Modeling the Evolution of Development Topics using Dynamic Topic Models

Jiajun Hu*, Xiaobing Sun*‡, David Lo†, Bin Li*‡
*School of Information Engineering, Yangzhou University, Yangzhou, China
†School of Information Systems, Singapore Management University, Singapore
‡State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
jiajunhu.yzu.edu@gmail.com, xbsun@yzu.edu.cn, davidlo@smu.edu.sg, lb@yzu.edu.cn

***Abstract*—As the development of a software project progresses, its complexity grows accordingly, making it difficult to understand and maintain. During software maintenance and evolution, software developers and stakeholders constantly shift their focus between different tasks and topics. They need to investigate into software repositories (e.g., revision control systems) to know what tasks have recently been worked on and how much effort has been devoted to them. For example, if an important new feature request is received, an amount of work that developers perform on ought to be relevant to the addition of the incoming feature. If this does not happen, project managers might wonder what kind of work developers are currently working on.**

Several topic analysis tools based on Latent Dirichlet Allocation (LDA) have been proposed to analyze information stored in software repositories to model software evolution, thus helping software stakeholders to be aware of the focus of development efforts at various time during software evolution. Previous LDA-based topic analysis tools can capture either changes on the strengths of various development topics over time (i.e., strength evolution) or changes in the content of existing topics over time (i.e., content evolution). Unfortunately, none of the existing techniques can capture both strength and content evolution. In this paper, we use Dynamic Topic Models (DTM) to analyze commit messages within a project's lifetime to capture both strength and content evolution simultaneously. We evaluate our approach by conducting a case study on commit messages of two well-known open source software systems, *jEdit* and *PostgreSQL*. The results show that our approach could capture not only how the strengths of various development topics change over time, but also how the content of each topic (i.e., words that form the topic) changes over time. Compared with existing topic analysis approaches, our approach can provide a more complete and valuable view of software evolution to help developers better understand the evolution of their projects.

I. Introduction

Mining unstructured software repositories (e.g., bug reports, mailing lists, commit messages, etc.) has emerged as a research direction over the past decade, which has achieved substantial success in both research and practice to support software maintenance [1]–[3]. These studies have shown that interesting and practical results can be obtained from mining these software repositories, thus allowing maintainers or managers to better understand how software evolves.

Unlike structured contents in software repositories (e.g., source code, execution traces, change logs, etc.), unstructured contents are often harder to analyze because the data is often vague and noisy [4], making it time-consuming for project stakeholders to manually analyze software repositories. One of recent advanced techniques is to use topic analysis tools (topic models), such as Latent Dirichlet Allocation (LDA) [5], to automatically extract topics from textual repositories to explore and organize the underlying structure of software documents [6]–[16]. Topic models can be used to discover a set of ideas or themes (aka., topics) that well describe the entire corpus. Topics are collections of words that co-occur frequently in the entire corpus and usually have a close semantic relationship. More specifically, a topic model can represent a set of documents as a set of topics, where each document contains one or more of these topics, and each topic is composed of a set of words that appear in the repository.

Understanding how development topics evolve, i.e., change, in a software repository over time can help project stakeholders to understand and monitor activities performed in their project at various time points during project's lifetime. For example, project managers can understand what tasks have recently been worked on and how much effort has been devoted to each task by retrieving revision control systems [10], while developers can understand the evolution of certain features of source code by mining source code repository [6], [9].

To help developers understand software evolution, a number of LDA-based approaches have been proposed. Thomas et al. applied *the Hall model* [17] to analyze the entire history of source code documents to recover information on how the strengths (i.e., popularity) of various topics change over time [6], [9]. They ran *LDA* once on all versions of a software project to get a set of topics, and then computed several metrics to represent the strength of a topic for each of the version. In such a way, their approach can capture the *strength evolution* of the development topics. However, the content of a topic (i.e., the set of words that form a topic), never changes across the versions. On the other hand, Hindle et al. applied *the Link model* [18] which runs LDA for each time window separately and then used a post-processing phase to link topics which are similar enough across successive time windows [10]. Their approach can capture changes in the content of each topic over time (i.e., *content evolution*). Unfortunately, it cannot recover the strength of a topic across all time windows – for some time windows, some topics do not exist and are expressed as combinations of other topics. Thus, none of existing approaches can capture both strength and content evolution.

As discussed above, existing work on understanding software evolution focused on either *topic strength evolution* or *topic content evolution*. But both pieces of information are necessary for developers to fully understand how software evolves. For example, project managers may hope to know how much work was related to *feature A* at a certain time period, which can be obtained by computing the corresponding topic strength at that time. They might also want to know what kind of work was done for *feature A* at a particular time point. Topic strength will not help project managers with this information. Rather, the content of the corresponding topic can shed light on activities that are performed for *feature A* at that time point.

In this paper, we propose a novel approach to capture both topic strength and content evolution simultaneously. We use Dynamic Topic Models (DTM) [19] on the commit messages of a software repository. Then, we capture *topic strength evolution* by computing the *Normalized-Assignment* metric at each time window to represent the strength of a topic for that time. We capture *topic content evolution* by extracting the top 10 words that characterize a topic for each time window.

We conduct a case study on the commit messages of two well-known open source software systems, *jEdit*[1] and *PostgreSQL*[2]. Moreover, we also empirically compare our approach with the state-of-the-art approaches (the link model and hall model). The results show that our approach can capture the evolution of both strength and contents of various development topics over time which corresponds to meaningful description of the whole development iteration. In addition, compared with existing topic analysis approaches, our approach can provide a more complete view of software evolution.

The rest of the paper is organized as follows. Section 2 provides background of unstructured contents in software repositories and several existing topic analysis approaches. Sections 3 presents our approach. Section 4 and Section 5 present our case study setup and the results, respectively. Section 6 enumerates the threats to the validity in our case study. Finally, we present the conclusion and future work in Section 7.

II. Background

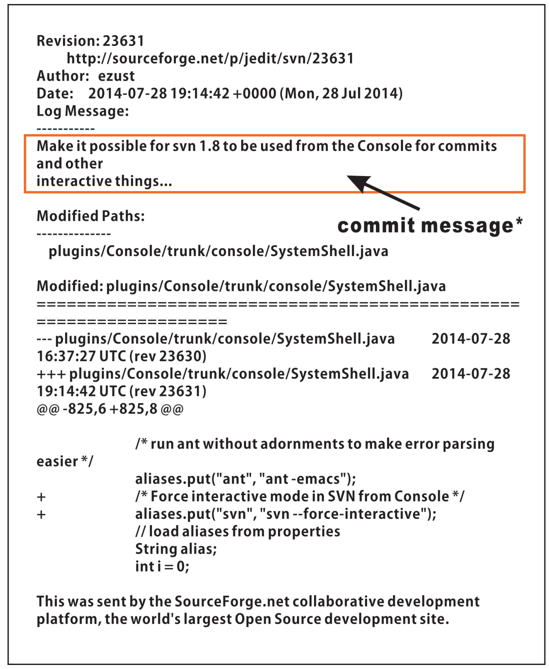
In this section, we introduce unstructured software repositories and several topic evolution models.

A. Unstructured Software Repositories

Unstructured software repositories, such as the comments and identifier names in source code, mailing lists, bug database and commit messages, often contain a variety of information about different facets of software development. For example, the nature language text embedded in source code (e.g., comments, identifier names and string literals) [20] often represents an important source of domain information and can help determine the high-level functionality of the source code. Due to its rich (statistical results showed that between 80% and 85% of the data in software repositories is unstructured [21], [22]) and useful information, mining unstructured software repositories can uncover interesting and actionable information (e.g., co-change coupling) about software systems [1], [2]. In these unstructured repositories, the revision control systems maintain and record the history of changes in their repositories. Developers typically use revision control systems to record various changes that occur on their project, such as bug fixing on source code and updates of system's documentation [23]. Most revision control systems (including CVS [24], Subversion (SVN) [25], and Git [26]) allow developers to enter a commit message (example commit see Figure 1) when they commit a change into the repository, describing the change at a high level, i.e., *Who changed what and when*. These unstructured commit messages are of grate value because they record the history of changes during the iterative development of a software project, thus describing how the project is evolving over time.

[1]http://www.jEdit.org/

[2]http://www.postgresql.org/



```
Revision: 23631
      http://sourceforge.net/p/jedit/svn/23631
Author:  ezust
Date:    2014-07-28 19:14:42 +0000 (Mon, 28 Jul 2014)
Log Message:
-----------
Make it possible for svn 1.8 to be used from the Console for commits
and other
interactive things...

Modified Paths:
--------------
   plugins/Console/trunk/console/SystemShell.java

Modified: plugins/Console/trunk/console/SystemShell.java
===================================================================
--- plugins/Console/trunk/console/SystemShell.java       2014-07-28
16:37:27 UTC (rev 23630)
+++ plugins/Console/trunk/console/SystemShell.java       2014-07-28
19:14:42 UTC (rev 23631)
@@ -825,6 +825,8 @@

                /* run ant without adornments to make error parsing
easier */
                aliases.put("ant", "ant -emacs");
+               /* Force interactive mode in SVN from Console */
+               aliases.put("svn", "svn --force-interactive");
                // load aliases from properties
                String alias;
                int i = 0;

This was sent by the SourceForge.net collaborative development
platform, the world's largest Open Source development site.
```


Fig. 1. An example of commit from jEdit. The text within the box is a commit message.

B. Topic Evolution Models

Topic evolution model is a topic model that accounts for time in some way, allowing documents to have timestamps. It is useful to detect and analyze how topics change and evolve over time. Given a topic, the strength of which may experience spikes or drops several times during the corpus's lifetime, indicating its dynamic contribution to the underlying corpus. What's more, the content of a topic may also experience changes with time forwarding, indicating the evolution of different aspects within a topic. So far, several popular topic evolution models have been introduced.

1) The Link Model: The Link Model, proposed by Mei et al. [18], was first applied to software repositories (i.e., commit messages) by Hindle et al. [10]. They used *LDA* for each time of the corpus separately to model the evolution of development topics. Since the topic in time T may be the same as the topic in time $T+1$, a post-processing phase is required to link topics which are similar enough (for example, they share 8 out of 10 words) across successive time interval.

2) The Hall Model: The Hall Model, proposed by Hall et al. [17], was first applied to software repositories (i.e., source code) by Linstead et al. [27] and validated by Thomas et al. to describe source code changes [6]. *The Hall Model* applied *LDA* to the whole versions of the software corpus at a time. A post-processing phase is also required to separate corpus at different versions and several topic metrics (such as weight and assignment [13]) are computed in a particular version to represent the contribution of that topic in this version.

3) Dynamic Topic Models: Dynamic Topic Models (DTM) [19], proposed by Blei et al., is the model we study in this paper. *DTM* models the evolution of a topic as a discrete Markov process with normally distributed changes between time periods, which allows only gradual changes over time. It captures the evolution of topics in a time-sequentially organized corpus of documents and produces a doc-topic matrix and topic-term matrix at each time. Doc-topic matrix represents that each document is a multi-membership mixture of topics, based on which we can compute topic metrics at each time and link them on the timestamps to represent *topic strength evolution*. Topic-term matrix represents that each topic is a multi-membership mixture of terms, based on which we can represent *topic content evolution* by inspecting the trends of word usage and its frequency within a topic.

In the next section, we will illustrate the usage of *DTM* on commit messages in detail.

III. Approach

Figure 2 depicts the general process of our approach. We first extract commit messages from a project's revision control system. Second, we apply several data preprocessing operations on the whole corpus to filter out noisy data. Third, we process each commit messages into word distributions and the time windows into time sequences. These word distributions and time sequences are then used as input for *DTM* to produce doc-topic matrix and topic-term matrix at each time window. Finally, we post-process the extracted topics by computing two topic metrics and mapping the results on timestamps to visualize the strength evolution and content evolution.

A. Data Preprocessing

There does exist some noise in the unstructured commit messages, which will confuse and distract the topic analysis tools. So the commit messages need to be preprocessed. Natural language preprocessing (NLP) techniques are usually used to perform one or more preprocessing operations before applying topic models to reduce noise and improve the quality of the resulting text [28].

We preprocess the commit messages by applying typical natural language preprocessing operations [28]. We first split the original commit messages into tokens and remove unrelated and unimportant words, such as the punctuation, numeric characters and others (e.g., @, *, !). We also remove common English language stop words (e.g., *the*, *it*, *in*) to reduce noise. Then, we stem each word to its original format (e.g., "replaced" becomes "replac") to reduce vocabulary size. Finally, we prune the vocabulary by removing overly rare words (those that occur in less than ω times, which is a threshold relying on the size and type of the corpus) because these words are of little use for topic analysis.

TABLE I
The notations used in our study

Notation	Description
$z_1, z_2, \cdots, z_k$	k topics
$w_1, w_2, \cdots, w_n$	n unique words
$T_1, T_2, \cdots, T_t$	t time intervals
d_{ij}	The message with index j in T_i
$\|T_i\|$	The total number of messages in T_i
$\theta_{d_{ij}}$	The topic distribution of message with index j in T_i
z_{ik}	The topic with index k in T_i
$\phi_{z_{ik}}$	The word distribution of topic with index k in T_i

B. Applying Dynamic Topic Models

After the preprocessing step, we first analyze each commit message's word counts (i.e., word distributions). For each commit message, we count the occurrence of each word and their total number in the message, thus producing the word distributions for that message. Second, we group the messages into time windows based on their commit dates and count the total number of time windows and the number of messages in each window, which we call time sequences for the whole corpus.

With the word distributions for each message and time sequences for the whole corpus, we use *DTM* to produce the doc-topic matrix and topic-term matrix at each time window.

C. Post-Processing and Visualization

After applying the *DTM* model, we post-process the doc-topic matrix and topic-term matrix by computing several topic metrics to represent topic strength evolution and topic content evolution. To compute these topic metrics, there are some notations as shown in Table I. First, we measure how the topic strength changes over time by computing a **Normalized-Assignment** (*NA*) metric at each time point. The *NA* of a topic is the average value of the topic memberships of all documents in that topic at a time, which indicates the total presence of the topic throughout the messages in that time. A higher *NA* means that a large portion of the messages is relevant to that topic. We define the Normalized-Assignment of topic z_k at time T_i as

$$NA(Z_k, T_i) = \frac{\sum_{j=1}^{|\mathrm{T_i}|} \theta_{d_{ij}}[k]}{|T_i|} \qquad (1)$$

The **strength evolution** (*SE*) of a topic z_k is a time-indexed vector of *NA* values for that topic:

$$SE(z_k) = [NA(z_k, T_1), \cdots, NA(z_k, T_t)]. \qquad (2)$$

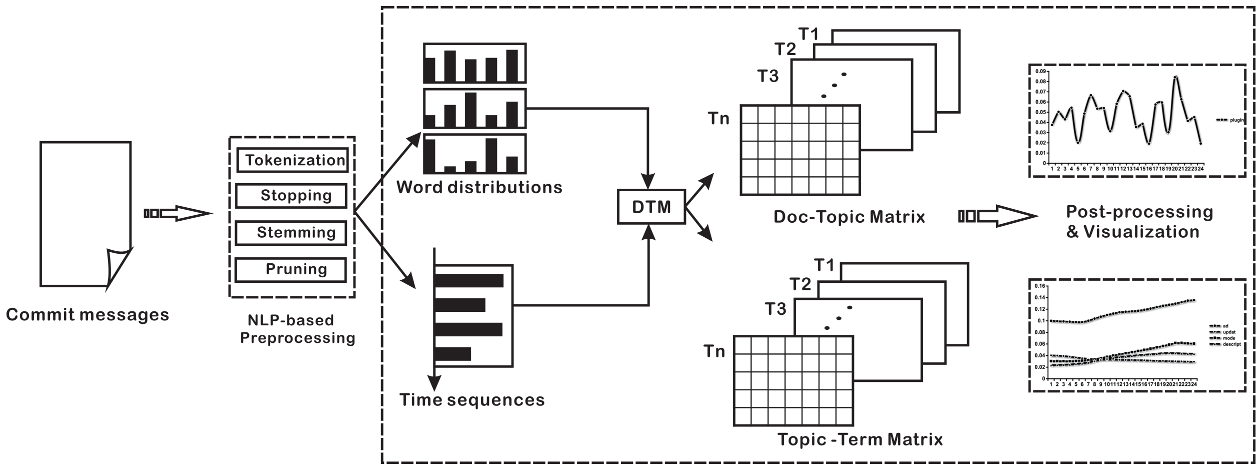


Fig. 2. The general steps for our approach

Then, for the topic content which includes the words and their distributions, it contains two parts: the word and its frequency within a topic. We choose the top 10 most frequent words (i.e., terms) to illustrate a topic and measure how the topic content changes over time by computing **Term-Frequency** (*TF*) at each time point. The *TF* of a term is the probability of the term memberships of a topic at a given time, indicating the contribution of different aspects within this topic. A higher *TF* means that this topic has a strong relationship with that term. We define the Term-Frequency of a word w_n in topic z_k at time T_i as:

$$TF(w_n, z_k, T_i) = \phi_{z_{ik}}[n] \tag{3}$$

The **content evolution** (*CE*) of a topic z_k is a time-indexed vector of *TF* values for terms used in that topic.

$$CE(z_k) = \begin{cases} [TF(w_1, z_k, T_1), & \cdots, TF(w_1, z_k, T_t)] \\ [TF(w_2, z_k, T_1), & \cdots, TF(w_2, z_k, T_t)] \\ & \vdots \\ [TF(w_n, z_k, T_1), & \cdots, TF(w_n, z_k, T_t)] \end{cases} \tag{4}$$

We notice that not all the words in the vocabulary should be taken into account. Normally, we only choose the top 10 most frequent words to illustrate content evolution.

Finally, to better understand the topic evolutions, we map *NA* and *TF* values to timestamps and use line chart to visualize the results.

Example results are illustrated in Figure 3 and Figure 4. They show several top words from those topics in each time period based on the *TF* value of each word in that topic. From the results, we see the topic strength evolution by mapping *NA* value of topics to timestamps as well as the topic content evolution by mapping *TF* values of several top words to timestamps.

IV. Case Study

A. Studied Systems

To validate our technique, we perform in-depth case studies on the commit messages of two well-known open source software systems, *jEdit* and *PostgreSQL*. *jEdit* is a medium-sized, mature open source text editor written in Java programming language. It provides rich features for developers, including built-in macro language, extensible plugin architecture and syntax highlighting. *PostgreSQL* is a powerful, open source object-relational database system written in C programming language. It has more than 15 years of active development and a proven architecture that has earned it a strong reputation for reliability, data integrity, and correctness. Both of them are well organized, have extensive documentation which can be extracted from sourceForge[3] and Git[4]. They have been widely used as case studies in the context of software maintenance and evolution [9], [10]. In our case study, we extracted commit messages from Jan. 2006 to Dec. 2010 in *jEdit* cvs-repositories. We also extracted commit messages from Jan. 2002 to Dec. 2008 in *PostgreSQL* git-repositories. For more details please refer to Table II.

TABLE II
Characteristics of the Subject Systems

Characteristics	jEdit	PostgreSQL
Purpose	Text editor	Relational database
Implementation language	Java	C
Time period considered	2006.1-2010.12	2002.1-2008.12
Number of commit messages	12116	15990
Number of words before Pre-processing	152149	456839
Number of words after Pre-processing	74604	196096
Number of unique-words	889	1763

B. Study Setup

We preprocessed the commit messages of each system using the steps described in Section III-A. We set ω as 9 and 10 for *jEdit* and *PostgreSQL* respectively to prune the vocabulary because we found these settings could produce more meaningful topics according to our experimental results. We set the number of topics as 20 for both the systems because existing experimentation showed that fewer topics

[3] http://sourceforge.net/p/jEdit/mailman/jEdit-cvs/

[4] http://git.postgresql.org/gitweb/?p=postgresql.git;a=log

might aggregate multiple unique topics while more topics seemed to duplicate results and create indistinct topics [10]. We choose the time interval as one month because it is smaller than the time between minor releases but large enough for enough number of commits to be analyzed. Then, we applied *DTM*[5] to produce doc-topic matrix and topic-term matrix at each time period. Finally, we compared the results produced by *DTM* with the results produced by *the Hall Model* and *Link Model* to figure out whether the combination of the strength and content evolution can describe more complete and comprehensive view of software evolution. Since *the Link Model* and *Hall Model* are all based on *LDA*, we used *MALLET*[6] to carry out topic analysis. For *the Link Model*, we linked topics that shared 5 out of 10 words between successive time intervals because we found higher threshold could produce tiny or even zero trends.

C. A Preliminary Experiment

In our first exploratory pass, we performed a preliminary case study to know what kind of information is contained in the large amount of commits and whether *DTM* can capture them as much as possible. Since information contained in revision control systems has been frequently used to support software evolution research, many taxonomical studies have been performed on commits to extract the general focus of them. Swanson [29] proposed a classification of maintenance activities as corrective, adaptive and perfective. Hindle et al. [23] created Extended Swanson Categories of Changes, which are the taxonomical standard we refer to in this paper and are shown in Table III. Based on these categories of changes, we performed a preliminary case study to see whether *DTM* can capture these notable types of changes (but not limited to them) from a real system.

We applied our approach to the repository of *jEdit* from Jan. 2006 to Dec. 2010 to produce results. We investigated into all topics and messages related to the topics in detail to manually decide which categories of changes these topics captured. We show the example topics and the captured information of *jEdit* in Jul. 2007 in Table IV. By investigating into the topics and the top related messages, we found that most types of changes can be captured by *DTM* (about 73%). In addition, the potential information provided by extracted topics is obviously not limited to traditional categories, for example, *topic 2* includes the information related to database, *topic 14* is partly about *regex* and *topic 3 and 15* have a strong relationship with *GUI*.

V. Results

In this section, we applied our approach as well as *the Link Model* and *Hall Model* to the repository of *jEdit* and *PostgreSQL*. We discussed our results in detail and compared the results with these two models. All the empirical data and results are available online[7].

[5]http://code.google.com/p/princeton-statistical-learning/downloads/list
[6]http://mallet.cs.umass.edu/
[7]http://www.risame.net/sun/Experiment%20Data%20and%20Results.zip

A. Results of Our Approach

1) jEdit: We applied our approach to *jEdit* from 2006 to 2010, which includes over 12K commits. We found that most topics' strength evolution fluctuated violently during the entire time period, which means that various development topics are distributed in each time and no one can dominate the development iteration across successive time intervals. We also found that the most common words across multiple topics are *add* and *plugin*, indicating the overall active growth of the project and large collection of plugins, which is one of the most core features of *jEdit*. We examined each topic's strength and content evolution and investigated into the commit messages related to that topic (where $\theta_{d_{ij}}[k] \geqslant 80\%$) in detail to understand the purpose of each topic. We notice that there were several notable topics:

Sort and Search. *Topic 2* has a strong relationship with *sort* and *search*. We found that the most frequent words contained in this topic were *search*, *sort*, *query*, *result* and *tag*. The topic's strength reaches to peak (15%-16%) after May. 2010, and before that time it varies from 2% to 8% and fluctuates violently. We looked at the top matching commits and found that 46% of them come after May. 2010. Then, we turned our attention to different aspects within this topic and found that the *TF* of *sort* seems to decline after Oct. 2006 while *query* seems to rise obviously since 2009. This was the real content evolution trend of this topic because we found that the sentences like "*quick search tag query*", "*DB queries*" and "*query results*" appeared so often during the late period while the sentences like "*sort the list*" and "*quick search sorted*" appeared frequently around Oct. 2006 but rarely after that time.

Module Adding and Removing. In *topic 6*, tokens such as *ad*, *file*, *method*, *mode* and *task* are common during the whole time, which clearly implicate the activities of adding *modules* (e.g., *javacc edit mode*, *jflex edit mode*) or *new features*. The topic's strength varies from 1% to 10% and also experiences violent fluctuation. The *TF* of *method* seems to decline since Sep. 2006 while *file* seems to be stable consistently, indicating that work related to *adding method* is becoming lower and lower but *adding file* is an on-going development job from the beginning to the end. On the other hand, *topic 13* includes common words such as *remov*, *deprec*, *rid*, *redund* and *unus*, which holds the opposite activity of *Module Add*. This topic has a strong relationship with removing unused class, method, icon and api and the topic's strength varies from 2% to 16% and also fluctuates violently (details see Figure 3).

Fixing Bug and Error. We extracted two topics related to bug fix but they represent different facets respectively. *Topic 7* includes common tokens such as *fix*, *error*, *gdb*, *typo*, *issu* and *compil* which tend to be about fixing error (such as compilation error or encoding error) while *topic 16* includes common tokens such as *fix*, *bug*, *npe*, *except*, *wrong*, *null* and *problem* which tend to be about fixing runtime bug or the bug recorded in bug database. Since these two topics are similar to some extent, we put them together and sum up their *NA* values at each time to

TABLE III
EXTENDED SWANSON CATEGORIES OF CHANGES

Categories of Change	Issues addressed	Type and abbreviation
Corrective	Processing failure Performance failure Implementation failure	bug fix (bug) debug (dbg)
Adaptive	Change in data environment Change in processing environment	platform specific (plat), testing (tst) build (bld), documentation (doc), data
Perfective	Processing inefficiency Performance enhancement maintainability	clean up (cln), indentation (ind), maintenance (mntn) refactoring (rfact), module move (mmod), module remove (rmod)
Implementation	New requirements	initialization (init), module add (add), feature add (fea) external (ext), internationalization (ind)
Non functional	Legal Source control system management Code clean-up	legal (lic), rename (ren), token replace (trpl) source control (scs), merge (mrg), versioning (ver)
other	other	cross, branch (brch)

TABLE IV
EXAMPLE TOPICS AND THE CAPTURED TYPES OF CHANGES FROM JEDIT IN JUL. 2007

Index	Topic (illustrated by top 10 words)	Captured types of changes
1	line work comment fold commit import command ad javadoc fix	init bug doc fea
2	properti search ad result string word sort databas queri tabl	fea
3	view textarea set parser sidekick dialog posit locat default jEdit	null
4	ad menu class action add list context descript filter macro	fea doc
5	file code style clean ad path fix directori vf sourc	bug fea data ind
6	mode ad miss edit file thread index method task name	add fea
7	fix error complet compil handl warn better variabl function issu	bug mntn dbg
8	plugin jEdit directori packag instal ad depend org dir repository	add
9	project save move load svn trunk file layout merg branch	brch mrg mmod
10	buffer close replac bufferset editpan except view user current method	null
11	option doc pane tree ad select set node configur color	fea bug mntn doc
12	chang updat document minor small changelog log pre prop list	doc cross
13	remov icon method api deprec call unus rid redund import	rmod
14	java chang html label regex beanshel txt window remov regexp	rmod doc
15	dockabl ad dock button action window featur area show framework	fea
16	fix bug npe pars indent wrong null except small match	bug
17	tag releas patch test initi appli creat explicit ad fold	scs init fea bug tst
18	version updat number depend bump pre jEdit requir oop ad	ver
19	build xml support file ad plugin script jar target php	bld
20	make code diff cleanup check chang log messag pv data	cln doc

produce the total strength evolution of two topics. We found that the total strength varies from 6% to 21% with an average of 12%, which is very high compared with other topics. Thus fixing bug and error is a general topic which makes up a large portion of the development work.

New Directory for Plugins. When looking at the commit messages related to *topic 8*, we found that most of the messages have a fixed format during the early time (before 2007), which is *Directory /cvsroot /jEdit /plugins / . . . added to the repository*, such as *Directory /cvsroot /jEdit /plugins /minitools /src /gatchan /jEdit /minitools /autosave added to the repository* committed by Matthieu Casanova at 20:08:22, 05-28-2006 . The common tokens included in this topic during that time are *plugin*, *jEdit*, *directori*, *repositori* and *ad* and the topic's strength reaches to top (about 17%), indicating that the topic has a strong relationship with creating new directory for a plugin and occupies a large portion of the work before 2007. However, after 2006, the topic's strength becomes lower (2%-8%) and in addition to creating new directory for plugins, other types of commits related to plugin, such as *add plugins* and *install package in plugins*, are also clustered into this topic.

GUI. In *topic 15*, tokens such as *dockabl*, *dock*, *action*, *button*, *panel*, *area* and *window* are common during the whole time, indicating different components related to *GUI*. The topic's strength reaches to peak (about 17%) around Sep. 2008 (see Figure 3) and we found that about 20% of the top matching commits come from this time period. We also found that *dockable window* is a notable *GUI* feature provided by *jEdit* because the word *dockabl* is always the top frequent term within this topic from the beginning to end. This is consistent

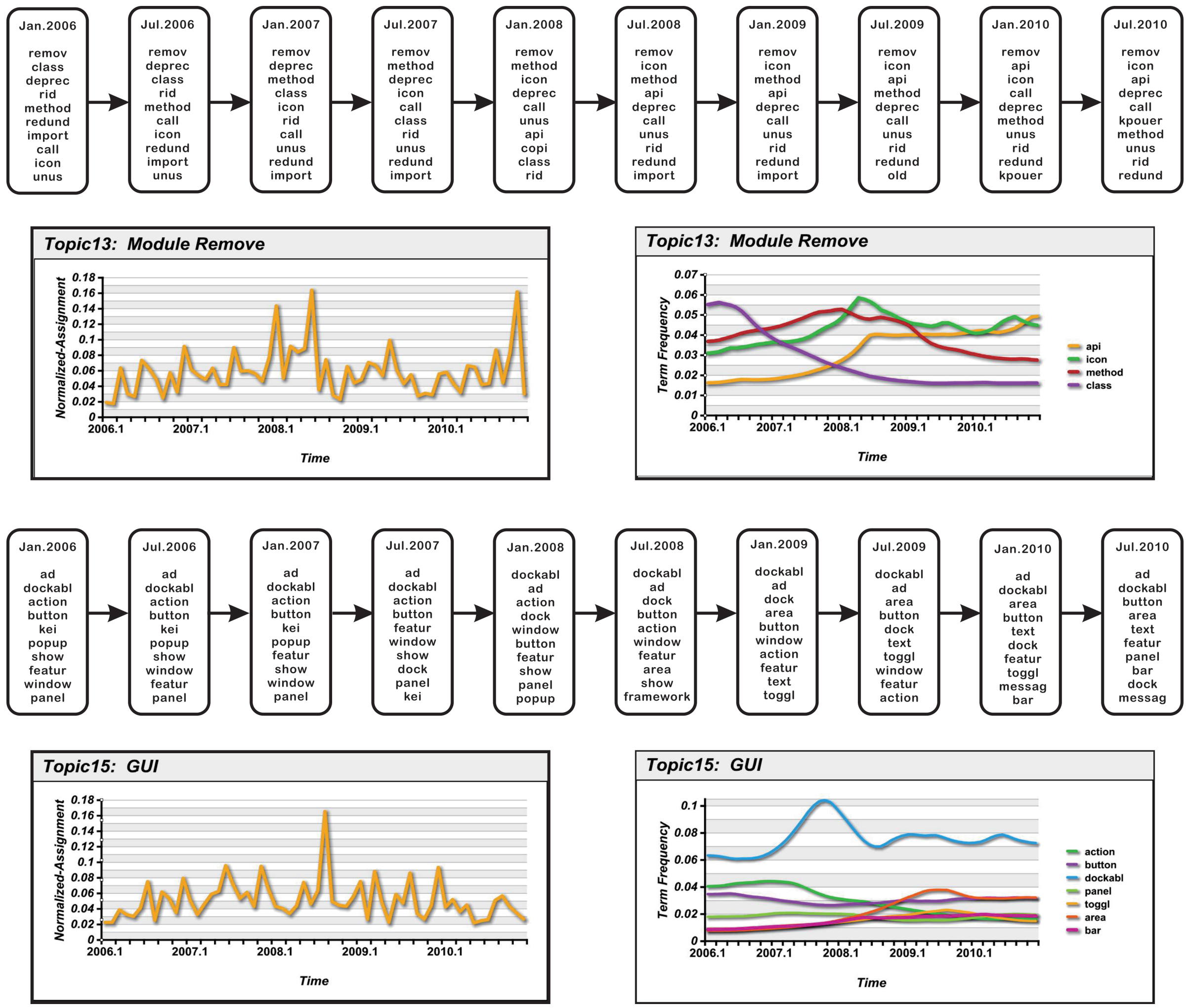


Fig. 3. Example topic evolutions estimated from jEdit. For two topics, we illustrate: (a) the top ten most frequent words at seven month lags (b) the strength evolution by mapping NA values to timestamps (c) the content evolution by mapping TF values of several notable words within the topic to timestamps.

with one of the *jEdit* features – "docked windows".

There are also some other development topics, which include sidekick plugin, encoding and code style, branching or merging the main trunk of the version control system, buffer and bufferset, update or change of the system's documentation, regular expressions, tag release, bumped version numbers, build system files and code cleanup.

2) PostgreSQL: We also applied our approach to *PostgreSQL* from 2002 to 2008, which includes over 15K commits. Like what we have found in *jEdit*, most topics in *PostgreSQL* also experienced violent fluctuation during the whole development time period. But the results are different from that of *jEdit*. Specifically, the topics extracted from *PostgreSQL* reveal more professional knowledge (about database) than *jEdit*. This provides some difficulty for us to understand each topic deeply because we are lack of the relevant knowledge. So in this section, we choose several topics that we feel confident for illustration.

Update. *Topic 4* has a strong relationship with update because we found the common tokens contained in this topic are *updat*, *faq*, *releas*, *note* and *document*. The topic's strength varies from 2% to 24% with an average of 8%, which is high compared with other topics, indicating that the work related to *update release* and *faq* constitutes a large part of the whole development work. From the perspective of content evolution, we found that the *TF* of "releas" and "note" rises steadily from the beginning to the end. So it seems that more *update* work is occupied by *release note* during the late period.

SQL. We extracted two topics related to SQL but they represent different aspects. *Topic 5* tends to implicate the *SQL statement* because common words contained in this topic are *tabl*, *transact*, *kei*, *trigger*, *lock*, *constraint*, *drop* and *alter* (as shown in Figure 4). While *topic 6* tends to implicate a different aspect from *topic 5*, i.e., *SQL function*. Words such as *function*, *sql*, *type*, *return*, *paramet*, *argument* and *call* occur frequently during the whole time period. From our manual analysis in the original corpus, we found much work related to SQL function concentrated on function's arguments and return types, which is consistent with the extracted topic.

Bug Fixing. Like *jEdit*, we also extracted two topics related

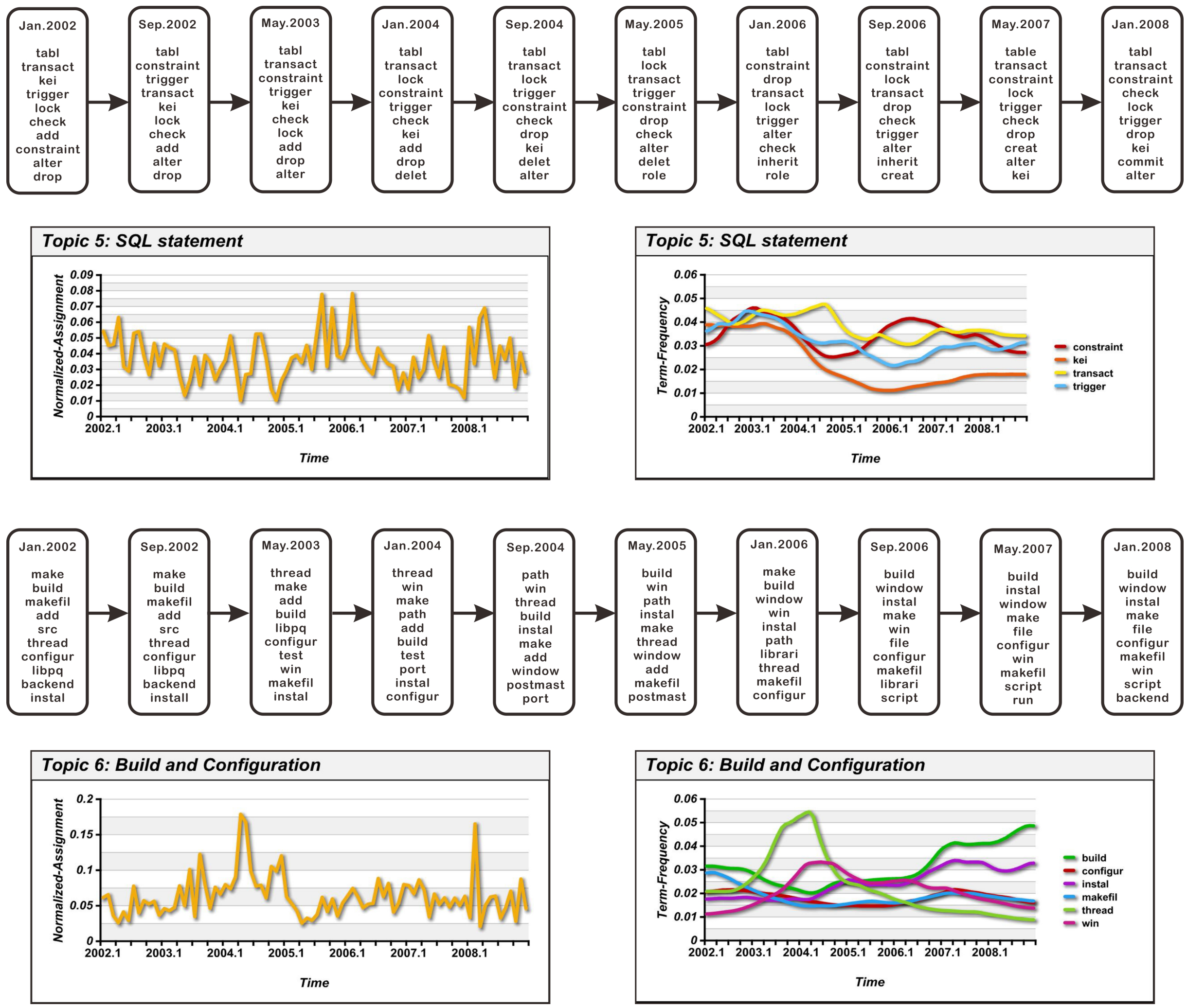


Fig. 4. Example topic evolutions estimated from PostgreSQL. For two topics, we illustrate: (a) the top ten most frequent words at eight month lags (b) the strength evolution by mapping NA values to timestamps (c) the content evolution by mapping TF values of several notable words within the topic to timestamps.

to bug fix from *PostgreSQL*'s repository. *Topic 3* contains common words such as *fix*, *bug*, *patch*, *problem*, *typo* and *broken*, while *topic 14* contains common words such as *error*, *messag*, *report*, *fix*, *elog*, *failur* and *log*. We investigated the top matching messages related to these two topics respectively to identify the differences between them. We found that *topic 3* emphasized on fixing a bug and resolving it immediately while *topic 14* emphasized on recording an error message in bug reports when meeting a bug or error, with developers sometimes trying to fix it, sometimes just recording it. Finally, we put these two topics together as we did in *jEdit*. We found the the total strength varies from 7% to 21% with an average of 12%, which is very high compared with other topics. So it seems that the development work related to bug fixing makes up a big part of the whole work in *PostgreSQL*'s development iteration.

Building and Configuration. The focus of *topic 6* is on the building or configuration of system files (such as *Makefiles*) because the common words contained in it are *make*, *build*, *makefil*, *thread*, *configur*, *instal* and *win* (as shown in Figure 4). We found that the topic's strength reached to peak around Mar. 2004, and meanwhile the *TF* of *thread* became obviously high around this time. So it seems that much effort was devoted to this topic and to some degree the topic may be more relevant to *thread* than any other time. To figure out whether we are correct, we investigated into the commits related to this topic around 2004. As a result, we found that *thread compile* is requested during this time as some messages such as *the enabling of threads for the OS* or *letting configure enable threads* occurred frequently.

There are also some other development topics, which include interval style and date format, database connection, documentation, regression test, module add, data types and file.

B. Comparison with the Link Model and Hall Model

In our last exploratory pass, we compared the results produced by *the Link Model*, which provides content evolution, and *the Hall Model*, which provides strength evolution, with the results produced by our approach to figure out whether both the information of strength evolution and content evolution can provide more complete and comprehensive view of software evolution.

Comparison with *the Hall Model*. *The Hall Model* only provides strength evolution for the extracted topics. Instead, *DTM* can provide strength evolution as well as the content evolution, which helps developers understand more detailed information of development iteration. For example, we found that the strength evolution of topic *Module Remove* reached peak around the middle time of 2008 for both the *DTM* and *the Hall Model*. However, from the content evolution provided by *DTM*, we could get additional information, i.e., *remove icon* and *remove method* which make a strong contribution to this topic at that time. The top word of this topic extracted by *the Hall Model* was always the *method* even the part of the work related to *remove method* was declining during the later time. Without the information provided by the content evolution, we may think this topic is dominated by *remove method* during the whole development iteration but it is not the real trend. What's more, we found that the topics extracted by *the Hall Model* could be dominated by the commit messages from a short time period. For example, we extracted a topic formed by words *jedit*, *ad*, *directori*, *repositori*, *pluguin*, *cvsroot*, *src*, *doc*, *api* and *org*. However, by investigating the messages related to this topic, we found that most of them belonged to the time windows before Aug. 2006 and became rare after that time. So it is more suitable to use this topic to describe the messages before that time instead of the whole time. Unfortunately, even the strength of this topic dropped quickly, it is still viewed as a global topic in the hall model and other topics in the later time may be ignored. While in *DTM*, the content of the topic is allowed to change with the evolution of the corpus, which is more useful in practical comprehension of software evolution.

Comparison with *the Link Model*. *The Link Model* produced more detailed and local topics to describe the development activities at different time intervals. However, it seems incomplete because we can only learn what has been done in each time but we cannot get the information about how much effort was devoted to each topic. For example, is the effort devoted to *GUI* much greater than the effort devoted to *remove API*? We cannot answer it if we only rely on the content evolution of these topics. What's more, the linking phase requires the use of a similarity threshold to determine if a topic found in one time is similar to another topic in the next time. In our topic analysis for *jEdit* repository during 2008, we varied the similarity threshold from 3 to 7 out of ten words and the number of trends are 17, 7, 2, 1 and 0 respectively. The selected time interval lasted at most 3 months. Thus it is difficult for us to choose an optimal threshold value to use this model. However, except the number of topics, there is no other redundant parameters for *DTM* to choose.

As discussed above, *topic content evolution* and *topic strength evolution* are complementary, and these two types of information can provide more complete and comprehensive results to describe the evolution of development topics than either one of them.

VI. Threats to Validity

In this section, we discuss several limitations to our study.

Quality of commit messages. Similar to other software repository analysis techniques, our results are dependent on the quality of the commit messages. Because of the poor discipline and conventions, there are a certain amount of commits with empty or irrelevant messages which do not have any sematic values, thus leading to some impacts on our results.

Selected systems. We only conducted our studies on two open-source systems due to their robust designs and extensive documentation. Thus we cannot guarantee that the results obtained in our study can be generalized to other types of systems, such as close-source systems or the systems with worse designs. Additional case studies are needed to investigate the generalization of our approach.

Preprocessing steps. In our study, we performed four preprocessing steps on the commit messages. There is currently no guidance or consensus on which steps are actually necessary or beneficial. What's more, the case happens that the word which should be chosen as stop word may be not chosen in our study, and vice versa. In addition, the choice of ω (the threshold of pruning) is set as fixed in our study. Other settings of stop words and ω may produce different results when using *DTM*.

Parameter values. In our study, we set the number of topics to 20, the time interval to one month, and the topic is represented by top 10 words. Since there is no standard way to determine optimal values, our choice is based on the previous work of Hindle et al. [10].

Interpretation on topics. Topics generated by *DTM* were interpreted and labelled by our manual analysis. Since we are lack of expertise on the studied systems, the interpretation and labelling on the topics may be inaccurate and questionable. The evaluation should ideally be made by impartial developers, which will become part of our future work.

Comparison approach. In the last step of our case study, we only compared the results from the perspective of whether the combination of the content and strength evolution can provide more complete and comprehensive results than either one of them. But we did not compare the results in the perspective of either the content evolution or the strength evolution. The occasion may happen that the strength evolution produced by *the Hall Model* would provide more accurate and sensitive trend than *DTM*, or the content evolution produced by *the Link Model* would provide more detailed topics than *DTM*, which will become our future work.

VII. Conclusion and Future Work

Traditional topic evolution models were designed to produce either the strength evolution or the content evolution of the unstructured software repositories. In some cases, developers may have the request to know not only the evolution of the strength of a topic but also the evolution of different aspects within it. In this paper, we applied the Dynamic Topic Models to the commit messages to represent both their topic strength and content evolution, respectively. Some studies were conducted on two well-known and open source projects, *jEdit* and *PostgreSQL*. Moreover, we compared the results with the state-of-the-art approaches, i.e., the link model and hall model. We found *DTM* could produce more complete and comprehensive view of software evolution, which is useful for developers and other project stakeholders to understand the changes of development topics from different aspects in a time interval.

Our approach can be improved by choosing more optimal preprocessing steps and parameter values. So in our future work, we want to perform more case studies to estimate the threshold of pruning and the set of stop words to be removed. In addition, the number of topics may be too small or too large, so we want to find a good estimation of the number of topics [7]. In our studies, we compare our approach with the state of the art based on the combination of the content and strength evolution, we want to conduct more comparative studies from the perspective of either one of them. Finally, we want to conduct more experiments on other types of projects to evaluate the generality of our approach.

Acknowledgment

This work is supported partially by Natural Science Foundation of China under Grant No. 61402396, No. 61472343, No. 61402395 and No. 61472344, partially by the Natural Science Foundation of the Jiangsu Higher Education Institutions of China under Grant No. 13KJB520027, partially by the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University under Grant No. KFKT2014B13, partially by the Jiangsu Training Programs of Innovation and Entrepreneurship for Undergraduates under Grant No. 201411117066X, and partially by the Cultivating Fund for Science and Technology Innovation of Yangzhou University under Grant No. 2013CXJ025.

References

[1] S. Wang and D. Lo, "Version history, similar report, and structure: putting them together for improved bug localization," in *22nd International Conference on Program Comprehension, ICPC 2014, Hyderabad, India, June 2-3, 2014*, 2014, pp. 53–63.

[2] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance*, vol. 19, no. 2, pp. 77–131, 2007.

[3] X. Sun, B. Li, Y. Li, and Y. Chen, "What information in software historical repositories do we need to support software maintenance tasks? an approach based on topic model," in *Computer and Information Science*. Springer International Publishing, 2015, pp. 27–37.

[4] A. E. Hassan, "The road ahead for mining software repositories," in *Frontiers of Software Maintenance, 2008. FoSM 2008.* IEEE, 2008, pp. 48–57.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[6] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Validating the use of topic models for software evolution," in *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*. IEEE, 2010, pp. 55–64.

[7] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th International Conference on Software Engineering*, 2013, pp. 522–531.

[8] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring latent dirichlet allocation based feature location," *Empirical Software Engineering*, vol. 19, no. 3, pp. 465–500, 2014.

[9] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, "Studying software evolution using topic models," *Science of Computer Programming*, vol. 80, pp. 457–479, 2014.

[10] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's hot and what's not: Windowed developer topic analysis," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 339–348.

[11] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 95–104.

[12] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[13] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," in *ACM Sigplan Notices*, vol. 43, no. 10. ACM, 2008, pp. 543–562.

[14] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code," *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[15] G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proceedings of the 1st India software engineering conference*. ACM, 2008, pp. 113–120.

[16] P. C. Rigby and A. E. Hassan, "What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 2007, p. 23.

[17] D. Hall, D. Jurafsky, and C. D. Manning, "Studying the history of ideas using topic models," in *Proceedings of the conference on empirical methods in natural language processing*. Association for Computational Linguistics, 2008, pp. 363–371.

[18] Q. Mei and C. Zhai, "Discovering evolutionary theme patterns from text: an exploration of temporal text mining," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 198–207.

[19] D. M. Blei and J. D. Lafferty, "Dynamic topic models," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 113–120.

[20] S. W. Thomas, A. E. Hassan, and D. Blostein, "Mining unstructured software repositories," in *Evolving Software Systems*. Springer, 2014, pp. 139–162.

[21] R. Blumberg and S. Atre, "The problem with unstructured data," *DM REVIEW*, vol. 13, pp. 42–49, 2003.

[22] S. Grimes, "Unstructured data and the 80 percent rule," *Carabridge Bridgepoints*, 2008.

[23] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 99–108.

[24] B. Berliner *et al.*, "Cvs ii: Parallelizing software development," in *Proceedings of the USENIX Winter 1990 Technical Conference*, vol. 341, 1990, p. 352.

[25] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version control with subversion*. " O'Reilly Media, Inc.", 2008.

[26] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 1–10.

[27] E. Linstead, C. Lopes, and P. Baldi, "An application of latent dirichlet allocation to analyzing software evolution," in *Machine Learning and Applications, 2008. ICMLA'08. Seventh International Conference on*. IEEE, 2008, pp. 813–818.

[28] X. Sun, X. Liu, J. Hu, and J. Zhu, "Empirical studies on the nlp techniques for source code data preprocessing," in *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*, ser. EAST 2014, 2014, pp. 32–39.

[29] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 492–497.