interaction traces and software crash reports to form an interpretable and useful model is challenging due to the complexity and variance in the combined data source. Therefore, we propose a probabilistic unsupervised learning approach, adapting the Nested Hierarchical Dirichlet Process, which is a Bayesian non-parametric topic model commonly applied

H. Chen
Department of Computer and Information Science
Brooklyn College of the City University of New York
Brooklyn, NY 11210, U.S.A.
E-mail: huichen@ieee.org

K. Damevski
Department of Computer Science
Virginia Commonwealth University
Richmond, VA 23284, U.S.A.
E-mail: damevski@acm.org

D.C. Shepherd
ABB Corporate Research
Raleigh, NC 27606 U.S.A.
E-mail: david.shepherd@us.abb.com

N.A. Kraft
ABB Corporate Research
Raleigh, NC 27606 U.S.A.
E-mail: nicholas.a.kraft@us.abb.com
to natural language data. This model infers a tree of topics, each of whom describes a set of commonly co-occurring commands and exceptions. The topic tree can be interpreted hierarchically to aid in categorizing the numerous types of exceptions and interactions. We apply the proposed approach to large scale datasets collected from the ABB RobotStudio software application, and evaluate it both numerically and with a small survey of the RobotStudio developers.

**Keywords** Stack Trace, Crash Report, Software Exception, Software Interaction Trace, Hierarchical Topic Model

1 Introduction

Continuous monitoring of deployed software usage is now a standard approach in industry. Developers can use this monitoring data to discover and correct faults, performance bottlenecks, or inefficient user interface design. Many successful examples of this are the results of a debugging practice called “debugging in the large”, a postmortem analysis of large amount of usage data to recognize patterns of bugs \[16, 18\]. For instance, Arnold et al. use application stack traces to group processes exhibiting similar behavior called “process equivalence classes”, and identify what differentiate these classes with the aim to discover the root cause of the bugs associated with the stack traces \[3\]. Han et al. clusters stack traces and recognize patterns of stack traces to discover impactful performance bugs \[15\].

Aligned with the above, our work begins with the following observation. In software-as-a-service applications, monitoring data is gathered at the service host, while in user-installed software, relevant traces (or logs) are periodically transferred from users’ machines to a server. The granularity and format of the collected data (e.g., whether the data are accumulated in a raw/log form or as a set of derivative metrics) depend on the specific application and deployment. Two types of data commonly collected via monitoring include *software exceptions*, containing a stack traces from software faults that occur in production, and *interaction traces*, containing details of user interactions with the software’s interface.

In this paper, we provide a novel perspective on interpreting frequently occurring stack traces resulting from software exceptions by modeling them in concert with the user interactions with which they co-occur. Our model probabilistically represents stack traces and their usage context for the purpose of increasing developer understanding of specific software faults and the contexts in which they are manifested. Over time, this understanding can help developers to reproduce exceptions, to prioritize software crash reports based on their user impact, or to identify specific user behaviors that tend to trigger failures. Existing work attempts to empirically characterize software crash reports in various application domains \[8, 21, 22, 38\], but the use of interaction data has not yet been proposed for this purpose.

Interaction traces can be challenging to analyze. First, the logged interactions are typically low-level, corresponding to most clicks and key presses
available in the software application, and therefore the raw number of interactions in these traces can be large — containing millions of messages from many different users. Second, for complex software applications, many reasonable interaction paths are possible and often a specific high-level task can be accomplished with numerous interaction sequences. To address these two challenges of scale and of ambiguity in interpreting interaction traces, we require a probabilistic dimension reduction technique that can extract frequent patterns from the low-level interaction data.

Topic models are a dimensionality reduction technique with the capacity to discover complex latent semantic structures. Typically applied to large textual document collections, such models can naturally capture the uncertainty in software interaction data using probabilistic assumptions. However, in cases where the interaction traces are particularly complex, e.g., in complex software applications such as IDEs or CAD tools, applying typical topic models may still result in a large space that is difficult to interpret. The special class of hierarchical topic models encodes a tree of related topics, enabling further reduction in complexity and dimensionality of the original interaction data and improving the interpretability of the model. We apply a hierarchical topic modeling technique, called the Nested Hierarchical Dirichlet Process (NHDP) \[25\] to combine interaction traces and stack traces gathered from complex software application into a single, compact representation. The NHDP discovers a hierarchical structure of usage events that has the following characteristics:

- Provides an interpretable summary of the user interactions that commonly co-occur with the stack trace.
- Allows for differentiating the strength of the relationship between specific interaction trace messages and the stack trace.
- Enables locating specific interactions that have co-occurred with numerous runtime errors.

In addition, as a Bayesian non-parametric modeling technique, NHDP has an additional advantage. It allows the model to grow structurally as more data are observed. Specifically, instead of being imposed a fixed set of topics or hypotheses about the relationship of the topics, the model grows its hierarchy to fit the data, i.e., to “let the data speak” \[4\].

The main contribution of this paper are as follows:

- We apply NHDP to a large collection of interaction and stack trace data produced by ABB RobotStudio, a popular robot programming platform developed at ABB Inc, and examine how effective it extracts latent thematic structures of the data set and how well the structure depicts a meaningful context for exceptions occurring during production use of RobotStudio.
- This is a first work that group users’ IDE interaction traces with stack traces hierarchically and probabilistically into “clusters”. These “clusters” provide usage contexts of stack traces. Since a stack trace may be the results of multiple usage contexts, this approach avoids the shortcoming to deterministically assign a stack trace to a single usage context. Instead, it associates a stack trace with multiple usage context probabilistically.
The remainder of this paper is organized as follows. Section 2 introduces the types of interaction and stack trace data we use and how these data sources are prepared for topic modeling. We describe the hierarchical topic modeling technique and its application to software interaction and crash data in Section 3. We apply the modeling technique to the large RobotStudio dataset and provide an evaluation in Section 4. In Section 5, we describe relevant related research and conclude this paper in Section 6.

2 Background

Interaction data gathered from complex software applications, such as IDEs, typically consists of a large vocabulary of messages, ordered in a time series. The data is typically collected exhaustively, in order to capture user actions in an interpretable, logical sequence. As users complete certain actions much more often than others, the occurrence of interaction messages follows a skewed distribution where few messages appear very often, while many occur relatively infrequently. Some of the messages are direct results of user actions (i.e., commands), while others may reflect the state of the application (i.e., events), such as the completion of a background task like a project build. Consider the below snippet of interactions, gathered in Visual Studio, part of the Blaze dataset [11,30].

2014-02-06 17:12:12 Debug.Start
2014-02-06 17:14:14 Build.BuildBegin

Fig. 1 Interaction traces that contain the same stack trace (left) are aggregated into a model similar to the one described in this paper (right). The model aggregates a collection of interaction traces coupled with stack traces into a hierarchy of topics (or contexts). Each topic expresses a set of interaction messages with different probabilities, depicted via text size in this figure.

Interaction data gathered from complex software applications, such as IDEs, typically consists of a large vocabulary of messages, ordered in a time series. The data is typically collected exhaustively, in order to capture user actions in an interpretable, logical sequence. As users complete certain actions much more often than others, the occurrence of interaction messages follows a skewed distribution where few messages appear very often, while many occur relatively infrequently. Some of the messages are direct results of user actions (i.e., commands), while others may reflect the state of the application (i.e., events), such as the completion of a background task like a project build. Consider the below snippet of interactions, gathered in Visual Studio, part of the Blaze dataset [11,30].

Fig. 1 Interaction traces that contain the same stack trace (left) are aggregated into a model similar to the one described in this paper (right). The model aggregates a collection of interaction traces coupled with stack traces into a hierarchy of topics (or contexts). Each topic expresses a set of interaction messages with different probabilities, depicted via text size in this figure.

Interaction data gathered from complex software applications, such as IDEs, typically consists of a large vocabulary of messages, ordered in a time series. The data is typically collected exhaustively, in order to capture user actions in an interpretable, logical sequence. As users complete certain actions much more often than others, the occurrence of interaction messages follows a skewed distribution where few messages appear very often, while many occur relatively infrequently. Some of the messages are direct results of user actions (i.e., commands), while others may reflect the state of the application (i.e., events), such as the completion of a background task like a project build. Consider the below snippet of interactions, gathered in Visual Studio, part of the Blaze dataset [11,30].

The Eclipse UDC dataset is a well known source of this type of data in the software engineering community. Available at: [http://archive.eclipse.org/projects/usagedata/](http://archive.eclipse.org/projects/usagedata/)
The developer that generated the above interaction log is starting the interactive debugger, observed by the `Debug.Start` command. This triggers an automatic build in Visual Studio, shown by the `Build.BuildBegin` and `Build.BuildDone`, the exact same log messages that appear when the build is triggered explicitly by the user. After the debugger stops at a breakpoint, `Debug.Debug Break Mode`, this developer enables two previously disabled breakpoints (e.g., `Debug.EnableBreakpoint`) and restarts (or resumes) debugging (such as, `Debug.Start` and `Debug.Debug Run Mode`).

In this paper, we leverage a probabilistic approach where each extracted high-level behavior is represented as a probability distribution of interaction messages. This type of model is able to capture the noisy nature of interaction data [31], which stems from the fact that 1) numerous paths that represent a specific high-level behavior exist (e.g. using `ToggleBreakpoint` versus `EnableBreakpoint` has the same effect) and 2) unrelated events may be recorded in the midst of a set of interactions (e.g. `Debug.BuildDone` can occur at various intervals from `Debug.BuildStart` and interspersed with various other messages).

One particular application domain where probabilistic models have been effective for extracting high-level context, or topics, is natural language processing. In natural language processing words are the most basic unit of the discrete data and documents can be represented as sets of words (i.e., a "bag" of words assumption). We can draw an analogy from the characteristics of interaction traces to natural language text, i.e., interaction traces exhibit naming relations such as synonymy and polysemy similar to those in natural language texts. A trace often contains multiple different messages that share meaning in a specific behavioral context, e.g., both the `ToggleBreakpoint` and `EnableBreakpoint` events 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, e.g., 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.

Figure 1 shows an example of two IDE traces containing both interactions and stack traces from the ABB RobotStudio IDE. Both of these traces correspond to user writing a program using a programming language called RAPID into this environment’s editor, and performing common actions like cutting-and-pasting and cursor movement (i.e., `EditCut`, `EditPaste`, and `ProgramSetCursor`). In both trace excerpts the users encountered the identical exception, `RobApiException`
RAPID symbol was not found, as identified by its type and message. While corresponding to the same high-level user behavior, the sequence and constituent messages occurring in the two interaction traces are slightly different. The modeling approach described in this paper is able to capture the common interaction context of RobApiException, forming high-level user behaviors that are represented as a probabilistic distribution of interaction messages, shown in the right part of Figure 1. The model is able to overcome the slightly different composition and order in the two interaction traces, extracting their commonalities, and can be used to help better characterize and understand the context of the shown exception’s stack trace.

The above motivates us to seek an algorithm to find not only useful sets of patterns of user behaviors, and learn to organize the these patterns according to a hierarchy in which more generic or abstract patterns near the root of the hierarchy and more concrete patterns are near the leaves. This hierarchy would allows us to explore stack traces and associated use context in a way no different from what we do in our daily lives, such as, when we go to a grocery store, we begin with a particular section, and then down to a specific aisle, finally locate a particular product. This leads us to a set of topic models called hierarchical topic models.

2.1 Topic Models for Interaction Data

Given a collection of natural language documents, topic modeling allows one to discover latent thematic structures in the document collection (commonly called a corpus). A document in the corpus is an unordered set of words (i.e., “a bag of words”). The vocabulary of the corpus, denoted as $V$, consists of the $|V|$ unique words in the corpus. The extracted thematic structures are expressed as topics, which are a set of discrete probability distributions over interaction messages, and their inter-relationship. For instance, given the vocabulary of a collection of documents, denoted as $V = \{m_1, m_2, \ldots, m_n\}$, a topic can be expressed as a probability mass function, $P(m = m_i) = P_{m_i}$, where $0 \leq P_{m_i} \leq 1$, $\sum_{i=1}^{|V|} P_{m_i} = 1$. The relationship among the topics can be expressed in many ways. For instance, in Latent Dirichlet allocation (LDA), a frequently used flat topic model, the thematic structures include the proportions of each topic exhibited in the collection or in a specific document in the collection.

Topic models are readily applied to other types of data because they do not rely on any natural language specific information or assumptions, such as, grammar. Examples of data types other than textual data where topic modeling was successfully used include collections of images, genetic information, and social network data [6, 27, 35]. In particular, when we examine a small segment of an interaction trace, we find that the number of interaction types is small and the segment consists of usually highly regular and repetitive patterns. This is expected, as within a small period of time, a user is likely focusing on a specific task and interacting with a small subset of the development environment which consists of relatively few interactions. In addition, interaction
traces exhibit two naming relations, namely synonymy and polysemy that are found in natural texts. The former refers to that a user can use a command to complete multiple types of tasks, and the later that a task can be accomplished via different types of commands [11]. We posit that the above described regular behavior and naming relations between the interaction types within small units of IDE usage time mimics the “naturalness” of text writing [19], and suggests that models used for analyzing natural language text might apply to IDE interaction data. It follows that we consider interaction traces consist of many different types of messages, each corresponding to an event, a command, or a stack trace, all of which constitute the “vocabulary” of the collection of interaction traces. That is to say, in this paper, interaction trace messages are the words, windows of interactions messages are the documents, and all of the observed windows are the corpus of the study.

Interaction traces consists of many low-level messages corresponding to 1) user actions and commands (e.g. copying text into the clipboard, pasting text from the clipboard, building the project); and 2) events that occur asynchronously (e.g. completion of the build, stopping at a breakpoint when debugging). The sequential order between the messages is also very relevant to some behaviors, but not to others. For instance, the event indicating the completion of the build may be important to the next set of actions the developer performs, or it may be occurring in the background without import.

In our model, following the “bag of words” assumption, we use a tight moving window of interaction messages generated by an individual developer, but ignore the message order within the window. This is a reasonable modeling assumption that captures sequential order but resilient to small permutations in message order within the window. In addition, developer interaction traces often contain large time gaps, stemming from breaks in the individual developer’s work. To take account of these we force window breaks when the time between two consecutive messages exceeds a predefined threshold. An interaction window is a sequence of \( N \) messages denoted as \( \mathbf{m} = (m_1, m_2, \ldots, m_N) \) where \( m_N \) is the \( N \)-th message in the sequence. A corpus is a set of \( M \) windows, denoted as \( \mathbb{D} = \{\mathbf{m}_1, \mathbf{m}_2, \ldots, \mathbf{m}_M\} = \mathbf{m}_{1:M} \) where \( M = |\mathbb{D}| \).

Software exceptions and stack traces, reporting a software fault, which may or may not be fatal and result in the software to crash, commonly contain a time stamp and some type of user/machine identifier that allows them to be correlated with interactions from the same user. We use a dataset that interleaves the interactions with the stack traces. Minor timing issues in relating interaction and software crash data are considered unimportant in the window-based modeling technique we use, as long as the crash is correlated with the relevant window of interaction messages. Assuming this reasonable assumption holds, we treat the stack trace as just another message in the interaction log, i.e., the “vocabulary” becomes \( \mathbb{V} = \{m_1, m_2, \ldots, m_n, s_1, s_2, \ldots, s_p\} \), where \( m \) is an interaction message and \( s \) is a stack trace.
3 NHDP for Interaction Data

The scale of IDE interaction traces collected from the field can pose a challenge to analysis. The size of the traces can grow quickly and become very large, for instance, the Eclipse Foundation Filtered UDC Data set consists of on the order of $10^7$ messages a day. Our approach is to divide the traces into message windows. To accomplish this, we first divide the traces into active sessions, using a prolonged time gap between messages as a delimiter, and further divide each session into one or more windows, each of which is a sequence of a fixed number of messages. Stack traces are interleaved in the interaction log, and are treated as ordinary messages in the window in the model. In the remainder of the paper, to be consistent with prior literature on topic models, we sometimes refer to a message window as a document, and messages within that window as words.

Our “windowing” approach bears similarity to the data processing method commonly used for streaming text corpora, such as, transcripts of automatic speech recognized streaming audio, transcripts of closed captioning, and feeds from the news wire [5]. Among these kinds of datasets, no explicit document breaks exist. A common approach is to divide the text into “documents” of a fixed length, as we have.

Most topic models, such as LDA, are flat topic models, in which the topics are independent and there is no structural relationship among the discovered topics. There are two challenges facing flat topic models. First, it is difficult or at least computationally expensive to discover the number of topics that we should model in a document collection. Second, since there is only a rudimentary relationship among topics, the meaning of the topics is difficult to interpret, in particular, when multiple topics may look alike based on their probability distributions.

We use a hierarchical topic model based on the Nested Hierarchical Dirichlet Process (NHDP), which, compared with a flat topic model, arranges the topics in a tree where more generic topics appear on upper levels of the tree while more specific topics appear at lower levels. We can achieve two objectives via a hierarchical topic model. The number of topics for a model can be easily expressed in the hierarchy, much like the hierarchical clustering algorithm where the number of clusters can be determined by increasing the depth and the branches of the tree of clusters. In addition, the very structure among the topics, i.e., more generic topics appearing on upper levels of the tree and more specific topics on lower levels can lead to improved human interpretability. As argued in [4], “if interpretability is the goal, then there are strong reasons to prefer” a hierarchical topic model, such as, NHDP over a flat topic model, such as, LDA.

A number of hierarchical topic models have been proposed in the literature. We choose the Nested Hierarchical Dirichlet Process (NHDP) [4] as it possesses several advantages over other popular hierarchical models, such as Hierarchical Latent Dirichlet Allocation (HLDA) [4]. Different from these models, NHDP results in a more compact hierarchy of topics (less branching) and produces
less repetitive topics as it allows a document to sample topics from a subtree that is not limited to a path from the root of the tree. NHDP is relevant model for analyzing IDE interaction traces as even for stack traces that occur at many different interaction contexts, NHDP allows for capturing variability effectively at higher (more general) levels of its hierarchy.

To understand how we may apply the NHDP topic model to analyze software interaction traces, we illustrate the model in Figure 2 as a directed graph, i.e., a Bayesian network. Since NHDP is a Bayesian model, it starts with a prior. In effect, the NHDP topic model is named after its prior, the nested hierarchical Dirichlet process. The prior expresses our assumption that the topics should be related in a tree-like structure and our assumption that a topic is branched to more specific topics. These assumptions are controlled by a number of parameters that are provided as input to the model (i.e., the prior to the model), commonly referred to as the hyperparameters of the model. We provide an overview of these hyperparameters and their relationship with other variables in the graph in Figure 2.

In NHDP, we consider words in documents to follow Multinomial distributions, given a topic. It follows that we consider topics themselves to be drawn from Dirichlet distributions, which is often used as a prior for Multinomial distributions. As shown in Figure 2, given a hyperparameter $\eta$ as the parameter for a Dirichlet distribution, we draw potentially infinite number of topics, denoted as $\theta_k$, $k = \{1, 2, \ldots\}$ in Figure 2. Since, in this paper we choose a symmetric Dirichlet distribution for generating topic distributions, hyperparameter $\eta$ is a positive scalar, and represents the concentration parameter of the Dirichlet distribution $\text{Dir}(\eta)$. The smaller $\eta$ is, more concentrated on fewer words we believe a topic to be.

A topic corresponds to a node in global topic tree $T$. The global topic tree $T$ can be either drawn using a nested Chinese Restaurant Process as illustrated in [4] or constructed directly using a nested Stick Breaking Process as shown in [25]. Both of these two methods yield an infinite set of Dirichlet processes, each corresponding to a node in the tree. A Dirichlet process, an infinitely decimated Dirichlet distribution, allows us to branch from a topic node to an infinite number of child topic nodes, which constitutes the mechanism to build the topic tree. A Dirichlet process is a distribution from which a draw is also a probability distribution. We denote drawing a probability distribution $G$ from a Dirichlet process as $G \sim \text{DP}(\alpha \text{H})$ where concentration parameter $\alpha$ and base measure $\text{H}$ are two hyperparameters as shown in Figure 2. The probability distributions drawn from the Dirichlet process provide a parameter to associate a node in the topic tree to its corresponding topic ($\theta_k$). The concentration parameter $\alpha$, where $\alpha > 0$ represents our belief on how topic node should be branched to next level. The greater the $\alpha$, the more branches we should expect from a corpus.

When examining the relationship of the topics, we know that the topics are related in the manner that document trees are derived. A document tree $T_d$ is a copy of the corpus topic tree with the same topics on the nodes but different branching probabilities. As discussed above, NHDP is characterized by the
nested hierarchical Dirichlet process, and each node in the global tree has a corresponding Dirichlet process. Let’s denote the Dirichlet process at a node $n$ in the global tree $T$ as $G_T^n$, the corresponding node in the topologically identical document topic tree for document $d$ has a Dirichlet process $G_d \sim DP(\beta G_T^n)$, where the concentration parameter $\beta$ controls our belief on how a document branches in the corresponding document tree, i.e., hyperparameter $\beta$ controls how the branching probability mass is distributed among branches. The higher the $\beta$, the less concentrated the branching probability mass, and in effect, the more branches we should expect from a corpus. For instance, if we expect a document in the corpus should branch to very small number of topics in next level, all the while these topics are expected to be different among different documents, we should begin with a large $\alpha$ and a small $\beta$ because we expect effectively a large global tree, but small document trees.

Furthermore, each word in a document is associated with a topic. A word’s topic is indexed by $c_{d,n}$ for the $n$-th word in document $d$ in Figure 2. The topic for the word is chosen following a two-step approach. First, we choose a path from the root in the document tree $T_d$ based on the tree’s branching probabilities. Next, we select a topic along the path for the word based on a probability distribution — starting from the root along the path, we draw $U_d$ from Beta distribution $Beta(\gamma_1, \gamma_2)$, and $U_d$ is the probability that we remain on the node, and $1 - U_d$ is the probability that we switch to next node along the path. The two parameters control the expected range of the level switching probabilities. The Beta distribution is chosen because it is commonly used to express a probability distribution of probabilities.

These hyperparameters have an impact on the learned NHDP model and inference of new documents. In Section 4, we evaluate how sensitive the learned NHDP model is to the hyperparameters. An insensitive model has stronger ability to correct inaccurate hyperparameter priors by learning what the data implies.

3.1 Learning the NHDP Model

To learn a NHDP model from a document corpus, we adopt the stochastic inference algorithm in [25], that is organized in the following steps:

1. Scan the documents from the training corpus, and extract words to form a vocabulary of the training corpus. In this step, the vocabulary consists of IDE messages and stack traces. A stack trace is treated as a single word. Denote the vocabulary as $V$ that consists of $|V|$ unique words.
2. Index words in the vocabulary from 0 to $|V| - 1$, and convert each document to a term-frequency vector where the value at position $i$ is the frequency of the word indexed by $i$ in the document.
3. Randomly select a small subset of documents from the training corpus, denote the set of documents as $D_I$. The random selection of documents will not stop until any word in the vocabulary appears at least once in the selected documents.
4. Running the $K$-means clustering algorithm repeatedly against $D_I$ to build a tree of clusters.
5. Initialize a NHDP tree for $D_I$, call the initial NHDP topic tree as $T_I$, and let $T_R = T_I$.
6. Randomly select a subset of documents from the training corpus, denote the set of documents as $D_R$.
7. Make adjustment to $T_R$ based on an inference algorithm against $D_R$. The result is a topic tree $T$.
8. Repeat steps [4 and 5] until $T$ converges.

From steps [3 to 5] we provide the maximum height and the maximum number of nodes at each level of tree $D_I$. The maximum height and number of nodes at each level should be greater than the final tree. Following the assumption that words are interchangeable, a document is expressed as a vector where each element is the frequency of the corresponding word appearing in the document. In Step 4, we use the $K$-means clustering algorithm to divide the documents into a number of clusters, and for each cluster, we estimate a topic distribution. These clusters and the topic distributions are the top level nodes in tree $D_I$ just beneath the root. We then repeat the process for each cluster, and each cluster is further divided into a number of subclusters. For each subcluster we estimate a topic distribution. This step is for computational efficiency. Given the number of clusters and the depth of the tree, the $K$-means algorithm builds a large tree quickly. This tree serves as the initial tree for the NHDP algorithm that learns the switching probabilities for different levels and the switching probabilities for different clusters at a level, which effectively shrinks the tree by learning the switching probabilities. Note in the above when applying the $K$-means algorithms, we adopt the $L_1$ distance, i.e., given two documents represented as two vectors $d_i$ and $d_j$, the distance of the two documents is $d(d_i, d_j) = \sum_{k=0}^{V-1} |d_{ik} - d_{jk}|$.

Steps [6 to 8] perform a randomized batch inference processing. Agrawal et al. demonstrate that topic modeling can suffer from “order effects”, i.e., a topic modeling algorithm yields different topics when the order of the training data is altered [2]. This randomized batch processing can reduce this “order effects” via averaging over many different random order of the training data set. Step 7 requires a specific inference algorithm. In [4, 33], Markov Chain Monte Carlo algorithms, specifically, Gibbs samplers are used. In this work, we used the variational inference algorithm in [25]. Variational inference algorithms are typically shown to scale better to large data sets than Gibbs samplers do. Steps [6 to 8] can begin with an arbitrary tree, however, it is much more computationally efficient to initialize the inference algorithm with a tree that shares statistical traits with the target data.

4 Evaluation

For evaluation, we use field interaction traces from ABB RobotStudio, a popular IDE intended for robotics development that supports both simulation and
The probabilistic graphical model of NHDP. The model is a directed graph, i.e., a Bayesian network. There are 3 plates in the graph, the topic plate that represents potentially infinite number of topics, the document plate for a document where the number of words in the document is denoted as $N_d$ for document $d$, and the corpus plate that consists of $M$ documents. The $n$-th word in document $d$, $w_{d,n}$ is the only observable variable in the model. For $n$-th word in document $d$, a topic indicator is drawn based on the topic tree $T_d$ and the switching probabilities $U_d$, where $T_d$ is drawn from global topic tree $T$ and $U_d$ is drawn from a Beta distribution with two hyperparameters $\gamma_1$ and $\gamma_2$.

physical robot programming using a programming language called RAPID. The interaction traces are collected for RobotStudio itself, and not of the robot application being developed by this IDE. The RobotStudio dataset we used represents 25,724 users over a maximum of 3 months of activity, or a total of 76,866 user-work hours. In the interaction traces, there are 7,425 unique messages, 134 types of exceptions, 1,975,474 sessions, and 2,251 unique stack traces, resulting in 1,978,081 documents of 50 messages. Note that a single exceptions in RobotStudio is often triggered by numerous users of the IDE, as such, an exception corresponds to many unique stack traces and each unique stack trace has many copies. We chose the window size of 50 messages based on empirically observing this to result in semantically interesting windows, which commonly represent a single activity by a developer [10].

The RobotStudio data consist of sequences of time-stamped messages, where each message corresponds to a RobotStudio command (e.g., RapidEditorShow) or an event representing application state (e.g., Exception and StartingVirtualController). Messages have a few additional attributes, such as the component that generates the command or the event, and the command or event type. For RobotStudio, the stack traces are embedded directly into the interaction log, so the two distinct data types considered in this paper are already combined.

The evaluation plan is as follows. First, we conduct a “held-out” document evaluation, i.e., we divide the documents into two sets, training dataset to learn the model, and held-out dataset to test the model. The purpose of the held-out document evaluation are two-fold. We want to know whether the training data set is sufficient to produce a stable model and to assess whether the parameters used in the learning process is reasonable. Second, we conduct a user survey
Fig. 3 Processing pipeline. (a) When forming the corpus, we divide interaction traces into sessions, and each session one or more windows. A type of message or a stack trace is a word. Scanning the windows, we obtain the vocabulary of the corpus. To improve computational efficiency and numerical stability, we remove the words that are overly frequent and those too rare. (b) When learning the model, we divide the corpus into the training dataset and the testing (held-out) dataset, and start with an initial set of parameters to infer a model, and vary these parameters. For each set of parameters, we obtain a model. (c) Next is to determine parameters. Via computing perplexity on the held-out dataset, we determine whether the model obtained converges and whether the model is sensitive to parameters, which informs us an appropriate set of parameters and use the parameters, we obtain a model for evaluation. (d) A survey is constructed based on a few randomly selected stack traces and their usage contexts. We evaluate the quality of the model by analyzing the responses of the developers to the survey. Although presented linearly, the pipeline is iterative.

to assess the usefulness of the model in understanding and debugging software faults. The entire processing pipeline is given in Figure 3.

4.1 Held-out Document Evaluation

Unsupervised learning algorithms, like NHDP, are typically more challenging to evaluate, as there is no ground truth to compare to. Perplexity and predictive likelihood are two standard metrics for informational retrieval evaluation that corresponds to a model’s ability to infer an unseen document from the same corpus. These two are a single metric in two different representation since perplexity is, in effect, the inverse of the predictive power of the model. The worse the model is, the more perplexed it is with unseen data, resulting in greater values for the perplexity metric. Similarly, the better the model is, the more likely that the model is able to infer the model of an unseen document. To further explain these two concepts and their relationship and how they may be computed, let us divide the dataset into two subsets, one is a training dataset that is considered as observed, and the other a held-out dataset that is considered as unseen. We denote the former as $D_{obs}$ and later as $D_{held-out}$. We consider $D_{obs}$ has $N_{obs}$ documents, and $D_{obs} = \{d_{obs,1}, d_{obs,2}, \ldots, d_{N_{obs}}\}$, and $D_{held-out}$ has $N_{held-out}$ documents, and $D_{held-out} = \{d_{held-out,1}, d_{held-out,2}, \ldots, d_{held-out,N_{held-out}}\}$. Given that we learn a model $M$ from the training dataset $D_{obs}$, we define the predictive power of the learned model is the following conditional probability, i.e., the probability of observing the unseen documents given the model learned from the observed document,
\[ P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) = P(d_{\text{held-out,1}}, \ldots, d_{\text{held-out,}\text{N}_{\text{held-out}}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M}) \]

\[ = \prod_{i=1}^{\text{N}_{\text{held-out}}} P(d_{\text{held-out,i}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M}) \] (1)

where held-out documents are considered independent to one another.

Since the probability in equation (1) varies on the size of the held-out dataset, \( \text{N}_{\text{held-out}} \), the probability is not comparable for held-out datasets of different sizes. To make it comparable among held-out dataset of different sizes, we take a geometric mean of the probability as follows,

\[ P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) = P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) \left( \prod_{i=1}^{\text{N}_{\text{held-out}}} P(d_{\text{held-out,i}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M}) \right) \] (2)

where \( |d_{\text{held-out,i}}| \) is the sum of all word counts in document \( d_{\text{held-out,i}} \).

We call \( P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) \) the predictive likelihood of the model \( \mathcal{M} \) on the unseen dataset \( D_{\text{held-out}} \). We can then define the predictive log likelihood as,

\[ \mathcal{L}(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) = \log P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) \]

\[ = \log P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) \left( \prod_{i=1}^{\text{N}_{\text{held-out}}} P(d_{\text{held-out,i}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M}) \right) \]

\[ = \sum_{i=1}^{\text{N}_{\text{held-out}}} \frac{1}{|d_{\text{held-out,i}}|} \log P(d_{\text{held-out,i}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M}) \] (3)

and define the perplexity as the inverse of the predictive likelihood,

\[ \text{Perplexity}(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M}) = \frac{1}{P(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M})} \]

\[ = \frac{1}{\prod_{i=1}^{\text{N}_{\text{held-out}}} P(d_{\text{held-out,i}}|d_{\text{obs,1}}, \ldots, d_{\text{obs,}\text{N}_{\text{obs}}}, \mathcal{M})} \]

\[ = e^{\frac{L(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M})}{\sum_{i=1}^{\text{N}_{\text{held-out}}} \frac{1}{|d_{\text{held-out,i}}|}}} \]

\[ = e^{-L(D_{\text{held-out}}|D_{\text{obs}}, \mathcal{M})} \] (4)
which establish the correspondence between perplexity and predictive log likelihood.

In the following, we describe the procedure to compute the perplexity and show the result. This evaluation method, inspired by earlier work in [29,34], is frequently used to evaluate topic models, such as in [24,25].

1. **Form training and testing datasets.** We divide interaction traces into a training dataset and a testing dataset based on a reasonable ratio \( r_{td} \), e.g., 0.9. To obtain the training dataset, randomly select \( r_{td}M \) documents from the \( M \) documents of interaction traces. The remaining \( (1 - r_{td})M \) documents are in the testing dataset.

2. **Form observed dataset and held-out dataset.** Select a document partition ratio \( r_{dp} \), e.g., 0.9. For each document \( d \) in the testing dataset, and the \( F_d \) appearances of words in the document, partition \( F_d \) into two partitions. The first \( r_{dp}F_d \) words goes to the first partition, and the second \( (1 - r_{dp})F_d \) words the second partition. Consider the two partitions as two documents, \( d_h \) and \( d_o \). Then all the \( d_h \) form the held-out dataset and all the \( d_o \) forms all the observed dataset, i.e., we obtain \( D_{\text{held-out}} \) and \( D_{\text{obs}} \) in equation (4).

3. **Train the model.** Use NHDP on the training dataset, i.e., infer the global topic tree \( T \) using the training dataset. The model is \( M \) in equation (4).

4. **Compute perplexity.** Use the definition in equation (4).

Figure 4 is a result of the perplexity obtained when we gradually increase the number of documents seen and the use the rest as the testing data. We take an approach inspired by \( N \)-fold cross-validation. For each training dataset size, we randomly select the training dataset from the collected dataset and then compute the perplexity. We illustrate 10 computed perplexities at each training dataset size in an \( x-y \) plot with error bar in Figure 4. The figure shows that both the perplexity and the variation of the perplexity decreases as training dataset size increases, indicative of the convergence of the algorithm and a stable model. In particular, when the document seen is at 40% of documents, we observe a significant drop of perplexity, and the magnitude of the drop is consistent with those in the topic modeling literature, such as, [24,25]. This suggests that the obtained model has converged to a stable state that reflects the underlying data, and can be used for our purpose of interpreting the context of software exceptions.

4.2 Sensitivity Analysis

As a Bayesian hierarchical model, for NHDP, we infer marginal and conditional probability distributions from the data for the many parameters in the model, as such, the model does not overfit. As a non-parametric model, we parametrize the model with infinite number of parameters, as such, the model does not underfit [15 page 101].
However, a Bayesian non-parametric model is established with hyperparameters whose values may be difficult to choose. To assess the effect of these values, a common evaluation method is sensitivity analysis. This is particularly important for Bayesian hierarchical models [28]. For sensitivity analysis, we examine how the hierarchy obtained varies with hyperparameters in the prior. Their values control the base distribution in the NHDP process, and the switching probabilities between levels of the tree. For a document, the topic at a node is drawn from a Dirichlet distribution, in particular, drawn from $\text{Dir}(\eta)$, a symmetric Dirichlet distribution controlled by the concentration parameter $\eta$. However, we need to choose which branch to visit to draw topics for its children, which is controlled by the hyperparameter $\beta$. When we generate a document, we decide whether to go to next level of the tree based on Beta distribution, $\text{Beta}(\gamma_1, \gamma_2)$. The effects of these parameters are discussed in Section 3.

A number of statistics can be used to evaluate how sensitive the learned model is to the hyperparameters. These statistics include the number of topics at each level of the tree for each document and the number of words at each topic. Figure 5 shows the average number of topics per document at tree levels 1, 2, and 3 when we increase hyperparameter $\beta$ from 0.1 to 1.0 when we infer the model from a set of 40% of randomly selected documents. The graph clearly shows that the inferred model is insensitive to the hyperparameter $\beta$.

Figure 6 shows the average number of topics per document at tree levels 1, 2, and 3 when we increase hyperparameter $\gamma_1$ from 0.2 to 1.0 and hold $\gamma_1 + \gamma_2 = 2$. It shows that the model is somewhat sensitive to $\gamma_1$ and $\gamma_2$. However, the variation of the number of topics is mostly less than 10%, which
is not a major change, particularly for the average number of topics at levels 2 and 3.

In summary, these sensitivity tests indicate that the inferred model is robust as it tolerates uninformed selections of hyperparameters. The tree structure is affected, but only in a minor way, by modifications to the hyperparameters. However, a specific caution is that one should choose $\gamma_1$ and $\gamma_2$ with more care than do $\beta$. Practically, one may compare the perplexities at different values of $\gamma_1$ and $\gamma_2$, and elect the pair with lower perplexity.

![Fig. 5 The average number of topics per document at tree levels 1 – 3 versus hyperparameter $\beta$. The graph indicates the desired characteristic of the model that it is insensitive to the hyperparameter $\beta$ of the prior.](image)

### 4.3 Example RobotStudio Topic Hierarchy

The result of our approach is a topic hierarchy learned from the combined interaction and software crash dataset. The tree hierarchy communicates a succinct model of the observed interactions, where each topic represents a group of commonly co-occurring interactions and the hierarchy encodes a relationship between general or popular topics and ones that are more specific and rare.

One may explore the hierarchy either bottom-up or top-down to observe its structure, or begin with a specific event, such as an exception or stack trace, and move in both directions with the idea of understanding the context of user behavior that produces the exception. For instance, Figure 7(b) shows a topic hierarchy learned from the dataset centered on an exception. The hierarchy shows a parent topic and two of its child topics. Since the messages with dominant probabilities are about simulation, the parent topic can be interpreted
Fig. 6 The average number of topics per document at tree levels 1 – 3 versus $\gamma_1$. When we vary $\gamma_1$, we hold $\gamma_1 + \gamma_2 = 2$. When $\gamma_1 = \gamma_2 = 1$, the Beta distribution becomes a uniform distribution in $[0,1]$. As $\gamma_1$ increases, we become less likely to draw smaller probabilities from the Beta distribution, which results in words more likely to stay on the current level of the tree.

to indicate that developers are starting, stopping, and stepping through a simulation using RobotStudio. The two child topics exhibit two sub-interactions when the user is doing the simulation. The first child, illustrated immediately below its parent indicates that the user conducts a conveyor simulation. The second child indicates that the simulation includes the user’s action that leads the simulated robot moving to a different location, which is often accompanied with saving project state, perhaps, because it is prudent to save the project state before a path change. Thus, we may conclude that this topic hierarchy suggests that the user starts with a more generic activity, simulating a robot, and the simulation consists of multiple sub-interactions. It also shows that the exception indicated by the message ...RobApiException... often occurs with the simulation for controlling a conveyor.

4.4 RobotStudio Developer Survey

In order to assess the interpretability and value of our technique, we conducted a survey of RobotStudio developers using the model we extracted from the user interaction dataset of this application. Note that they are the individuals who develop and maintains RobotStudio, and are not users who use RobotStudio in production and from who the interaction data are collected. One important goal is to help the developers from using the model built from the data collected from the users. The survey consisted of a sample of five random RobotStudio exceptions that were displayed one at a time together with their surrounding context hierarchy.
The composed survey was sent via e-mail to the entire RobotStudio development team, consisting of 17 individuals, out of which we received 6 responses. All but one of the respondents had 3 or more years of experience on the RobotStudio team and all of them had worked as software developers for at least 3 years. Five out of six respondents were familiar with the RobotStudio interaction dataset, and had examined it in the past, and all of them believed that knowing which commands in the interaction log an exception co-occurs with could be helpful in debugging. Figure 7 displays two of the images shown in the survey, which depict an exception and its nearby surrounding command context hierarchy. Below, we highlight the salient conclusion from the study, coupled with the evidence to support them, including any additional relevant explanation extracted from open-ended questions in the survey.

The model was very useful for understanding and debugging some exceptions, but not useful for others. The survey showed a strong variance between the responses for the usefulness of specific parts of the model and specific exceptions. For instance, for RobApiException, listed in Figure 7(b), the respondents rated the usefulness of the usage context in understanding the exception an average of 7.83 (s = 1.52) on a scale of 1 (least useful) to 10 (most useful). This high rating can be contrasted to the usefulness rating received by the usage context of the remaining 4 exceptions: FormatException - 4.0/10.0 (s = 2.83); ApplicationException - 3.66/10.0 (s = 3.44); KeyNotFoundException - 4.0/10.0 (s = 3.83); and GeoException - 4.0/10.0 (s = 2.92). Three of the developers already formed the same hypothesis for the fault by examining the model for RobApiException, stating...
the following:

```
[...] VC returns an error saying that we cannot set the program pointer to main
in the current execution state. Perhaps RobotStudio tries to move the program
pointer when it is in running state.
```

For the less useful exception models, a number of the RobotStudio developers suggested a concrete set of improvements that they believed would raise its level of usefulness, including labeling each of the contexts and providing additional command characteristics, whenever available, to make the model clearer. For instance, one participant stated:

“It’s like watching the user over the shoulder but too far away. I can see which
tools and windows he or she opens, which commands are issued. But I cannot
see any name of an object, no version number of a controller, no file name,
not really anything concrete and specific. I think that needs to be tied in.”

Additionally, the survey result that some exceptions are more useful while
the others are not based on the users’ ratings may be in part attributed
to the following observation. Some exceptions, e.g., `FormatException`
and `KeyNotFoundException` may actually not results of program faults, instead,
they are used for input validation. And yet, when asked about `FormatException`,
one developer stated:

“[… ] it tells me that the user explicitly or implicitly (as far as I remember it
is always done explicitly) was loading a distribution package. The package has
it version number defined as part of the root folder name. The version part of
the folder name could not be parsed to a .NET Version object.”

In contrast, the developers view exceptions like `RobApiException` and their
corresponding stack traces are more useful because these exceptions are about
the movement and the control of the industrial robot, and perceive them as
the results of actual program faults as discussed above.

5 Related Work

Although topic models have often been applied to software engineering data [7]
[10][26][32], hierarchical topic models, and, in particular, Bayesian non-parametric
hierarchical topic models have yet to be explored. We focus our related work
discussion on the set of prior work that exists, separately, for both of the data
types used in this paper, i.e. both for mining and understanding application

crush reports as well as interaction data.

As interaction data is large-scale, consisting of several messages per minute
of user interaction with the application, a common goal is to extract high-level
behaviors from the data that express common behavioral patterns exhibited
by a significant cluster of users. Numerous approaches have been suggested to
extract such behaviors from IDE data, using hidden Markov models, sequential

---

See the Stack Overflow discussion “Is it a good or bad idea throwing Exceptions
when validating data?” at [https://stackoverflow.com/questions/1504302/is-it-a-good-or-bad-idea-throwing-exceptions-when-validating-data](https://stackoverflow.com/questions/1504302/is-it-a-good-or-bad-idea-throwing-exceptions-when-validating-data) and many
other discussions on the subject.
patterns, Petri nets, and others [1,9,23], with the purpose of extracting high-
level common behaviors exhibited by developers in the field. Our prior work 
explores the use of the Latent Dirichlet Allocation topic modeling technique, 
more specifically its temporal variant, for the prediction and recommendation 
of IDE commands for a specific developer [10]. 

Mining software crash reports have been a popular area of study in recent 
years, with the ubiquity of systems that collect these reports and the availabil-
ity of public datasets. Here we highlight only the most relevant studies, which 
focus on mining exceptions and stack traces in a corpus of crash reports. 

Han et al. built wait graphs from stack traces and other messages to diag-
nose performance bugs [18]. Dang et al. clustered crash reports based on call 
stack similarity [12], while Wu et al. located bugs by expanding crash stack 
with functions in static call graphs from crash reports that contains stack 
traces [37]. Davie et al. researched whether a new bug in the same source code 
as known bug can be found via bug report similarity measures [13]. 

Crash reports that contains stack traces can be too numerous for engineers 
to manage. Dhaliwal et al. investigated how to group crash reports based on 
bugs [14]. Kaushik and Tahvildari applied information retrieval methods or 
models to detect duplicate bug reports. They compared multiple information 
retrieval methods and models including both word-based models and topic-
based models [20]. Williams and Hollingsworth used source code change history 
of a software project to drive and help to refine the search for bugs [36].

Since bug reports are duplicative and prior knowledge may be used to fix 
new bugs, crash reports can help reuse debugging knowledge. Gu et al. created 
a system to query similar bugs from a bug reports database [17]. 

Different from prior work, in this paper, our aim is to produce a contextual 
understanding of stack traces, and their relationship with user interactions. 
This is based on a large set of interaction traces with embedded stack traces, 
where a stack trace can be considered as a special message in the interaction 
traces. While in this paper we always assume a dataset with already combined 
interaction and stack traces, they need not be a priori, as long as relatively 
reliable timestamps exist in both data sources. The proposed approach is also 
resilient to minor clock synchronization issues that may arise if combining 
stack traces and interaction traces that are collected on disparate machines, 
since it does not require perfect message ordering.

6 Conclusions

Large quantities of software interaction traces are gathered from complex soft-
ware daily. It is advantageous to leverage such data to improve software qual-
ity by discovering faults, performance bottlenecks, or inefficient user interface 
design. We posit that high-level comprehension of these datasets, via unsup-
ervised approaches to dimension reduction, is useful to improving a myriad 
of software engineering activities. In this paper, we aim at modeling a large 
set of user interaction data combined with software crash reports. We leverage
a combined dataset collected from ABB RobotStudio a software application with many thousands of active users. The described approach is novel in attempting to model the combination of the two datasets.

As a modeling technique, hierarchical models, such as, the Nested Hierarchical Dirichlet Process (NHDP) Bayesian non-parametric topic model enable human interpretation of complex datasets. The model allows us to extract topics, i.e., probability distributions of interactions and crashes, from the document collections and assemble these topics into tree-like structure. The hierarchical structure of the model allows browsing from a more generic topic to a more specific topic. The tree also reveals certain structure among users’ interaction with the software. Most importantly, the structure also demonstrates an understanding how an exception co-occur with other messages, and thus provide a context on these messages. We surveyed ABB RobotStudio developers who consistently found parts of the model very useful, although significant more work is required to understand and predict the parts of the model that yielded no insight to the developers. The future work also includes investigating semi-supervised learning models that can leverage developer feedback in formulating an interpretable and useful model.

Acknowledgements The authors would like to thank the RobotStudio team at ABB Inc for providing the interaction dataset and responding to the survey. The authors are also grateful to the anonymous reviewers’ constructive comments.

References

1. van der Aalst, W., Weijters, T., Maruster, L.: Workflow mining: discovering process models from event logs. IEEE Transactions on Knowledge and Data Engineering 16(9), 1128–1142 (2004). DOI 10.1109/TKDE.2004.47
2. Agrawal, A., Fu, W., Menzies, T.: What is wrong with topic modeling? and how to fix it using search-based software engineering. Information and Software Technology 98, 74–88 (2018)
3. Arnold, D.C., Ahn, D.H., De Supinski, B.R., Lee, G.L., Miller, B.P., Schulz, M.: Stack trace analysis for large scale debugging. In: 2007 IEEE International Parallel and Distributed Processing Symposium, p. 64. IEEE (2007)
4. Blei, D.M., Griffiths, T.L., Jordan, M.I.: The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. J. ACM 57(2), 7:1–7:30 (2010). DOI 10.1145/1667053.1667056. URL http://doi.acm.org/10.1145/1667053.1667056
5. Blei, D.M., Moreno, P.J.: Topic segmentation with an aspect hidden markov model. In: Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '01, pp. 343–348. ACM, New York, NY, USA (2001). DOI 10.1145/383952.384021. URL http://doi.acm.org/10.1145/383952.384021
6. Cao, L., Fei-Fei, L.: Spatially coherent latent topic model for concurrent segmentation and classification of objects and scenes. In: 2007 IEEE 11th International Conference on Computer Vision, pp. 1–8 (2007). DOI 10.1109/ICCV.2007.4409865
7. Chen, T.H., Thomas, S.W., Hassan, A.E.: A survey on the use of topic models when mining software repositories. Empirical Software Engineering 21(5), 1843–1919 (2016). DOI 10.1007/s10664-015-9402-8. URL http://dx.doi.org/10.1007/s10664-015-9402-8
8. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. SIGOPS Oper. Syst. Rev. 35(5), 73–88 (2001). DOI 10.1145/502059.502042. URL http://doi.acm.org/10.1145/502059.502042
9. Damevski, K., Chen, H., Shepherd, D., Pollock, L.: Interactive exploration of developer interaction traces using a hidden markov model. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, pp. 126–136. ACM, New York, NY, USA (2016). DOI 10.1145/2901739.2901741. URL http://doi.acm.org/10.1145/2901739.2901741

10. Damevski, K., Chen, H., Shepherd, D.C., Kraft, N.A., Pollock, L.: Predicting future developer behavior in the IDE using topic models. IEEE Transactions on Software Engineering 44(11), 1100–1111 (2018). DOI 10.1109/TSE.2017.2748134

11. Damevski, K., Shepherd, D.C., Schneider, J., Pollock, L.: Mining sequences of developer interactions in visual studio for usage smells. IEEE Transactions on Software Engineering 43(4), 359–371 (2017). DOI 10.1109/TSE.2016.2592905

12. Dang, Y., Wu, R., Zhang, H., Zhang, D., Nobels, P.: Rebuket: A method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 1084–1093. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337354

13. Davies, S., Roper, M., Wood, M.: Using bug report similarity to enhance bug localisation. In: 2012 19th Working Conference on Reverse Engineering, pp. 125–134 (2012). DOI 10.1109/WCRE.2012.22

14. Dhalwali, T., Khomh, F., Zou, Y.: Classifying field crash reports for fixing bugs: A case study of mozilla firefox. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 333–342 (2011). DOI 10.1109/ICSM.2011.6080800

15. Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B.: Bayesian data analysis, vol. 2, 3 edn. Chapman & Hall/CRC Boca Raton, FL, USA (2014)

16. Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., Hunt, G.: Debugging in the (very) large: ten years of implementation and experience. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 103–116. ACM (2009)

17. Gu, Z., Barr, E.T., Schleck, D., Su, Z.: Reusing debugging knowledge via trace-based bug search. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, pp. 927–942. ACM, New York, NY, USA (2012). DOI 10.1145/2384616.2384684. URL http://doi.acm.org/10.1145/2384616.2384684

18. Han, S., Dang, Y., Ge, S., Zhang, D., Xie, T.: Performance debugging in the large via mining millions of stack traces. In: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, pp. 145–155. IEEE Press, Piscataway, NJ, USA (2012). URL http://dl.acm.org/citation.cfm?id=2337223.2337241

19. Hindle, A., Barr, E.T., Su, Z., Gabel, M.: De booktitle=2012 34th International Conference on Software Engineering (ICSE) pages=837–847, y.o.: On the naturalness of software

20. Kaushik, N., Tahvildari, L.: A comparative study of the performance of ir models on duplicate bug detection. In: 2012 16th European Conference on Software Maintenance and Reengineering, pp. 159–168 (2012). DOI 10.1109/CSMR.2012.78

21. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now?: An empirical study of bug characteristics in modern open source software. In: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASP '06, pp. 25–33. ACM, New York, NY, USA (2006). DOI 10.1145/1181309.1181314. URL http://doi.acm.org/10.1145/1181309.1181314

22. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. SIGOPS Oper. Syst. Rev. 42(2), 329–339 (2008). DOI 10.1145/1353535.1346323. URL http://doi.acm.org/10.1145/1353535.1346323

23. Murphy-Hill, E., Jiresal, R., Murphy, G.C.: Improving software developers' fluency by recommending development environment commands. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pp. 421–4211. ACM, New York, NY, USA (2012). DOI 10.1145/2393596.2393645. URL http://doi.acm.org/10.1145/2393596.2393645
24. Nguyen, V.A., Boyd-Graber, J.L., Resnik, P., Chang, J.: Learning a concept hierarchy from multi-labeled documents. In: Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, K.Q. Weinberger (eds.) Advances in Neural Information Processing Systems 27, pp. 3671–3679. Curran Associates, Inc. (2014). URL http://papers.nips.cc/paper/5303-learning-a-concept-hierarchy-from-multi-labeled-documents.pdf

25. Paisley, J., Wang, C., Blei, D.M., Jordan, M.I.: Nested hierarchical dirichlet processes. IEEE Transactions on Pattern Analysis and Machine Intelligence 37(2), 256–270 (2015). DOI 10.1109/TPAMI.2014.2318728

26. Panichella, A., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., De Lucia, A.: How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13, pp. 522–531. IEEE Press, Piscataway, NJ, USA (2013). URL http://dl.acm.org/citation.cfm?id=2486788.2486857

27. Pritchard, J.K., Stephens, M., Donnelly, P.: Inference of population structure using multilocus genotype data 155(2), 945–959 (2000)

28. Roos, M., Martins, T.G., Held, L., Rue, H., et al.: Sensitivity analysis for bayesian hierarchical models. Bayesian Analysis 10(2), 321–349 (2015)

29. Rosen-Zvi, M., Griffiths, T., Steyvers, M., Smyth, P.: The author-topic model for authors and documents. In: Proceedings of the 20th Conference on Uncertainty in Artificial Intelligence, UAI ’04, pp. 487–494. AUAI Press, Arlington, Virginia, United States (2004). URL http://dl.acm.org/citation.cfm?id=1036843.1036902

30. Snipes, W., Nair, A.R., Murphy-Hill, E.: Experiences gamifying developer adoption of practices and tools. In: Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014, pp. 105–114. ACM, New York, NY, USA (2014). DOI 10.1145/2591062.2591171. URL http://doi.acm.org/10.1145/2591062.2591171

31. Soh, Z., Drioul, T., Rappe, P.A., Khomh, F., Gueheneuc, Y.G., Habra, N.: Noises in interaction traces data and their impact on previous research studies. In: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–10 (2015). DOI 10.1109/ESEM.2015.7321209

32. Sun, X., Liu, X., Li, B., Duan, Y., Yang, H., Hu, J.: Exploring topic models in software engineering data analysis: A survey. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’12, pp. 357–362 (2012). DOI 10.1145/2339530.2339552

33. Teh, Y.W., Jordan, M.I., Beal, M.J., Blei, D.M.: Hierarchical dirichlet processes. Journal of the American Statistical Association 101(476), 1566–1581 (2006). DOI 10.1198/016214506000000302. URL http://dx.doi.org/10.1198/016214506000000302

34. Wallach, H.M., Murray, I., Salakhutdinov, R., Mimno, D.: Evaluation methods for topic models. In: Proceedings of the 26th Annual International Conference on Machine Learning, ICML ’09, pp. 1105–1112. ACM, New York, NY, USA (2009). DOI 10.1145/1553374.1553515. URL http://doi.acm.org/10.1145/1553374.1553515

35. Wang, Y., Agichtein, E., Benzi, M.: TM-LDA: Efficient online modeling of latent topic transitions in social media. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’12, pp. 123–131. ACM, New York, NY, USA (2012). DOI 10.1145/2339530.2339552. URL http://doi.acm.org/10.1145/2339530.2339552

36. Williams, C.C., Hollingsworth, J.K.: Automatic mining of source code repositories to improve bug finding techniques. IEEE Transactions on Software Engineering 31(6), 466–480 (2005). DOI 10.1109/TSE.2005.63

37. Wu, R., Zhang, H., Cheung, S.C., Kim, S.: Crashlocator: Locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 204–214. ACM, New York, NY, USA (2014). DOI 10.1145/2610384.2610386. URL http://doi.acm.org/10.1145/2610384.2610386

38. Yin, Z., Caesar, M., Zhou, Y.: Towards understanding bugs in open source router software. SIGCOMM Comput. Commun. Rev. 40(3), 34–40 (2010). DOI 10.1145/1823844.1823849. URL http://doi.acm.org/10.1145/1823844.1823849