Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Studying software evolution using topic models

Stephen W. Thomas[a,*], Bram Adams[b], Ahmed E. Hassan[a], Dorothea Blostein[a]

[a] *Software Analysis and Intelligence Lab (SAIL), 156 Barrie Street, School of Computing, Queen's University, Canada*

[b] *Lab on Maintenance, Construction and Intelligence of Software (MCIS), Département de Génie Informatique et Génie Logiciel (GIGL), École Polytechnique de Montréal, Canada*

ARTICLE INFO

Article history:
Received 7 March 2011
Received in revised form 20 June 2012
Accepted 20 August 2012
Available online xxxx

Keywords:
Software evolution
Topic model
Latent Dirichlet allocation
Mining software repositories

ABSTRACT

Topic models are generative probabilistic models which have been applied to information retrieval to automatically organize and provide structure to a text corpus. Topic models discover *topics* in the corpus, which represent real world concepts by frequently co-occurring words. Recently, researchers found topics to be effective tools for structuring various software artifacts, such as source code, requirements documents, and bug reports. This research also hypothesized that using topics to describe the *evolution* of software repositories could be useful for maintenance and understanding tasks. However, research has yet to determine whether these automatically discovered topic evolutions describe the evolution of source code in a way that is relevant or meaningful to project stakeholders, and thus it is not clear whether topic models are a suitable tool for this task.

In this paper, we take a first step towards evaluating topic models in the analysis of software evolution by performing a detailed manual analysis on the source code histories of two well-known and well-documented systems, JHotDraw and jEdit. We define and compute various metrics on the discovered topic evolutions and manually investigate how and why the metrics evolve over time. We find that the large majority (87%–89%) of topic evolutions correspond well with actual code change activities by developers. We are thus encouraged to use topic models as tools for studying the evolution of a software system.

1. Introduction

To combat the complexities of software development, researchers realized that they could mine the repositories related to a software project to look for answers. For example, the location of bugs can be accurately predicted based on source code metrics or change activity [1–3], traceability links between requirements documents and source code can be automatically established by advanced information retrieval techniques [4,5], and developers can be automatically advised which source code documents to edit to address a new bug report [6].

One of the fundamental challenges in mining software repositories is understanding and using the unstructured natural-language text found in many of the repositories, for example email archives, commit logs, bug reports, and source code identifiers and comments. A recent advancement has been the use of statistical *topic models*, an unsupervised information retrieval technique, to automatically extract topics from textual repositories, such as a snapshot of the source code [7–10], documentation [11,12], and bug reports [6] in an effort to understand and use the natural language text embedded in these repositories. In this context, *topics* are collections of words that co-occur frequently in the documents of a repository and usually have a clear semantic relationship. For example, one topic might contain the words *{mouse move up down over left*

[*] Corresponding author. Tel.: +1 6134536162.
E-mail addresses: sthomas@cs.queensu.ca (S.W. Thomas), bram.adams@polymtl.ca (B. Adams), ahmed@cs.queensu.ca (A.E. Hassan), blostein@cs.queensu.ca (D. Blostein).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.08.003

right} and another might contain *{file open read write close}*. In practice, it has been found that these topics serve to represent the major themes that span the corpus, providing many benefits over existing information retrieval techniques [13].

Understanding how the use of a topic evolves, i.e. changes, in a software repository over time could also provide many benefits for project stakeholders [14]. For example, stakeholders could monitor the *drift* of a topic, i.e., when the implementation of a topic in the source code gradually diverges from the original design (similar to architectural drift [15]). Because of refactoring, re-engineering, maintenance and other development activities, a topic that was once focused and modularized may become more scattered across the system over time, getting out-of-sync with the mental model that designers and architects have about the system. Automatically discovering and monitoring these topic drifts would be a useful technique for developers and project managers wishing to keep their project in good health. Topic evolutions could also be used by new developers wishing to quickly understand the history of certain aspects of the system, such as when specific features were added or removed from the project. Additionally, topic evolutions could be used to monitor the day-to-day development activities, answering questions such as "Who is working on what concepts?" and "What concepts changed since last week?"

One could measure the evolution of topics by applying a topic modeling technique to each version of the system separately, then linking topics in different versions together according to a similarity measure (for example, KL divergence [16]). Another more common method for discovering topic evolutions is to apply a topic modeling technique to all of the versions of the system at once (ignoring time), then mapping the topics back to individual versions of the system. One can then determine how the values of various topic metrics (e.g., *assignment* or *scattering* [10]) change over time. In this paper, we consider the latter approach, and use the term *topic evolution* to refer to the evolution of the topic's metrics over time.

Although previous work has applied topic models to the history of the source code of a project to recover such topic evolutions [14], it is not yet clear how or why the topics evolve as they do. Topics lie at a higher level of abstraction than other elements found in the source code, such as classes [17], and it is yet to be determined whether the automatically discovered topic evolutions are consistent with the actual changes made by developers (the *change activity*) in the source code. Topic evolution models were originally developed for and tested against natural language corpora, such as newspaper articles or conference proceedings. Our goal is to validate the use of topic evolution models to describe software change activities.

In this paper, we take a first step towards such a validation by performing a qualitative case study of topic evolution on two well-known and well-documented systems, JHotDraw and jEdit. We apply latent Dirichlet allocation (LDA) [18], a statistical topic modeling technique, to the release history of the systems' source code, compute several metrics on the discovered topics, and manually investigate the source code and project documentation to verify that the evolution of metric values is useful and consistent with the actual change activity in the source code. We use a simple characterization of topic evolutions and find that such topic evolutions are meaningful descriptions of the source code evolutions and have the potential to provide project stakeholders with valuable information for understanding and monitoring their project.

Specifically, the contributions of this paper can be summarized as follows.

- We formalize our approach for applying an existing topic evolution model to the source code history of a software project, and explore two visualizations of topic evolutions: line charts and heatmaps.
- We manually validate the topic evolutions discovered by our approach, and find that most (87%–89%) of the evolutions are useful and consistent with descriptions of the actual changes to the source code.
- We manually investigate the high-level relationship between code changes and topic evolutions, and find that topics evolve due to several different types of developer activities, including bug fixes, refactoring, and feature additions.
- We explore and quantify the patterns of topic evolutions, and find that the topic evolutions of JHotDraw, for example, are much more active than those of jEdit.

This paper is organized as follows. Section 2 provides background information on topic models and topic evolution models. Section 3 motivates and describes our research questions. Sections 4–8 present our detailed case studies on JHotDraw and jEdit. Section 9 enumerates threats to the validity of our study and outlines future work directions. Section 10 summarizes related work. Finally, Section 11 concludes the paper.

2. Background

In this section we introduce LDA, a popular topic modeling technique that is the building block of many topic evolution models. We then describe topic evolution models, introduce required notation, and present a set of metrics that can be used to measure topic evolutions.

2.1. Latent Dirichlet allocation

LDA is a popular statistical topic modeling technique [18]. *Topic models* are statistical models that infer latent topics to describe a corpus of text documents [13]. *Topics* are collections of words that co-occur frequently in the corpus. For example, a topic discovered from a newspaper collection might contain the words *{bank finance money cash loan}*, representing the "finance industry" concept; another might contain *{river bank water stream fish}*, representing the "river" concept. Documents can be represented by the topics within them, allowing the entire otherwise unstructured corpus to be organized in terms of this discovered semantic structure.

In the LDA model, each document is a *multi-membership mixture* of topics, meaning that each document can contain multiple topics, and each topic can be contained in more than one document. For example, a newspaper article might contain both the "education" and "funding" topics; another article might contain the "political" and "funding" topics. Similarly in the LDA model, each topic is a multi-membership mixture of words, meaning that topics can contain multiple words, and words can belong to multiple topics. In this way, LDA is able to discover a representation of ideas or themes that describe the corpus as a whole [13].

In the LDA model, each word in a document is said to be *generated* by a topic. By aggregating the number of words generated by each topic across all documents, we can obtain a sense of how important (or unimportant) a given topic is. While one topic may be responsible for generating 20% of all the words in the corpus, another may only be responsible for generating 2%. On the other hand, some topics generate a few words in many documents, while some topics generate many words in only a few documents. We capture these notions with *topic metrics* (Section 2.2.2).

The *words* (or sometimes *terms* or *tokens*) in the LDA model are an abstract unit, taking the form of any unique sequence or pattern of data elements. When dealing with text corpora, words can be any sequence of characters and do not need to belong to a pre-defined vocabulary or ontology—both the words "button" and "asdf" are equally valid in the LDA model.

LDA has recently been applied to a variety of domains, due to its attractive features. First, LDA enables a low-dimensional representation of text, which (i) uncovers *latent* semantic relationships and (ii) allows faster analysis on text [19]. Second, LDA is unsupervised, meaning no labeled training data is required for it to automatically discover topics in a corpus, making LDA attractive for use in practical settings where labeled training data is difficult or expensive to create. And finally, LDA has proven to be fast and scalable to millions of documents or more [20]. For these reasons, in this paper we use LDA as our topic model. However, our approach is general enough so that any topic model may be used in its place. We use the term "topic model" for the remainder of this paper to emphasize this generality.

2.2. Topic evolution models

A *topic evolution model* is a topic model that accounts for time in some way, allowing the documents to have timestamps and the corpus to be versioned. Topic evolution models are useful for detecting and analyzing how topics change, or evolve, over the lifetime of a corpus. For example, some topics might become evermore present, increasing their overall importance. Other topics may disappear, indicating a marked change in the underlying corpus.

Topic evolution models typically involve two general steps, although different models will vary slightly in each step. In the first step, topics are discovered from the time-stamped documents in a corpus in the usual way for topic models, based on word co-occurrences in the documents. In the second step, the amount of change to a topic at each point in time is measured by the topic evolution model, typically by computing a topic metric (such as the assignment metric in Section 2.2.2) on each individual version, then analyzing the metric across all versions.

Several topic evolution models have been introduced, but many make assumptions about the corpora and place constraints on the amount that a topic can evolve. For example, the Dynamic Topic model [21] assumes that topics evolve according to a normal distribution, while the Topics Over Time model [22] assumes that the lifetime of a topic will follow a beta distribution. While useful in some domains, the assumptions made by these models are too restrictive for use on source code histories.

Thus far in the software engineering literature, two topic evolution models have been used.

The Link model. In the Link model [23] (first applied to software repositories by Hindle et al. [12]), a topic model is applied to each version of the corpus separately. Since different versions of the corpus may have slightly different topics (LDA is probabilistic), a post-processing phase is required to link topics across successive versions. (For example, is the "GUI" topic discovered from version 1 the same as the "GUI" topic discovered in version 2?) The linking phase requires the use of a similarity threshold to determine if a topic found in one version is the same as a topic found in another. Finding an optimal threshold is an open research problem that makes analysis difficult.

The Hall model. In the Hall model [24] (applied to source code by Linstead et al. [14] and Thomas et al. [25]), a topic model is applied collectively to all the versions of the corpus at the same time. Unlike the Link model, the Hall model discovers a single set of topics that span the entire life-time of the corpus. In a post-processing phase, the corpus is sliced at various points of time for further analysis. This threshold-free model is the model we study in this paper.

2.2.1. Notation

Topic models, such as LDA, discover K ***topics***, $z_1, \ldots, z_K$, where K is an input to the model. For each topic z_i, an m-length word (or term) membership vector ϕ_{z_i} is produced that describes the probability that each unique word appears in topic z_i. For each ***document*** d_j in the corpus, $j = 1, \ldots, n$, LDA produces a K-dimensional topic membership vector θ_{d_j} that describes the probability that each topic appears in d_j. A document is a general notion in topic models, and can be any string of characters, although in software engineering research, a document is typically defined to be either an entire source code document or an individual method. We say that a document d_j contains $|d_j|$ words.

A ***version*** V of a corpus is a set of documents $\{d_1, d_2, \ldots\}$ with the same time-stamp. The ***version history*** H of a corpus is the set of versions of the corpus, $H = \{V_1, V_2, \ldots, V_v\}$. We say that d_{ij} is the document with index i in version V_j, and that there are $|V_j|$ documents in a particular version V_j of the corpus.

2.2.2. Topic evolution metrics

We measure how a topic changes over time by computing metrics on the topic at each point in time and comparing the values. In this section we describe three metrics that can be used to characterize topic evolutions, although many more exist.

The ***assignment*** of a topic is the sum of the topic memberships of all documents in that topic, which gives an indication of the total presence of the topic throughout the code [10]. A higher topic assignment means that a larger portion of the code is related to the topic. We define the assignment of topic z_k at version V_j as

$$\mathrm{A}(z_k, V_j) = \sum_{i=1}^{|V_j|} \theta_{d_{ij}}[k]. \tag{1}$$

The ***weight*** of a topic is similar to its assignment, but in addition considers the length of each document. The weight metric exactly captures how many words in the entire corpus were generated by a topic, whereas the assignment metric captures the portion of documents that were generated by a topic, and therefore can be skewed by small documents. We define the weight of topic z_k at version V_j by

$$\mathrm{W}(z_k, V_j) = \sum_{i=1}^{|V_j|} \theta_{d_{ij}}[k] * |d_{ij}|. \tag{2}$$

The ***scattering*** of a topic is the normalized entropy of that topic over all documents [10]. Entropy is a common metric used in information theory to determine how uncertain, or spread out, a distribution is [26]; we normalize it by the number of documents to account for differing numbers of documents in each version. A topic with a high entropy value will be more spread throughout the system than a topic with a low entropy value. We define the scattering of topic z_k at version V_j by

$$\mathrm{S}(z_k, V_j) = \frac{1}{|V_j|} * \left[- \sum_{i}^{|V_j|} \theta_{d_{ij}}[k] * \log(\theta_{d_{ij}}[k]) \right]. \tag{3}$$

Other topic metrics, which we only briefly describe, are: the ***tangle*** of a topic, which indicates how many other topics a given topic is usually co-located with in a document; the ***α-support*** of a topic, which indicates the number of documents that have a membership of α or higher in the topic; the ***turnover*** of a topic, which captures the number of new documents matching the topic at a given version, compared to the previous version; and the ***similarity*** between two topics, which describes how similar the word distributions are between the two topics.

Finally, the ***evolution*** E of a metric m of a topic z_k is a time-indexed vector of metric values for that topic: $E(z_k, m) = [m(z_k, V_1), m(z_k, V_2), \ldots, m(z_k, V_v)]$.

We define a ***change event*** in a topic evolution as an increase (*spike*), decrease (*drop*), or no change (*stay*) in a metric value between successive versions. We classify a change event as a spike or a drop if there is at least a δ% increase or decrease in metric value compared to the previous version, and as a stay otherwise. Formally, for a metric m of topic z_k at version V_j, the change $c = (m(z_k, V_j) - m(z_k, V_{j-1}))/(m(z_k, V_{j-1}))$ is classified as

$$\mathrm{Event}(m, z_k, V_j) = \begin{cases} \text{spike} & \text{if } c \geq \delta \text{, or if } m(z_k, V_{j-1}) = 0 \text{ and } m(z_k, V_j) > 0; \\ \text{drop} & \text{if } c \leq -\delta; \\ \text{stay} & \text{otherwise.} \end{cases} \tag{4}$$

In later sections, we will use the notion of change events to characterize and validate topic evolutions.

3. Motivation and research questions

Topic evolution models have the potential for automatically monitoring a project's source code over time, which can be useful to practitioners for several tasks.

Retrospective analysis. Often, practitioners are interested in understanding the history of a project [27]. *When was the Undo/Redo feature added? When was the file format changed from flat text to XML? What was changed for release 3.4.5?* Currently, the only way to answer these questions is to dig through release notes and commit messages (assuming the documentation is complete) or to manually inspect code changes. With topic evolution models, identifying these histories and changes can be done by examining the topics and their changes over time. Topics are higher-level descriptions of source code, compared to traditional techniques based on lines of code or class dependencies. By looking at an individual topic evolution (e.g., the

"XML" topic), one can easily view the history and trends of activities related to the topic. Similarly, by looking at all the topic change events at a particular version, one can determine which topics experienced modification at that version.

Preemptive maintenance. Imagine having a system that monitored the source code of a project and automatically alerted developers when the health of the source code was degrading—module boundaries were no longer clear, coupling between concepts were higher than desired, or the implementations of specific concepts were scattered across the system, making maintenance difficult and time consuming. Such a system would be helpful for maintenance teams of all sizes. Part of such a system might be realized by monitoring the scattering, tangling, and α-support metrics (Section 2.2.2) of a topic's evolution.

Real-time, high-level monitoring of source code changes. Project managers are often interested in how the project is progressing: whether goals are being met, which parts of the system the developers are currently working on and whether the project is on track. Traditional real-time monitoring of source code involves somewhat crude metrics, such as the number of lines of code or number of bugs fixed. While important, these metrics only capture that *something* is changing, but they fail to capture *what* is changing. Topic evolution models could play a crucial role of a much more detailed and specific monitoring system, one in which project managers could view which topics are exhibiting change right now, who is working on which topics, and whether the topics being modified are consistent with upcoming goals.

These benefits of topic models all make one key assumption: that topic evolution models, when applied to source code histories, describe the changes made to the source code in a way that is useful to a project stakeholder. But this assumption is not trivially validated: topic models were built for natural language corpora, which source code clearly is not. Further, changes to source code are often frequent and random, violating assumptions by many topic evolution models. Even if topic evolution models can be successfully applied to source code changes, it is not clear whether these discovered evolutions will represent the high-level concepts in which practitioners are interested.

In this paper, we investigate the assumption that topic evolution models are *valid* when applied to source code—that is, the models describe the changes made to the source code in a meaningful way. In particular, we focus on the following research questions.

RQ1 *How well do the discovered topic evolutions correspond to actual change activities in the source code?*
We wish to determine the accuracy of topic evolution models. Given a set of change events in a discovered topic evolution, as well as a set of change activities to the source code, what is the correspondence?

RQ2 *What is the relationship between code change categories and topic evolution?*
For those evolutions that are meaningful, we wish to perform a descriptive study to determine the common relationships between evolutions and *code change categories* (i.e., bug fixes, feature additions, and refactorings).

RQ3 *What are the patterns of topic evolution?*
We wish to quantify and study how topics evolve, to gain insight into both the abstract notion of topic evolution as well as into the development processes of the studied systems.

Answering RQ1 allows us to evaluate the use of topic models for studying the evolution of source code, bringing us one step closer towards a robust software monitoring technique built upon topic evolution models (Section 6). Answering RQ2 and RQ3 allows us to better understand *why* and *how* topics evolve in source code, strengthening our understanding of the results of topic models and software evolution in general (Sections 7 and 8).

4. Study methodology

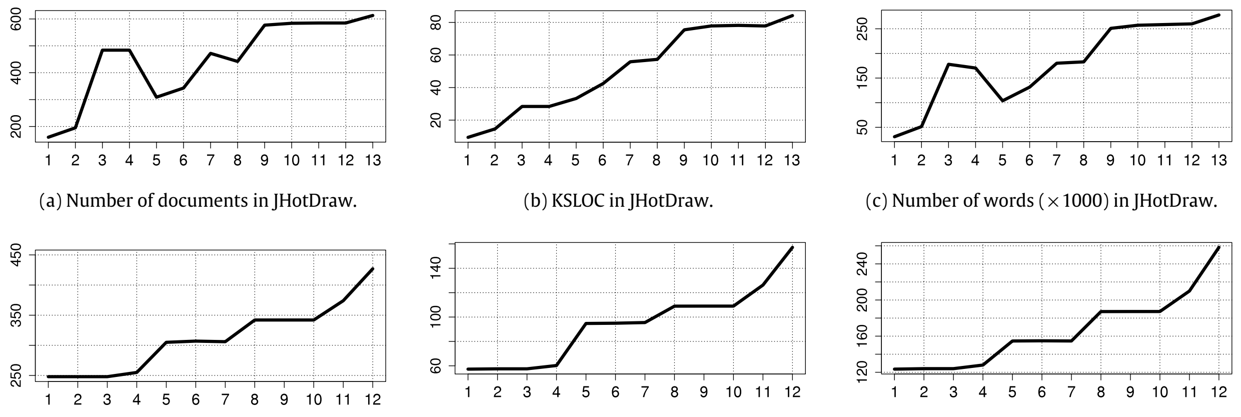
4.1. Systems under study

We address our research questions by performing in-depth case studies on the source code histories of two well-known software systems, JHotDraw and jEdit. JHotDraw is a medium-sized, open source, 2-D drawing framework developed in the Java programming language [28]. It was originally developed as an exercise of good program design and has become the de facto standard system for experiments and analysis in topic and concern mining (for example, by Robillard and Murphy [29] and Binkley et al. [30]). JHotDraw is a good choice for our purposes due to its good design practices and manageable size for manual analysis. We consider 13 release versions of JHotDraw (5.2.0–7.5.1). These versions were released over a nine year period and saw a growth of over 600% in the number of lines of code, several complete restructurings, and the addition of several new features (see Table 1 and Fig. 1(a)–(c)). During this time, 11 individual developers committed changes to the code base.

jEdit is a medium-sized, open source text editor written in the Java programming language [31]. jEdit focuses on providing rich features for developers, including syntax highlighting, macro scripting, and a comprehensive plug-in environment. jEdit is a good choice for our study because it is well organized, has extensive documentation, and has a manageable size for manual analysis. We consider 12 release versions of jEdit (3.0.0–4.2.0). The versions span a four year period where the code base grew by almost 200% (see Table 1 and Fig. 1(d)–(f)). During this time, 120 individual developers committed changes to the code base.

Table 1
Characteristics of our two systems under study, JHotDraw and jEdit.

	JHotDraw	jEdit
Purpose	Drawing framework	Text editor
Implementation language	Java	Java
License	Open source	Open source
Time period considered	Feb. 2001–Aug. 2010	Dec. 2000–Dec. 2004
Number of releases	13	12
Lines of code (thousands)	9.4–84	57–157
Number of source code documents	160–613	248–427
Number of committers	11	120



(a) Number of documents in JHotDraw. (b) KSLOC in JHotDraw. (c) Number of words (×1000) in JHotDraw.

(d) Number of documents in jEdit. (e) KSLOC in jEdit. (f) Number of words (×1,000) in jEdit.

Fig. 1. Data characteristics over time (version number) for JHotDraw and jEdit. In (c) and (f), we show the number of words remaining after the preprocessing steps have been performed.

4.2. Study setup

We preprocess the source code histories of each system by applying the typical source code preprocessing steps required by any information retrieval technique. We first isolate identifiers and comments, stripping away syntax and programming language keywords. We then tokenize each word based on common naming practices, such as camel case (`oneTwo`) and underscores (`one_two`). We remove common English language stop words (`the`, `it`, `on`) to reduce noise. We stem those words that remain (e.g., "changing" becomes "chang") to reduce the vocabulary size. Finally, we prune the vocabulary by removing overly common words (those that occur in more than 80% of the documents) and overly rare words (those that occur in less than 2% of the documents) because these words would have little effect on co-occurrence statistics. For JHotDraw, the preprocessing resulted in a total of 2.3 M words (comprised of 964 unique words) in 5833 documents, an average of 394 words per document. For jEdit, the preprocessing resulted in a total of 1.9 M words (816 unique) in 3744 documents, an average of 507 words per document.

For the LDA computation, we used MALLET version 2.0.6 [32]. MALLET is a highly scalable Java implementation of the Gibbs sampling algorithm. We ran for 10,000 sampling iterations, the first 1000 of which were used for parameter optimization [33]. We allowed MALLET to use hyper-optimization for the α and β input parameters, which are smoothing parameters for the model. Appendix D provides a brief replication guide for our study.

In addition to discovering topics, MALLET also automatically discovers a two- or three-word label for each topic that helps describe the topic in a compact way (e.g., "mouse click", "file format"). The label is based on commonly-occurring n-grams (i.e., co-located words) in the documents containing the topic. Although not all of the discovered labels are ideal (i.e., what a human would create) and sometimes produce double words (e.g., "elem elem"), we find the labels to be useful. For the remainder of this paper, when we present a label for a topic, we are presenting the label automatically discovered by MALLET.

4.2.1. Choosing the number of topics (K)

For any given corpus, there is no provably optimal choice for K [34]. The choice is a trade-off between coarser topics (smaller K) and finer-grained topics (larger K). Setting K to extremely small values results in topics that contain multiple concepts (imagine only a single topic, which will contain all of the concepts in the corpus!), while setting K to extremely large values results in topics that are too fine to be meaningful and only reveal the idiosyncrasies of the data. Our goal in this study is to discover topics of medium granularity, so we seek a non-extreme value for K.

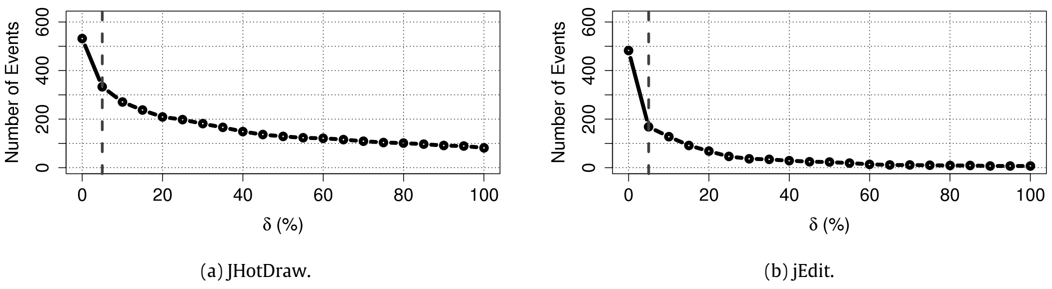


Fig. 2. Number of detected events (spikes and drops) for various δ thresholds, according to Eq. (4). The vertical dashed lines represents the "knee" in the curve, which we use as our cut-off value.

Previous work has used 45 topics for JHotDraw, arguing that 45 discovers topics of medium granularity [10,25]. To be comparable to these studies, we also set K to 45 for both JHotDraw and jEdit. However, as we show in Section 5.3, the discovered topics are stable (i.e., not particularly sensitive to the exact value of K) in at least the range of 30–60.

In the Hall model, each version of the source code will not necessarily contain all K topics. Since all of the versions are applied to the LDA model at once, and LDA is looking for a total of K topics, some versions may contain fewer than K topics. Imagine, for example, that new XML functionality is inserted into the code at version V_2, and that LDA discovers a topic representing XML concepts. Since no code dealt with XML in version V_1, this topic will not appear in V_1. We say that the topic is *born* at V_2, and that V_1 has at most K-1 topics. Similarly, a topic *dies* if all source code related to the topic is removed at some version. Thus, the number of topics K represents the total number of topics that exist across all time in the corpus, not necessarily at each point in time.

4.2.2. *Choosing the change threshold (δ)*

Due to the probabilistic nature of the LDA model, the same word in a given document may be assigned to different topics in different versions of the document. For example, the word "button" in a document d could be assigned to a "GUI"-related topic in version V_1, and then assigned to a "dialog box"-related topic in version V_2. Because of this small, unlikely, and statistically uninteresting change, both topics will experience a small change in their metric values (i.e., the weight metric will increase or decrease by 1).

To account for these small, probabilistic changes in topics over time, we introduce the δ threshold for determining if a metric value has changed significantly from one version to the next. This threshold, used in Eq. (4) to classify changes as spikes, drops, or stays, will help weed out uninteresting changes while preserving the interesting ones. Our goal is to set the threshold to a value that will achieve this balance.

Fig. 2 shows the number of events (i.e., spikes and drops) as a function of δ. When δ is zero, there are many events; almost one for every topic at every version. Many of these are obviously not interesting, as the metric's value only changes by less than 0.1%. In this paper, we are interested in studying major topic events, so we choose a δ value of 5% for both systems, as this appears to weed out many uninteresting events without weeding out too many major events (i.e., near the "knee" of the curves in Fig. 2). We note that different thresholds may be appropriate for other systems and other tasks, depending on the desired sensitivity to change.

5. Examining the discovered topics

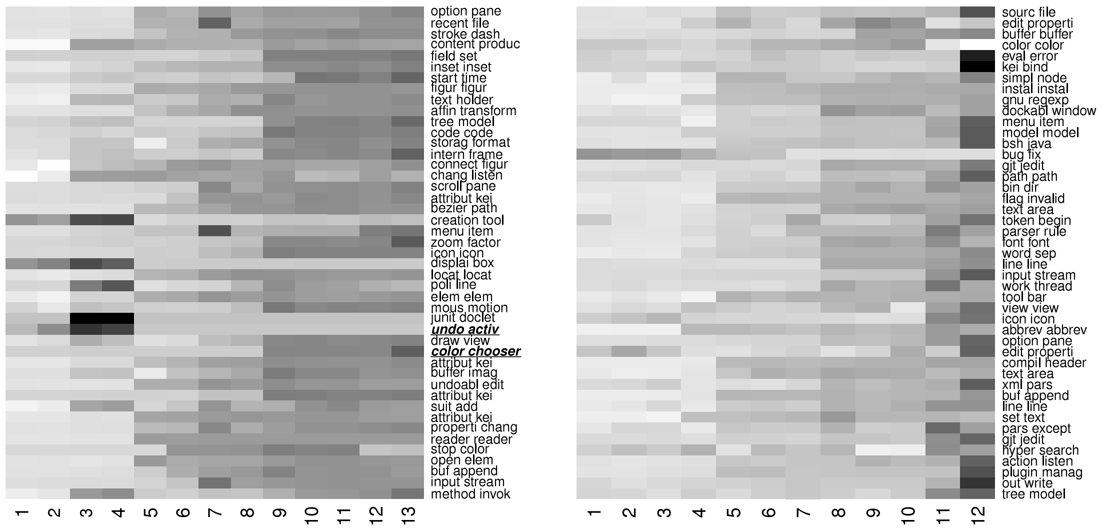
Before we begin our analyses of the discovered topic evolutions and their ability to describe actual changes in source code, we familiarize ourselves with the nature of the topics and evolutions discovered by our approach. To do so, we examine a few selected topics, consider various visualizations of their evolutions, and examine the stability of the topics.

5.1. *The topics themselves*

The full listing of the topics discovered by our approach is given in Appendix A (JHotDraw) and Appendix B (jEdit). We find topics that span a range of concepts, including "mous motion", "affin transform", "zoom factor", and "undoable edit" topics in JHotDraw and "xml pars", "tool bar", "bin dir", "font", "hyper search", and "gnu regexp" topics in jEdit. Table 2 shows selected topics, their top (stemmed) words, and their top three matching documents for each system. The groupings of top words seem to make sense to a human reader, based on their semantic similarities. Anyone who is familiar with XML, for example, will agree that the words "attribute", "element", and "child" naturally go together in this context. Further, the top documents seem to be a natural fit with both the given topic and the other topic documents. Just as other domains have found topics to make sense and be useful for their purposes, we find source code topics to be coherent and useful.

Table 2
Example topics from JHotDraw and jEdit. The labels are automatically generated by MALLET. The top words define the topic, and we display the documents with the three highest membership values (shown in parenthesis) for each topic.

Label	Top words	Top 3 related docs
JHotDraw		
"bezier path"	*path bezier node index coord mask point geom*	`BezierPath.java` (0.92), `GrowStroke.java` (0.59), `BezierPointLocator.java` (0.56)
"elem elem"	*attribut elem param child full attr return type*	`IXMLElement.java` (1.00), `XMLAttribute.java` (1.00), `XMLElement.java` (1.00)
"input stream"	*stream read end encod length offset line charact*	`PIReader.java` (0.54), `ContentReader.java` (0.52), `CDATAReader.ja va` (0.50)
jEdit		
"abbrev abbrev"	*abbrev mode expand line global expans set*	`Abbrevs.java` (1.00), `AbbrevsOptionPane.java` (0.44), `AddAbbre vDialog.java` (0.29)
"gnu regexp"	*index input match token retoken rematch current*	`RE.java` (1.00), `RETokenRepeated.java` (1.00), `REMatchEnumer ation.java` (1.00)
"plugin manag"	*plugin jar instal string edit version list return*	`PluginList.java` (1.00), `JARClassLoader.java` (0.53), `PluginList Handler.java` (0.51)



(a) JHotDraw. (b) jEdit.

Fig. 3. Heatmap views of the 45 topic evolutions discovered from the JHotDraw and jEdit source code. Darker cells indicate a higher assignment metric value. Rows are topics (shown with their automatically-generated topic labels) while columns are versions.

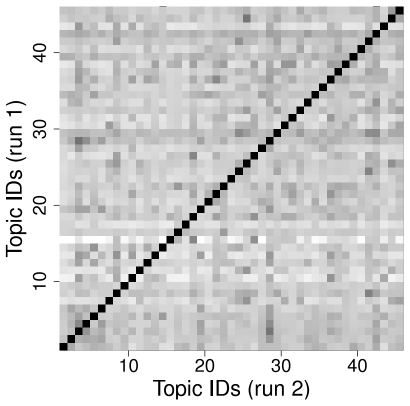
5.2. *Visualizing the evolutions*

To better understand the topic evolutions, we experiment with two visualization techniques, each with its own advantages and disadvantages.

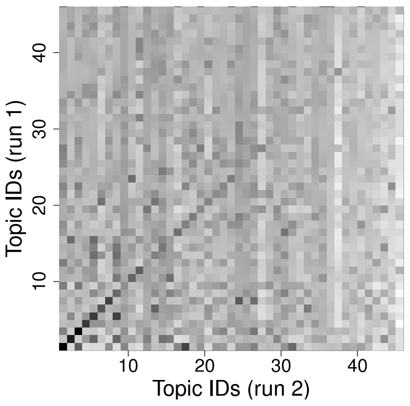
Line charts. Fig. 7 shows a traditional line chart view of the weight evolutions of four selected topics. The layout of a line chart is just as one would expect: the x-axis shows time and the y-axis shows the metric value at each point in time. Line charts are helpful for closely inspecting an individual topic evolution: to visualize the periods of spikes and drops and quickly reveal periods of activity and interest in the topic. The line chart is intuitive to understand but requires considerable space, especially when visualizing hundreds of topics.

For a complete listing of the discovered topics, Appendices A and B show all of the topics discovered for both systems, along with a simplified line chart to save space and give a quick impression of the general behavior of the topic evolution. The simplified line charts include up to four thick circles on the trend lines, indicating the first and last values on the line as well as the minimum and maximum values.

Heatmaps. Fig. 3 shows a heatmap view of the assignment evolutions for both JHotDraw and jEdit. The color of each cell represents the assignment value for a topic at a particular time, where darker colors indicate higher assignment values. This visualization allows us to quickly and compactly compare and contrast the trends exhibited by the various topics. For example, Fig. 3(a) indicates that some topics in JHotDraw (e.g., "undo activ", bolded in the figure) become increasingly active



(a) JHotDraw vs. itself.



(b) JHotDraw vs. jEdit.

Fig. 4. Example topic similarity matrices. (a) JHotDraw is compared against itself, which yields the most stable topics possible, indicated by the black diagonal. (b) JHotDraw is compared against a completely different system, jEdit, which yields completely unstable topics, indicated by a lack of a strong diagonal.

during the initial versions of the system, but then die at later versions. Other topics do not see any activity until later versions, such as the "color chooser" topic (bolded in the figure).

An interesting quality of the heatmap visualization is the ability to visually detect different phases of development, or releases with many changes to many topics. For example, there is a *visual wall* effect in Fig. 3(a) between versions V_4 and V_5, evidenced by a large color change in several topics. This suggests that there was a large development effort that concurrently affected multiple concepts. Indeed, version V_5 was a major release that experienced a multitude of refactorings and changes to the core framework of the system (cf. large drop in Fig. 1(a)).

5.3. *Examining topic stability*

The topics discovered from statistical topic models are a result of the input data, the input parameters, and the statistical sampling methods. *Topic stability* refers to how stable the discovered topics are across these factors [35]. For example, it would be undesirable if changing K from 45 to 46 would cause a completely new, unrelated set of topics to be discovered and all of our results to be affected. Likewise, sampling for a few more or less iterations ideally should not have a large impact on the topics. In this section, we investigate topic stability.

Topic stability is determined by measuring the Kullback–Leibler (KL) distance [16] between pairs of topics in different runs (or instantiations) of the topic model [35]. The KL distance between two topics is given by

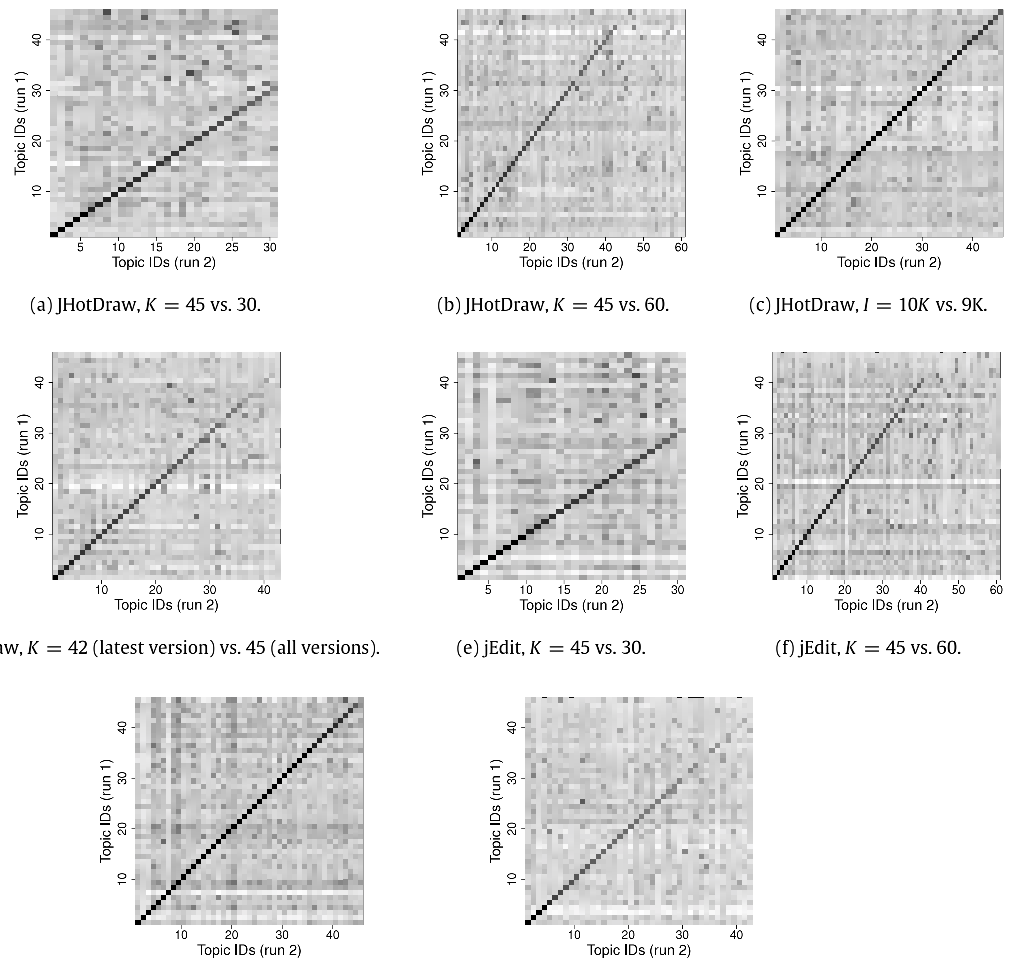
$$\mathrm{KL}(z_1, z_2) = \frac{1}{2}\sum_{i=1}^{N} \phi_{z_1}[i] \log_2 \frac{\phi_{z_1}[i]}{\phi_{z_2}[i]} + \frac{1}{2}\sum_{i=1}^{N} \phi_{z_2}[i] \log_2 \frac{\phi_{z_2}[i]}{\phi_{z_1}[i]}, \tag{5}$$

taking care to align the word orders in the word vectors. By computing the KL distance matrix for pairs of topics in each run, reordering the matrix in a greedy fashion so that the most similar topics are on the diagonal, and viewing the output as a heatmap, we can get a sense for how similar the topics are in the two runs. If the topics are indeed similar, that is, each topic in run 1 has a corresponding topic in run 2, then we can be confident that the topics are stable in the two runs. For example, Fig. 4(a) shows a similarity matrix of JHotDraw compared against itself, which will be the most stable case possible. A dark line down the diagonal indicates that, indeed, each topic in run 1 is very similar to a topic in run 2 (which are the same in this case). Fig. 4(b), on the other hand, shows the similarity matrix of the topics discovered from JHotDraw measured against those of jEdit, which we do not expect to be very similar. Less than 10 of the topics show high similarity (e.g., both systems have an "input stream" topic), but most topics are more dissimilar and appear in the figure as random noise.

Fig. 5 shows the results of topic stability for JHotDraw and jEdit for different numbers of topics ($K = 30, 45$, and 60), different numbers of sampling iterations ($I = 9000$ and 10,000), and different amounts of input data (all versions at once with $K = 45$ and only the latest version with $K = 42$). Overall, we find that the topics in both JHotDraw and jEdit are moderately to very stable, as indicated by the presence of dark diagonals that span most topics.

6. Manual analysis of validity (RQ1)

Our first research question examines the meaningfulness of topic evolution models applied to source code: do the discovered topics evolve because of actual change activity in the source code? If so, then topic evolution models may be



(a) JHotDraw, $K = 45$ vs. 30.

(b) JHotDraw, $K = 45$ vs. 60.

(c) JHotDraw, $I = 10K$ vs. 9K.

(d) JHotDraw, $K = 42$ (latest version) vs. 45 (all versions).

(e) jEdit, $K = 45$ vs. 30.

(f) jEdit, $K = 45$ vs. 60.

(g) jEdit, $I = 10K$ vs. 9K.

(h) jEdit, $K = 42$ (latest version) vs. 45 (all versions).

Fig. 5. Topic similarity matrices indicating the stability of the topic models. Each cell is the KL distance between pairs of topics. Darker cells mean less distance (more similarity). A column having one dark cell indicates that the topic in run 2 is very similar to exactly one topic in run 1.

an appropriate tool for understanding the change history of source code. However, if the discovered topic evolutions do not correspond well with the change activity in the source code, then topic evolutions are not a good choice.

We determine the meaningfulness of the discovered topic evolutions by analyzing in detail the change events of the evolutions. In particular, we perform a manual analysis of a random subset of the discovered change events of the evolutions. We selected enough random samples to yield a 90% confidence level with a margin of error of 10% [36], Our sample size, n, for each type of event of each system is calculated as

$$n = \frac{t}{1 + \frac{t-1}{N}}$$

where N is the total number of events of this type, $t = (Z^2 p(1-p))/B^2$, $B = 0.10$ and $1 - \alpha = 0.90$, so $Z = z_{\alpha/2} = z_{0.05} = 1.645$ [36]. Since we have no prior knowledge on the probability p of each event, we set p to 0.5. To account for differing numbers of spikes, drops, and stays, we sample each event independently. Thus, this calculation totals 132 out of 540 events for JHotDraw and 113 out of 495 events for jEdit.

In our study, there are four possible outcomes for each event:

- *True positive.* A spike or drop event is detected in a topic's evolution, and the source code exhibits a change relating to that topic.
- *False positive.* A spike or drop event is detected in a topic's evolution, but the source code does not exhibit a change relating to that topic.
- *True negative.* A stay event is detected in a topic's evolution, and the source code does not exhibit a change relating to that topic.

Table 3
Results of the manual analysis (RQ1) for JHotDraw and jEdit. Valid spikes and drops correspond to true positives, whereas valid stays correspond to true negatives.

	Spikes		Drops		Stays		Total	
	Sample size	Valid (%)	Sample size	Valid (%)	Sample size	Valid (%)	Sample size	Valid (%)
JHotDraw	53/251	87±10	26/42	96±10	53/247	89±10	132/540	89±10
jEdit	47/150	83±10	10/11	80±10	56/334	91±10	113/495	87±10

Version V_9	
	z_{18}
`EditServer.java`	0.69
`Install.java`	0.43
`LatestVersionPlugin.java`	0.55
...	
$A(z_{18}, V_9) =$	4.22

Version V_{10}	
	z_{18}
`EditServer.java`	0.60
`Install.java`	0.30
`LatestVersionPlugin.java`	0.29
...	
$A(z_{18}, V_{10}) =$	3.22

Fig. 6. An example from jEdit of an assignment change caused by noise in the LDA model. The topic memberships for topic 18 ("view") are shown for three documents for versions V_9 and V_{10}. In this example, none of the text in the three documents changed between the two versions, but due to the probabilistic processes of LDA, some of their topic memberships did change slightly. As a result, the assignment metric for this topic changes between the two versions.

– *False negative.* A stay event is detected in a topic's evolution, but the source code exhibits a change relating to that topic.

To aid us in our manual analysis, we developed a tool that randomly selects a given number of spike, drop, and stay events in the evolution of the weight metric. We choose the weight metric because it most accurately depicts the number of lines added and removed for a topic, making manual analysis more intuitive. For each event selected (between two versions V_j and V_{j+1}), the tool presents the following information (as shown in Appendix C).

– The topic label and the 10 most probable words.
– A plot showing the weight evolution, with the selected change event highlighted.
– A description of the change event (i.e., metric values before and after the event).
– A list of the top 15 documents matching the topic at version V_j. For each document, the tool lists the document's membership for this topic, the size of the document, a link to the source code diff between versions V_j and V_{j+1} of the document, and a link to the SCM entry for the document.
– A list of the top 15 documents matching the topic at version V_{j+1} (along with the metrics and links just mentioned).

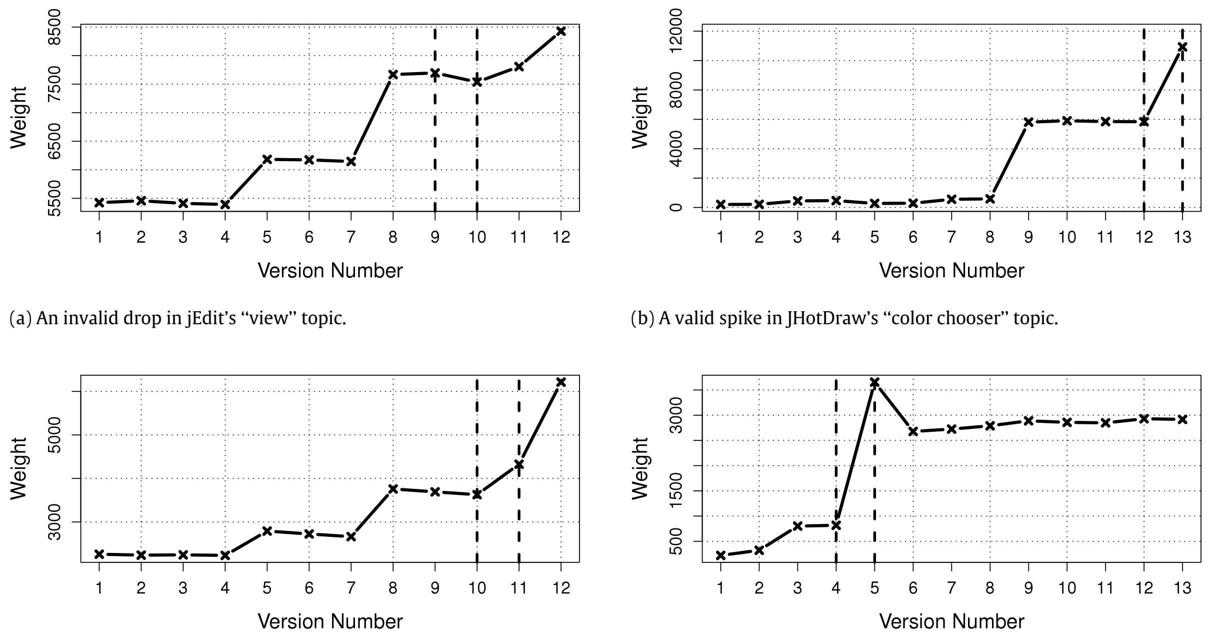
Armed with this information, we examined the project documentation (including release notes, commit logs, and source code comments), looking for evidence that supported each change event. If the change event clearly corresponded to an actual change activity in the source code, then we classified the event as valid. Examples of clear correspondence for spikes and drops include: a direct mention of the topic in the release notes; multiple new documents being added to or deleted from the system that match the topic; and multiple matching documents being heavily modified. For stay events, we looked for a lack of change, including the topic not being mentioned in the release notes, related documents remaining unchanged, and no new related documents being added or deleted. In any case, if we found no such correspondence, then we classified the event as invalid.

6.1. Results

Table 3 shows our results. After performing the manual analysis of the selected change events, we found that almost all (89% and 87% for JHotDraw and jEdit, respectively) of the randomly selected events correspond to actual change activities in the source code (i.e., true positives and true negatives), backed by at least one source of project documentation. In these cases, there was a clear correspondence between the topic changes and the changes in the source code.

We found that the change events that were not backed by documentation and did not correspond to change activity in source code (i.e., false positives) were largely caused by one of the following issues.

Noisy membership changes. As LDA is a probabilistic model, it is possible that a given document will be assigned slightly different topic memberships on different executions of the model. In the Hall model, two successive versions of a document, although exactly the same in textual content, might be assigned slightly different topic memberships. Further, if this happens for several documents that all match a given topic, then the topic's metrics will change, even though the source code never did (see Fig. 6).



(a) An invalid drop in jEdit's "view" topic.

(b) A valid spike in JHotDraw's "color chooser" topic.

(c) A spike in jEdit's "macro action" topic, which was caused by the addition of new functionality (AppleScript).

(d) A spike in JHotDraw's "elem elem" (XML) topic, which was caused by the addition of a new library (NanoXML).

Fig. 7. Example topic evolutions from JHotDraw and jEdit. The selected change event is highlighted by the vertical dashed lines.

Confounded topics. Ideally, LDA will discover topics that perfectly represent single real-world concepts. However, due to noise in the data, suboptimal parameter choices, and the sampling techniques of the inference algorithm, LDA sometimes discovers topics in which two or more logical concepts coexist—the topic is confounded with multiple logical concepts. Furthermore, these concepts do not necessary have an equal representation in the topic. In these cases, the evolutions for a topic can appear spurious and incorrect, due to a lighter concept receiving a legitimate spike or drop, while the larger concept should not experience any, or vice versa. An example of this is the "view" topic in jEdit. This topic contains the words "view", "buffer", and "edit", clearly representing the concept of different editor views of a file (i.e., buffer). This topic also contains the words "properti" and "set", representing the concept of managing properties (e.g., global properties or plugin properties). This single topic thus represents two concepts, a result of these concepts frequently being co-located in the source code. As a result, at versions when the "property" concept is changed, the "view" topic will exhibit a spike or drop, even though no code related to the "editor views" concept was actually changed.

We found that the majority (89%–91%) of stay events were valid (true negatives). Many topics indeed exhibited periods of inactivity: none of their related documents were changed, no new documents were added that matched the topic, and there was no mention of the topic in the project documentation. However, in a few cases, large refactorings or document name changes occurred that the topic evolutions were unable to detect (false negatives). In these cases, even though documents were moved from one directory to another and methods were moved from one document to another, the topic weight metric remained the same or very similar. As a result, the topic evolution exhibited a stay and the project stakeholders would not be informed about these changes. We note that in some refactorings, enough changes were made to produce a spike or drop in the topic, which *were* detected by our approach.

6.2. Examples

Fig. 7(a) shows a change event for the "view" topic in jEdit that we classified as invalid. In this example, the weight metric decreased by 23% between versions V_9 and V_{10}, which is indeed a drop according to our definition in Eq. (4). However, after manual analysis, it became clear that no actual changes occurred to any of the source code documents that matched this topic. The change in metric value was caused by noise in the LDA model: four of the seven documents that matched this topic received a lower topic membership in V_{10} than they did in V_9, even though no changes occurred to any of the documents during this period.

On the other hand, Fig. 7(b) shows a change event from JHotDraw that we classified as valid. The evolution for the "color chooser" topic experiences a large spike (87%) in the weight metric between versions V_{12} and V_{13}. The release notes for version V_{13} include the description: "*An UI delegate for JColorChooser has been added to the "Palette" look and*

Table 4
Relationship between code changes and topic evolutions in JHotDraw and jEdit. The columns may add up to more than 100% because the categories are not mutually exclusive.

	JHotDraw			jEdit		
	All (%)	Spikes (%)	Drops (%)	All (%)	Spikes (%)	Drops (%)
C1. Corrective evolution (i.e., bug fixes)	8	11	4	64	67	50
C2. Refactoring						
C2.1. New coding conventions	4	2	8	6	5	13
C2.2. New frameworks, libraries, etc.	10	4	0	2	3	0
C2.3. Internal improvement	79	70	96	28	26	38
C3. New functionalities and features	19	46	8	49	59	0

feel". Additionally, new source code documents like `DefaultColorSliderModel.java` and `HSLColorSpace.java` were added into the `org.jhotdraw.color` package, both matching the "color chooser" topic. Thus, there is a clear correspondence between the spike event in the topic evolution and the actual change activity in the source code.

6.3. Conclusion

In both of our case studies, we found that the majority of topics evolve due to actual change activities in the source code, with only a small minority of change events caused by noise or confounded topics in the probabilistic LDA model. We conclude that topic evolution models are appropriate and useful for describing source code evolution.

7. Investigation of code change categories (RQ2)

In this section, we investigate the relationship between topic evolutions and source code change categories. Longo et al. recently proposed three categories of software evolution interventions (i.e., reasons for software evolution) [37]:

C1. Corrective evolution (i.e., bug fixes)
C2. Refactoring (i.e., code improvement and adaptation)
  C2.1. Adoption of coding conventions and style
  C2.2. Adoption of new framework or libraries
  C2.3. Improvement of the internal structure of the code
C3. New functionalities and features.

We refer to each item above as a *change category*. These change categories are not mutually exclusive—it is possible, for example, for the adoption of new libraries to also result in new features, or for bug fixes to result in improvement of the internal structure of the code.

We adopt this taxonomy as a high-level model for why software changes. We revisit our two systems under study, JHotDraw and jEdit, and manually analyze the valid change events from the randomly-selected subset of change events. (We use the same subset as we did in RQ1.) We do not consider stay events because by definition stay events involve no code changes. We again use our analysis tool Appendix C, now with the goal of determining to which change category (or change categories) each change event belongs. We investigate project documentation and source code changes to determine whether each spike or drop is related to a bug fix, refactoring, or new functionalities or features.

7.1. Results

Table 4 shows the results of manually classifying each randomly-selected change event as one of the five possible change activities. We make a few observations.

- JHotDraw and jEdit do not exhibit the same behaviors. For example, JHotDraw development is less concerned with bug fixes (8% compared to 64%) while being more concerned with internal improvement (79% compared to 28%). This makes sense, because the development history of JHotDraw was focused on the internal design of the system with emphasis placed on increasing the use of design patterns and standard libraries.
- New coding conventions (C2.1) do not occur often for either system (4% and 6%).
- Drops are rarely caused by the addition of new functionalities or features. This makes intuitive sense, because adding features involves adding new code, which always causes spikes in the weight metric for one or more topics.
- Drops in JHotDraw almost always indicate one of the refactoring change activities. Drops in jEdit, on the other hand, could indicate any change activity.

Table 5
Characteristics of the discovered evolutions for JHotDraw and jEdit, for the assignment and weight metrics.

	JHotDraw				jEdit			
	Min.	Median	Mean	Max.	Min.	Median	Mean	Max.
Number of assignment spikes	0.0	2.0	1.9	4.0	0.0	0.0	0.5	3.0
Number of assignment drops	0.0	0.0	0.4	2.0	0.0	0.0	0.0	1.0
Number of assignment stays	7.0	10.0	9.7	12.0	8.0	11.0	10.5	11.0
Assignment change (spikes)	0.1	2.7	5.8	147.9	1.0	2.0	2.8	13.7
Assignment change (drops)	−147.4	−3.2	−11.9	−1.0	−2.4	−2.0	−1.9	−1.2
Number of weight spikes	1.0	4.0	3.9	6.0	0.0	1.0	1.4	4.0
Number of weight drops	0.0	1.0	0.8	4.0	0.0	0.0	0.1	2.0
Number of weight stays	4.0	7.0	7.4	11.0	7.0	10.0	9.5	11.0
Weight change (spikes)	39.0	670.0	1589.0	72230.0	54.9	537.5	903.3	7090.0
Weight change (drops)	−68210.0	−838.1	−3749.0	−68.9	−614.2	−365.6	−352.4	−99.1

7.2. Examples

Fig. 7(c) shows an example spike at version V_{11} in the evolution of the "macro action" topic in jEdit. In our manual analysis, we classified this spike as being related to the addition of new functionality (C3). This topic deals primarily with the macro scripting language feature in jEdit. The release notes for version V_{11} include the line: *"You can* [now] *run AppleScripts (compiled, uncompiled and standalone)"*, referring to the new capability of running scripts written in the AppleScript language. This capability was realized by adding new documents at version V_{11}, for example the `AppleScriptHandler.java` which has a membership of 0.88 in the "macro action" topic.

Fig. 7(d) shows an example spike at version V_5 in the evolution of the "elem elem" (XML) topic in JHotDraw. In our manual analysis, we classified this spike as being related to the addition of a new library. The release notes for version V_5 include the line: *"Added a tweaked version of NanoXML back into the framework"*. The NanoXML library consists of 24 documents, many of which match the "elem elem" topic with a membership of 0.50 or higher. Thus, this spike was related to the addition of a new library.

7.3. Conclusion

We have found that topics evolve due to a variety of underlying change activities, including bug fixes, refactoring efforts, and the addition of new functionalities. For this reason, we conclude that topic evolutions are a convenient and meaningful analysis tool for understanding source code evolution.

8. Exploring evolution patterns (RQ3)

We now examine topic evolution patterns. Our goal is to quantify common and uncommon behavior and explore the tendencies of topic evolutions.

8.1. Results

Table 5 summarizes the types and amounts of change events in the discovered topic evolutions for the two systems, for both the assignment and weight metrics. In both systems, there are more spikes than drops in the weight metric, consistent with the overall code growth over the studied periods. On average, a topic's weight evolution in JHotDraw has 4 spikes, 1 drop, and 7 stays; compare this to jEdit's relatively inactive 1 spike, 0 drops, and 10 stays. When a spike does occur, it does so by a median of 670 (JHotDraw) and 537 (jEdit) non-unique words (i.e., the median spike contains an addition of 670 and 537 non-unique words into the source code), but much smaller and larger spikes do occur. When a drop occurs, it is marked by the removal of a median of 838 (JHotDraw) and 365 (jEdit) non-unique words. In total, the weight evolutions in JHotDraw exhibited 251 spikes, 42 drops, and 247 stays, while jEdit exhibited 150 spikes, 11 drops and 334 stays.

We identify the following patterns of the evolution of the weight metric and provide relevant examples.

Overall growth. Ninety-six percent (JHotDraw and jEdit) of topics have a higher weight in the last version of the source code than they do in the first, indicating total overall growth over the lifetime of the source code. This can also be observed from Fig. 3, in which most topics appear darker towards the end of their lifetime. For example, in JHotDraw, the "input stream" topic (topic 2) has a weight of 306.0 in version V_1 and a weight of 4,920.8 in version V_{13}, capturing the increasing capabilities of JHotDraw for reading various input file formats. In jEdit, the "hyper search" topic (topic 5) increased its weight from 2,623.0 in version V_1 to 4,453.3 in version V_{12}, as the searching functionality in jEdit was greatly enhanced over time.

Similarly, only 4% (JHotDraw and jEdit) of topics have a lower weight in the last version than they do in the first version. For example, in JHotDraw, the "displai box" topic (topic 22) has a large weight in the first version of the source code, but almost no weight in the last version. This indicates the removal of the "display box" topic. (In this case, the removal occurred at version V_5, when a new `jhotdraw.draw` package was added to refactor older classes such as `RectangleFigure.java`, which contained the "displai box" topic.) No topics in either JHotDraw or jEdit had the same weight in the last version as they did in the first version.

Major events. Ninety-six percent (JHotDraw) and 16% (jEdit) of topics have at least one change event that changed the weight metric value by more than 75% of the previous version, indicating a major change of some kind. (To calculate this measure, we only included changes that involved an actual metric change of 50.0, to avoid scenarios where small changes to small values result in large percentage changes, for example when a topic weight changes from 2.0 to 4.0, i.e., a 100% increase that is obviously not important.) As JHotDraw was completely refactored and rewritten several times, it makes sense that most topics will exhibit major events. For example, the weight of the "color chooser" topic (topic 14) changed from 5,838.4 in version V_{12} to 10,930.3 in version V_{13}, a percentage difference of $+87\%$. As previously noted in Section 6.2, the color chooser module of JHotDraw was heavily modified in version V_{13}, changing its GUI look and feel and adding several new classes.

Births and deaths. Seven percent of topics in JHotDraw die at some point (i.e., have a positive weight value in some version V_j followed by a zero value in the remaining versions $V_{j+1}, \ldots, V_n$), indicating the removal of features and deletion of code. For example, in JHotDraw, the "undo activ" topic (topic 16) is removed at version V_5. (The corresponding release notes include the entry "Undo/Redo is now implemented based on the Swing undo package", indicating that JHotDraw no longer implements its own undo/redo functionality, and instead uses an off-the-shelf package in Swing.)

In jEdit, on the other hand, no topics die. This could mean that either jEdit is more stable and all features are useful, or that when features are removed, the obsolete code is not deleted.

Likewise, 45% (JHotDraw) and 7% (jEdit) first appear (i.e., are born) at some version $V_j, j > 1$. This indicates that JHotDraw has experienced a significant change between the first and last versions, since almost half of the topics were not present in the first version. For example, in JHotDraw, the "junit doclet" topic (topic 17) is first introduced in version V_2, which corresponds to when JUnit testing was first added to the system. jEdit is relatively more stable, but still exhibits some completely new topics. For example, the "instal" topic (topic 38) first appears in version V_5. Three new packages were added in this version, all related to an enhanced installation procedure for jEdit: `windows.jeditshell.jedinstl`, `windows.jeditshell.jeditinit`, and `windows.jeditshell.jeditlauncher`.

Constant topics. Whereas none of the topics in JHotDraw are constant (i.e., experience no spikes or drops), 22% of the topics in jEdit are. While these topics do exhibit some change over time, the change is always less than the δ threshold. This finding reveals that no feature or concept is safe from change in JHotDraw, likely due to the constant-improvement mentality of the project. On the other hand, features and concepts in jEdit follow more the "if it is not broken, do not fix it" paradigm. For example, the "xml pars" topic (topic 11), which captures jEdit's XML parsing feature, exhibits hardly any changes, and when it does, the changes are trivial.

Changes per version. On average, 60% (JHotDraw) and 7% (jEdit) of topics change in any given version. This implies that each release in JHotDraw brings changes to many topics concurrently, possibly indicating a hidden coupling of the code or perhaps the aggressive nature of the developers. On the other hand, fewer topics are changed at each release in jEdit, indicating either a more modular code design or a more focused development cycle.

Unstable topics. We call a topic *unstable* if it exhibits both more spikes and drops than stays—the topic metric rapidly spikes and drops in succession. Such a behavior could indicate a poorly designed topic that undergoes constant refactorings. We observe that only 5% (JHotDraw) and 0% (jEdit) of topics are unstable. An example unstable topic in JHotDraw is the "content produc" topic (topic 42), which represents the abstract concept of producing content in the included sample applications. Since the sample applications change significantly from version to version in order to highlight the changes made to the JHotDraw framework, the topic experiences unstable behavior.

Spike-only topics. Forty-three percent (JHotDraw) and 74% (jEdit) of topics exhibit spikes without also exhibiting drops. No topics (JHotDraw and jEdit) exhibit drops without spikes. This is due to the tendency of source code of the studied systems to only grow over time (Figs. 1(b) and 1(e)) as features and functionalities are added.

An example in JHotDraw is the "attribut kei" topic (topic 8), which is related to GUI buttons (which change internal attributes based on attribute keys). As JHotDraw grows over time and more GUI functionality is added, this topic experiences many spikes. In jEdit, the topic "out write" (topic 2), which captures the logging functionality of jEdit, is added to more and more of jEdit's classes over time, and rarely removed from a class.

8.2. Conclusion

Overall, the topic evolutions in JHotDraw are very active, as evidenced by their overall growth (96%), the number of major events (95%), few constant topics (0%), and many changed topics per version (60%). This is consistent with JHotDraw's active development cycle and multiple redesign efforts. Comparatively, the topic evolutions in jEdit are more calm: many are constant (22%), and few are changed per version (7%). This is consistent with jEdit's more mature and stable project status. Topics in both systems tend to grow, not shrink, as does the size of the source code. Finally, topics can be born and can die—not all topics exist in every version of the source code, representing major new or deleted features and concepts in the source code.

9. Limitations, threats to validity, and future work

In this section we enumerate the limitations and potential threats to the validity of our study, and propose future work directions.

9.1. Limitations and threats to validity

Quality of identifier names and comments. As with all source code analysis techniques based on identifier names and comments, our results are dependent on the quality of the naming conventions and commenting style of the project developers. Poor discipline or unorthodox conventions could result in topics with low semantic value. However, a recent study showed that most systems have identifiers and comments that are sufficient for such topic analyses [38]. Even still, our approach could be improved by recent advances in identifier splitting techniques [39] and vocabulary normalization [40].

Selected systems. We have focused our in-depth analysis efforts on JHotDraw and jEdit, due to their robust designs, extensive documentation, and manageable sizes. However, our results may be dependent on these qualities and thus generalize poorly to systems with worse designs. Furthermore, as both JHotDraw and jEdit are medium-sized open-source systems, we cannot yet be sure if our results generalize well to small or large sized open-source systems, or to any closed-source systems. We also cannot generalize our results with any confidence to systems from different domains, such as databases or web browsers. Additional case studies are needed to investigate these alternatives.

Parameter values. Our work involves choosing several parameters for LDA computation, perhaps most importantly the number of topics, K. Also required for LDA is the number of sampling iterations, as well as prior distributions for topic and document smoothing parameters, α and β. There is currently no theoretically guaranteed method for choosing optimal values for these parameters, even though the resulting topics are obviously affected by this choice. To counteract this limitation, we use the same value for K as previous work [10,25] and let MALLET automatically choose optimal values for α and β [32].

Additionally, our choice of δ (metric change threshold) was based on finding a knee in a curve of change events. We used this data-driven technique in an attempt to choose a value that eliminated most noisy changes and preserved most actual changes. Still, our results are affected by this subjective choice to some degree.

Preprocessing steps. We performed several preprocessing steps on the source code documents, such as splitting, stemming, stopping, and pruning. Although most research to date also performs some combination of these steps before applying information retrieval models to source code, there is currently no guidance or consensus on which steps are actually necessary or beneficial.

Manual analysis. We performed a detailed manual analysis of the discovered topic evolutions. However, as we were familiar with the details of the topic modeling techniques, it is possible that we were unknowingly biased (overly harsh or overly generous) during evaluation. Our work could be enhanced by also having outsiders to our project manually analyze the discovered topic evolutions. Further, as only the first author performed the manual analysis, our work can be enhanced by having additional people perform the analysis and comparing results.

9.2. Avenues for future work

Improved topic evolution models. We have found that some of the incorrectly discovered change events (i.e., spikes, drops, and stays) in our approach are due to noise in the Hall model. Since LDA is a probabilistic process based on sampling techniques, some randomness is inevitable when LDA assigns topics to documents. This randomness is responsible for some false-alarm change events. We believe that performing one or more smoothing operations on the discovered evolutions may mitigate this issue, further improving the usefulness of our approach.

In addition, some of the incorrectly discovered change events are due to confounded topics, i.e., those topics that try to represent more than one real-world concept. In future work we wish to explore the possibility of automatically detecting and removing (or splitting) these topics to further increase the accuracy of our approach.

Very recently, a new topic evolution model, called the Diff model [41], has been developed specifically for source code histories. The Diff model is an extension to the Hall model that applies LDA only to the changes of a document between successive versions, as opposed to the full document of each version. The idea is that since source code changes relatively infrequently, a given document is likely to be mostly similar to its previous version. This similarity can bias the topics that LDA discovers by skewing the word co-occurrence frequencies. The Diff model has been shown to provide modest to significant improvement over the Hall model, in terms of topic distinctness, evolution accuracy, and evolution sensitivity [41]. In future work, we wish to consider the Diff model as well as the Hall model.

Improved detection of change events. We use a simple technique to detect change events, namely Eq. (4), which is based on the δ threshold. In future work, we would like to explore more advanced detection techniques, such as trend detection or time-series analysis. More advanced techniques could help deal with noise in the LDA model and help detect slowly-spiking or slowly-dropping topics.

Additional metrics. In this paper, we have focused primarily on the assignment and weight metrics, which give different views on the overall presence of a topic in the source code. These metrics allowed us to perform our initial studies on the correspondence between changes in topic metrics and change activity in source code.

However, additional metrics exist for topics, as outlined in Section 2.2.2. We hypothesize that some of these metrics (or a combination of metrics) can be useful for automatically determining the type of change activity to the source code. For example, a refactoring or restructuring of the source code will likely result in a decreased scatter metric, while the weight metric will mostly stay unchanged or increase. We are investigating additional metrics.

Revision-level evolution models. We focused this paper on release-level topic evolutions, based on the public releases of each system. A more detailed approach would instead consider weekly or monthly snapshots of the code, or even each individual revision to the source control repository. The additional level of detail may be more revealing in that the topic evolutions will now show a more accurate time line of events and can be traced back to smaller change sets.

Industrial case studies. We have argued that the topic evolutions found in source code identifiers and comments can be useful for practitioners, namely for retrospective analysis, preemptive maintenance, and real-time monitoring of the concepts in source code. We should further validate these arguments by conducting case studies with practitioners in the industry. As a first step, we wish to distribute surveys to such an audience that describes topic evolutions and their potential uses, to gather practitioners' initial thoughts and views. A more detailed analysis would involve deploying our tools and techniques into the hands of actual developers and managers to measure practical benefits.

10. Related work

Recently, there has been an increasing interest in using topic models and other information retrieval techniques to help increase the utility of the unstructured and unlabeled text found in many software repositories. Most of this work is focused on extracting concepts from source code and calculating coupling/cohesion metrics from source code; some initial work has also been performed on the evolution of the software repositories, and some tools have been developed.

10.1. Modeling repository evolution using LDA

Hindle et al. apply the Link model to commit log messages in order to see what topics are being worked on by developers at any given time [12] . The authors apply the Link model (based on LDA) to a collection of commit logs over a period of 30 days, then link topics from successive periods using an 8-out-of-10 top-term similarity measure (i.e., if at least 8 of the 10 top words for a topic at period i are shared by a topic at period $i + 1$, then the topics are considered the same). The authors find LDA to be useful in identifying activity trends and present several visualization techniques to understand the results. Our work differs in that we extract topics from the source code as opposed to the commit log messages, and that we apply the Hall model instead of the Link model.

Linstead et al. use the Hall model (based on LDA) to analyze source code evolution, claiming that LDA provides better results than LSI [14]. The authors present line plots of topic assignment percentages over time for two systems, Eclipse and ArgoUML. These plots reveal integration points and other changes that shape a project's lifetime. We build on this work by formalizing the approach, considering additional topic metrics to better understand topic change events, and providing a detailed, manual analysis of the topic change events to validate and characterize the results of the approach.

In previous work, we explored the validity of applying topic models to source code histories [25]. We extend that work in this paper by conducting an additional case study (jEdit) and performing additional analyses on the systems (i.e., describing how and why topics evolve).

10.2. Concern mining

Marcus et al. initially presented the use of LSI for concern mining [42]. (Kellens et al. created a useful survey of concern mining techniques [43].) The authors find this technique to be better than existing static analysis techniques, due to its robustness, flexibility, and language-independence. We extend this work by using LDA to mine the concerns, and also consider the evolution of source code.

Kuhn et al. introduce *semantic clustering*, a technique based on LSI that groups source code documents sharing a similar vocabulary [9]. After applying LSI to the source code, the authors cluster the documents based on their textual similarity, resulting in semantic clusters of documents that implement similar functionalities. Our work uses LDA as the underlying topic model, and considers the evolution of source code.

Baldi et al. apply LDA to a single snapshot of source code in an effort to mine concerns and summarize functionality [10]. The authors also outline techniques to compute topic metrics such as scattering and tangling based on the output of the LDA model. In a similar technique, Maskeri et al. use LDA on a single snapshot of the source code to extract business topics in order to ease the comprehension of large systems for newcomers [44]. Building on this work, our work also considers the evolution of source code over time.

10.3. Source code metrics based on LSI/LDA

Poshyvanyk uses LSI to measure the conceptual coupling between classes in Object-Oriented systems [45]. The authors study the identifiers and comments in the source code and show that these data sources provide sufficient information to provide valid and useful measurements about classes, which can be used as predictors of fault potential. Similarly, Liu et al. used LDA to measure the cohesion of classes in Object-Oriented systems and show that their LDA-based approach is able to accurately describe cohesion of classes [46]. Kagdi et al. [47] also use LSI to compute the conceptual similarity between pairs of source code methods. In this approach, the authors use these metrics as part of a novel change impact analysis technique.

Bavota et al. develop an approach to support the automatic refactoring of so-called blob classes (i.e., classes that contain too much functionality and thus have a low cohesion score) [48]. To support this, they use three cohesion metrics for a class, one of which is the Conceptual cohesion based on LSI. Their approach is able to refactor the system so that it has an overall higher cohesion.

Gethers and Poshyvanyk introduce a new coupling metric, the Relational Topic-based Coupling (RTC) metric, based on a variant of LDA called Relational Topic Models (RTM) [49]. RTM extends LDA by explicitly modeling links between documents in the corpus. RTC uses these links to define the coupling between two documents in the corpus. The authors show that their proposed metric provides value because it is statistically different from existing metrics.

Ujhazi et al. define two new conceptual metrics that measure the coupling and cohesion of methods in software systems [50]. Both methods are based on a method's representation in an LSI subspace. The authors compare their metrics to an existing suite of metrics and find that the new metrics provide statistically significant improvements compared to previous metrics.

Oliveto et al. propose an approach to identify method friendships (two methods share similar functionality) with the overall goal of supporting Move Method refactoring (moving a method into a new class) [51]. The approach uses the Relational Topic Model (RTM) to calculate a metric for method friendships: methods are friends if RTM gives them a high similarity.

Oliveto et al. compared the effectiveness of four information retrieval techniques for traceability recovery, including LSI and LDA [52]. The authors showed that LDA provides unique dimensionality compared to the other three techniques.

All of the preceding work uses similar IR models as our work to accomplish different tasks. Whereas we are interested in finding source code topics and their evolution, the preceding work finds the relationship between classes and methods in a given snapshot of the source code.

10.4. Developer tools based on LDA

Tian et al. developed LACT, a tool for automatically organizing large collections of open-source software systems (e.g., SourceForge and Google Code) into related groups [53]. LACT uses LDA to discover the topics in each individual software system and creates groups based on these topics.

Savage et al. introduced a topic visualization tool, called Topic$_{XP}$, which supports interactive exploration of discovered topics located in source code [54].

Gethers et al. introduce a tool called CodeTopics which supports developers by showing which topics and High Level Artifacts (HLAs) to which a given source code is related [55]. CodeTopics uses the Relational Topic Models technique, a variant of LDA, to discover the underlying topics in the HLAs and source code.

While our work is focused on the accuracy of topic evolution, we can implement our approach into these types of tools.

11. Conclusion

In this paper, we performed a detailed investigation of the usefulness of topic evolution models for analyzing software evolution. We applied a topic evolution modeling technique to the source code history of a software system and computed metrics on the discovered topics. We found that most of these metric changes correspond well with actual software change activities (87%–89%), such as corrective evolution, internal improvements, and the addition of new features. The change events that we found to be inaccurate or invalid were mostly caused by noise in the probabilistic LDA model, although they did not occur often. Overall, we believe that topic models are an effective technique for automatically discovering and summarizing software change activities.

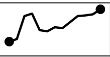
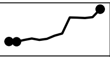
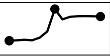
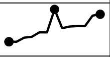
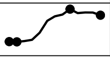
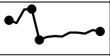
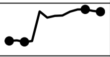
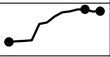
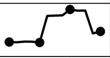
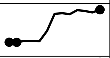
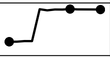
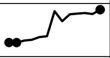
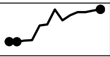
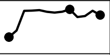
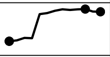
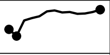
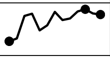
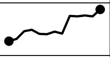
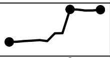
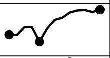
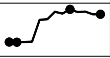
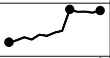
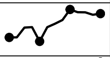
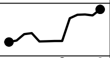
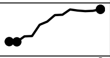
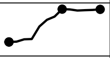
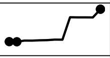
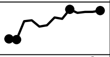
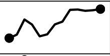
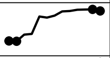
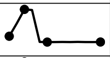
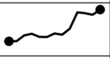
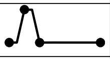
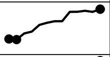
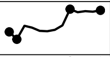
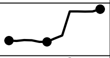
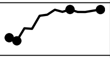
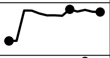
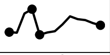
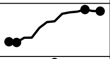
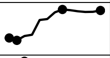
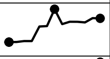
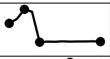
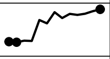
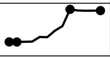
Our case studies on JHotDraw and jEdit suggest that using topic models to study the evolution of the source code of a software project could be useful for developers and other project stakeholders, providing a deep understanding of their system's history and allowing them to monitor and uncover design issues as they appear. Our findings encourage us to continue this work by exploring revision-level monitoring of topics, defining and measuring additional topic metrics, investigating additional topic models, and performing additional case studies.

Acknowledgments

We would like to thank the anonymous reviewers for their careful and insightful feedback, which helped improve the quality of this paper.

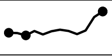
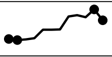
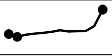
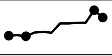
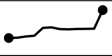
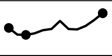
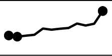
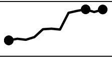
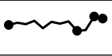
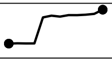
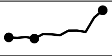
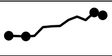
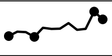
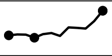
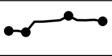
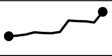
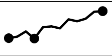
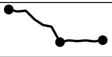
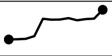
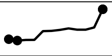
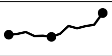
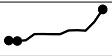
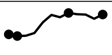
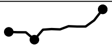
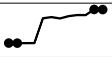
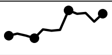
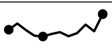
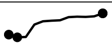
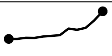
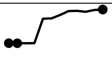
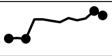
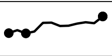
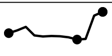
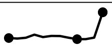
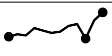
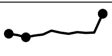
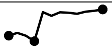
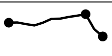
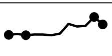
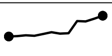
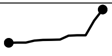
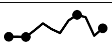
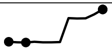
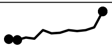
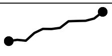
Appendix A. Trend lines for JHotDraw

The table below shows the topic trendlines for JHotDraw. For each topic, we show the automatically-generated topic label, the top eight words in the topic, and the trend line of topic assignment evolution.

Label	Top Words	Trend	Label	Top Words	Trend
1. method invok	*method except object invok resourc param target obj*		24. zoom factor	*grid bag constraint awt swing javax set add*	
2. input stream	*stream read end encod length offset line charact*		25. menu item	*action menu add window view item set jmenu*	
3. buf append	*elem attribut write buf append creat figur ixmlel*		26. creation tool	*tool figur creat mous draw creation view prototyp*	
4. open elem	*elem attribut current add object write docum ioexcept*		27. bezier path	*path bezier node index coord mask point geom*	
5. stop color	*color gradient space paint focu fraction stop arrai*		28. attribut kei object	*elem attribut read str style inherit ioexcept number*	
6. reader reader	*reader param system elem except entiti line data*		29. scroll pane	*set layout add pane swing editor line javax*	
7. properti chang	*properti listen chang enabl event action handler updat*		30. chang listen	*listen event figur chang remov draw list invalid*	
8. attribut kei	*button editor add attribut action label tool draw*		31. connect figur	*connect figur connector start end target point draw*	
9. suit add	*flavor suit transfer data drag focu compon clipboard*		32. intern frame	*border frame bar pane desktop set compon window*	
10. attribut kei	*color stroke kei attribut handl bound fill transform*		33. storag format	*format file draw input output filter extens storag*	
11. undoabl edit	*action edit undo label redo util bundl chang*		34. code code	*code point pointd param curv digit bound max*	
12. buffer imag	*imag buffer draw render width height hint set*		35. tree model	*font list node famili collect select tree path*	
13. attribut kei	*attribut kei set figur object map entri draw*		36. affin transform	*transform rectangl figur width draw stroke handl clone*	
14. color chooser	*color compon slider index icon model space system*		37. text holder	*text font figur holder edit area size tab*	
15. draw view	*view draw editor select constrain figur activ set*		38. figur figur	*figur draw child composit children add list remov*	
16. undo activ	*public command return undo activ figur set undoabl*		39. start time	*link code return param task target method figur*	
17. junit doclet	*test junit method doclet begin end javadoc except*		40. inset inset	*inset layout width height left bound size top*	
18. mous motion	*handl select figur mous view draw event evt*		41. field set	*field gbc set button grid label opac bag*	
19. elem elem	*attribut elem param child full attr return type*		42. content produc	*draw set paramet applet content url input produc*	
20. poli line	*point line angl width rectangl pointd height corner*		43. stroke dash	*stroke color map put fill width dash text*	
21. locat locat	*handl locat owner bound transform pointd figur point*		44. recent file	*view file applic app action project set chooser*	
22. displai box	*figur public box displai draw decor return read*		45. option pane	*code sheet pane messag listen option dialog compon*	
23. icon icon	*icon descriptor bean properti event gen method color*				

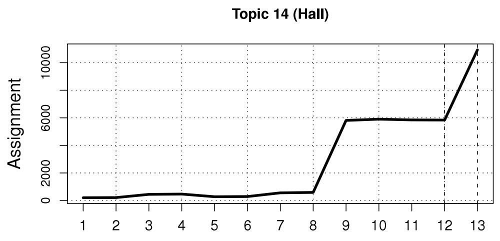
Appendix B. Trend lines for jEdit

The table below shows the topic trendlines for jEdit. For each topic, we show the automatically-generated topic label, the top eight words in the topic, and the trend line of topic assignment evolution.

Label	Top Words	Trend		Label	Top Words	Trend
1. tree model	*tree event node path model select result mous*			24. font font	*font print color style size text width privat*	
2. out write	*log url error privat return elem file els*			25. parser rule	*rule token context line set last parser keyword*	
3. plugin manag	*plugin jar instal string edit version list return*			26. token begin token	*activ move liter state kind start token check*	
4. action listen	*action list add con button select label set*			27. text area	*text area regist edit select caret set public*	
5. hyper search	*search replac set view buffer matcher find edit*			28. flag invalid	*offset size defin flag type stack format endif*	
6. gjt jedit	*view macro action edit record code repeat buffer*			29. bin dir	*instal size dir system public progress oper directori*	
7. pars except	*token node scope except consum close pars scan*			30. path path	*file path directori browser view session entri filter*	
8. set text	*add set box border action layout panel listen*			31. gjt jedit	*public return param edit compon pre comp gjt*	
9. line line	*line select caret start end offset buffer int*			32. bug fix revision	*revision syntax set fix bit bug updat improv*	
10. buf append	*append path buf return length str compar case*			33. bsh java	*method type return bsh variabl object interpret space*	
11. xml pars	*read buffer entiti attribut except encod elem type*			34. model model	*model tabl row col entri public return size*	
12. text area	*line color text highlight area gutter set font*			35. menu item	*menu item action add histori set mous popup*	
13. compil header	*file log script launcher path return modul server*			36. dockabl window	*window dockabl position width entri contain height top*	
14. edit properti	*line marker edit return buffer properti set file*			37. gnu regexp	*index input match token retoken rematch current mymatch*	
15. option pane	*option edit properti pane set box select add*			38. instal instal	*str instal kei path data reg error log*	
16. abbrev abbrev	*abbrev mode expand line global expans set hashtabl*			39. simpl node	*node eval type interpret jjt simpl callstack child*	
17. icon icon	*icon menu properti color return edit code style*			40. kei bind	*kei bind event shortcut return evt code case*	
18. view view	*buffer view file edit log properti set return*			41. eval error	*object type error primit eval field obj interpret*	
19. tool bar	*pane edit bar tool text buffer split area*			42. color color	*color style tabl option set add border background*	
20. work thread	*thread request progress work run log pool abort*			43. buffer buffer	*buffer view edit set properti updat jedit pane*	
21. input stream	*stream read code buffer length size write input*			44. edit properti	*mode properti edit set tab line indent size*	
22. line line	*line offset fold info method buffer start count*			45. sourc file	*error eval except interpret messag file sourc target*	
23. word sep	*index word text length start charact return end*					

Appendix C. Manual analysis tool

Below is an excerpt from our manual analysis tool. The excerpt shows a single change event, in this case a spike between versions V_{12} and V_{13} of JHotDraw. In the tool, the “diff” and “scm” words are clickable, taking the user to a web page containing the diff report of the document between the two versions or the SCM entry for that documents, respectively. In addition, when the user clicks on the version numbers (e.g., 7.5.1), the release notes for that version are displayed.



Hall Change Event #26

Change from version id 12 (7.4.1) to 13 (7.5.1) (dates 2010-01-17 to 2010-08-01)
Change in assignment from 28.70 to 38.09 (+32.70%)

Topic words: "color chooser", {*color compon slider index icon model space system rgb chooser track*}

Top 15 documents that match at version 12 (SCM link: 7.4.1):

		Version 12		Version 13			
Name	Package	Theta	Size	Theta	Size		
AbstractColorSystem.java	org.jhotdraw.color	1.00	25	NA	NA	diff	scm
AbstractHarmonicRule.java	org.jhotdraw.color	1.00	34	1.00	38	diff	scm
CMYKNominalColorSystem.java	org.jhotdraw.color	1.00	98	NA	NA	diff	scm
HSLRGBColorSystem.java	org.jhotdraw.color	1.00	154	NA	NA	diff	scm
HSLRYBColorSystem.java	org.jhotdraw.color	1.00	169	NA	NA	diff	scm
HSVRYBColorSystem.java	org.jhotdraw.color	1.00	134	NA	NA	diff	scm
RGBColorSystem.java	org.jhotdraw.color	1.00	37	NA	NA	diff	scm
HSVRGBColorSystem.java	org.jhotdraw.color	0.99	110	NA	NA	diff	scm
HarmonicRule.java	org.jhotdraw.color	0.97	32	1.00	32	diff	scm
DefaultColorSliderModel.java	org.jhotdraw.color	0.91	420	0.92	547	diff	scm
Colors.java	org.jhotdraw.draw.action	0.91	32	NA	NA	diff	scm
SimpleHarmonicRule.java	org.jhotdraw.color	0.90	106	0.90	101	diff	scm
ColorSystem.java	org.jhotdraw.color	0.88	56	NA	NA	diff	scm
ColorTrackImageProducer.java	org.jhotdraw.color	0.87	241	0.88	264	diff	scm
CompositeColor.java	org.jhotdraw.color	0.84	121	0.40	554	diff	scm

Top 15 documents that match at version 13 (SCM link: 7.5.1):

		Version 12		Version 13			
Name	Package	Theta	Size	Theta	Size		
AbstractHarmonicRule.java	org.jhotdraw.color	1.00	34	1.00	38	diff	scm
HarmonicRule.java	org.jhotdraw.color	0.97	32	1.00	32	diff	scm
PaletteColorSliderModel.java	org.jhotdraw.gui.plaf.palette.colorchooser	NA	NA	0.95	129	diff	scm
HSLPhysiologicColorSpace.java	org.jhotdraw.color	NA	NA	0.95	210	diff	scm
ColorSquareImageProducer.java	org.jhotdraw.color	NA	NA	0.94	298	diff	scm
HSVPhysiologicColorSpace.java	org.jhotdraw.color	NA	NA	0.94	178	diff	scm
QuantizingColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.94	296	diff	scm
HSLColorSpace.java	org.jhotdraw.color	NA	NA	0.94	185	diff	scm
CMYKNominalColorSpace.java	org.jhotdraw.color	NA	NA	0.93	193	diff	scm
DefaultColorSliderModel.java	org.jhotdraw.color	0.91	420	0.92	547	diff	scm
HSVColorSpace.java	org.jhotdraw.color	NA	NA	0.92	149	diff	scm
PolarColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.92	294	diff	scm
SimpleHarmonicRule.java	org.jhotdraw.color	0.90	106	0.90	101	diff	scm
AbstractColorWheelImageProducer.java	org.jhotdraw.color	NA	NA	0.89	123	diff	scm
ColorTrackImageProducer.java	org.jhotdraw.color	0.87	241	0.88	264	diff	scm

Appendix D. Replication guide

For the sake of completeness, we include a guide for replicating our study. For each system under study, perform the following steps.

1. Collect source code data. This can be performed by either checking out copies from the system's software configuration manager (e.g., SVN, CVS, Git) or by downloading source code snapshots from the system's webpage. Either way, the end result should be a collection of source code snapshots corresponding to the versions of interest (in our case, major releases).

2. Preprocess the data. Isolate source code identifiers and comments. Split the words based on common naming schemes. Convert all letters to lower case. Remove stop words. Stem each word. Remove overly common words (those that appear in more than 80% of the documents) and rare words (less than 2%).
3. Transform the data into MALLET format. If *input-dir* is the name of the top-level directory containing the preprocessed source code documents, and ${MALLET-BIN} is the path to the MALLET executable, then the command
```
${MALLET-BIN} import-dir --input input-dir --output data.mallet --keep-sequence
```
   will create the output file *data.mallet*.
4. Discover the topics. Run the command
```
   ${MALLET-BIN} train-topics \
--input data.mallet \
--num-topics 45 \
--num-iterations 10000 \
--optimize-burn-in 1000 \
--optimize-interval 100 \
--output-doc-topics allfiles.txt \
--output-topic-keys topics.dat \
--topic-word-weights-file wordweights.dat \
--word-topic-counts-file topiccounts.dat \
--xml-topic-phrase-report topic-phrases.xml
```
   substituting the number of topics, iterations, etc. as desired.
5. Analyze the data. The output file *wordweights.dat* contains the unnormalized, unsorted word weights for each topic. The output file *allfiles.txt* contains the resulting topic memberships for each input document. The output file *topic-phrases.xml* contains the topic labels for each topics. Using these documents, compute topic metrics according to Eqs. (1)–(3) on each slice of time (i.e., all documents at each version). Perform additional analyses and visualizations as desired. In our case, we relied on the R statistical environment [56].

References

[1] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (4–5) (2012) 531–577.
[2] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change bursts as defect predictors, in: Proceedings of the 21st International Symposium on Software Reliability Engineering, 2010, pp. 309–318.
[3] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, A.E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: Proceedings of the 26th International Conference on Software Maintenance, 2010, pp. 1–10.
[4] H.U. Asuncion, A.U. Asuncion, R.N. Taylor, Software traceability with topic modeling, in: Proceedings of the 32nd International Conference on Software Engineering, 2010, pp. 95–104.
[5] A. Marcus, J.I. Maletic, Recovering documentation-to-source-code traceability links using Latent Semantic Indexing, in: Proceedings of the 25th International Conference on Software Engineering, 2003, pp. 125–135.
[6] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, Source code retrieval for bug localization using latent Dirichlet allocation, in: Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp. 155–164.
[7] J.I. Maletic, A. Marcus, Supporting program comprehension using semantic and structural information, in: Proceedings of the 23rd International Conference on Software Engineering, 2001, pp. 103–112.
[8] D. Poshyvanyk, A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, in: Proceedings of the 15th International Conference on Program Comprehension, 2007, pp. 37–48.
[9] A. Kuhn, S. Ducasse, T. Girba, Semantic clustering: identifying topics in source code, Information and Software Technology 49 (3) (2007) 230–243.
[10] P.F. Baldi, C.V. Lopes, E.J. Linstead, S.K. Bajracharya, A theory of aspects as latent topics, ACM SIGPLAN Notices 43 (10) (2008) 543–562.
[11] S. Deerwester, S.T. Dumais, G.W. Furnas, T.K. Landauer, R. Harshman, Indexing by latent semantic analysis, Journal of the American Society for Information Science 41 (6) (1990) 391–407.
[12] A. Hindle, M.W. Godfrey, R.C. Holt, What's hot and what's not: windowed developer topic analysis, in: Proceedings of the 25th International Conference on Software Maintenance, 2009, pp. 339–348.
[13] D.M. Blei, J.D. Lafferty, Topic models, in: Text Mining: Classification, Clustering, and Applications, Chapman & Hall, London, UK, 2009, pp. 71–94.
[14] E. Linstead, C. Lopes, P. Baldi, An application of latent Dirichlet allocation to analyzing software evolution, in: Proceedings of the 7th International Conference on Machine Learning and Applications, 2008, pp. 813–818.
[15] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes 17 (4) (1992) 40–52.
[16] T.M. Cover, J.A. Thomas, Elements of Information Theory, John Wiley and sons, 2006.
[17] M. Lanza, The evolution matrix: recovering software evolution using software visualization techniques, in: Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, pp. 37–42.
[18] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent Dirichlet allocation, Journal of Machine Learning Research 3 (2003) 993–1022.
[19] C.X. Zhai, Statistical language models for information retrieval, Synthesis Lectures on Human Language Technologies 1 (1) (2008) 1–141.
[20] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, M. Welling, Fast collapsed Gibbs sampling for latent Dirichlet allocation, in: Proceeding of the 14th International Conference on Knowledge Discovery and Data Mining, 2008, pp. 569–577.
[21] D.M. Blei, J.D. Lafferty, Dynamic topic models, in: Proceedings of the 23rd International Conference on Machine Learning, ACM, 2006, pp. 113–120.
[22] X. Wang, A. McCallum, Topics over time: a non-Markov continuous-time model of topical trends, in: Proceedings of the 12th International Conference on Knowledge Discovery and Data Mining, ACM, 2006, pp. 424–433.
[23] Q. Mei, C.X. Zhai, Discovering evolutionary theme patterns from text: an exploration of temporal text mining, in: Proceedings of the 11th International Conference on Knowledge Discovery in Data Mining, 2005, pp. 198–207.
[24] D. Hall, D. Jurafsky, C.D. Manning, Studying the history of ideas using topic models, in: Proceedings of the Conference on Empirical Methods in Natural Language Processing, ACL, 2008, pp. 363–371.
[25] S.W. Thomas, B. Adams, A.E. Hassan, D. Blostein, Validating the use of topic models for software evolution, in: Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation, 2010, pp. 55–64.
[26] C.E. Shannon, A mathematical theory of communication, ACM SIGMOBILE Mobile Computing and Communications Review 5 (1) (2001) 3–55.

[27] T. Fritz, G.C. Murphy, Using information fragments to answer the questions developers ask, in: Proceedings of the 32nd International Conference on Software Engineering, 2010, pp. 175–184.
[28] E. Gamma, JHotDraw. URL http://www.jhotdraw.org/, 2007.
[29] M.P. Robillard, G.C. Murphy, Representing concerns in source code, ACM Transactions on Software Engineering and Methodology 16 (1).
[30] D. Binkley, M. Ceccato, M. Harman, F. Ricca, P. Tonella, Tool-supported refactoring of existing object-oriented code into aspects, IEEE Transactions on Software Engineering (2006) 698–717.
[31] S. Pestov, jEdit. URL http://www.jedit.org/, 2011.
[32] A.K. McCallum, Mallet: a machine learning for language toolkit. URL http://mallet.cs.umass.edu, 2002.
[33] T.L. Griffiths, M. Steyvers, Finding scientific topics, Proceedings of the National Academy of Sciences 101 (2004) 5228–5235.
[34] H.M. Wallach, I. Murray, R. Salakhutdinov, D. Mimno, Evaluation methods for topic models, in: Proceedings of the 26th International Conference on Machine Learning, 2009, pp. 1105–1112.
[35] M. Steyvers, T. Griffiths, Probabilistic topic models, in: Latent Semantic Analysis: A Road to Meaning, Laurence Erlbaum, 2007.
[36] R.L. Scheaffer, J.T. McClave, Probability and Statistics for Engineers, Duxbury Press Boston, Massachusetts, USA, 1994.
[37] F. Longo, R. Tiella, P. Tonella, A. Villafiorita, Measuring the impact of different categories of software evolution, Software Process and Product Measurement (2008) 344–351.
[38] S. Haiduc, A. Marcus, On the use of domain terms in source code, in: Proceedings of the 16th IEEE International Conference on Program Comprehension, 2008, pp. 113–122.
[39] N. Madani, L. Guerrouj, M. Di Penta, Y. Gueheneuc, G. Antoniol, Recognizing words from source code identifiers using speech recognition techniques, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, 2010, pp. 68–77.
[40] D. Lawrie, D. Binkley, C. Morrell, Normalizing source code vocabulary, in: Proceedings of the 17th Working Conference on Reverse Engineering, 2010, pp. 3–12.
[41] S.W. Thomas, B. Adams, A.E. Hassan, D. Blostein, Modeling the evolution of topics in source code histories, in: Proceedings of the 8th Working Conference on Mining Software Repositories, 2011, pp. 173–182.
[42] A. Marcus, A. Sergeyev, V. Rajlich, J.I. Maletic, An information retrieval approach to concept location in source code, in: Proceedings of the 11th Working Conference on Reverse Engineering, 2004, pp. 214–223.
[43] A. Kellens, K. Mens, P. Tonella, A survey of automated code-level aspect mining techniques, in: Transactions on Aspect-Oriented Software Development IV, in: LNCS, vol. 4640, 2007, pp. 143–162.
[44] G. Maskeri, S. Sarkar, K. Heafield, Mining business topics in source code using latent Dirichlet allocation, in: Proceedings of the 1st conference on India software engineering conference, 2008, pp. 113–120.
[45] D. Poshyvanyk, Using information retrieval to support software maintenance tasks, in: Proceedings of 25th International Conference on Software Maintenance, 2009, pp. 453–456.
[46] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, N. Chrisochoides, Modeling class cohesion as mixtures of latent topics, in: Proceedings of the 25th International Conference on Software Maintenance, 2009, pp. 233–242.
[47] H. Kagdi, M. Gethers, D. Poshyvanyk, M. Collard, Blending conceptual and evolutionary couplings to support change impact analysis in source code, in: Proceedings of the 17th Working Conference on Reverse Engineering, 2010, pp. 119–128.
[48] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, A two-step technique for extract class refactoring, in: Proceedings of the 25th International Conference on Automated Software Engineering, 2010, pp. 151–154.
[49] M. Gethers, D. Poshyvanyk, Using relational topic models to capture coupling among classes in object-oriented software systems, in: Proceedings of the 26th International Conference on Software Maintenance, 2010, pp. 1–10.
[50] B. Ujhazi, R. Ferenc, D. Poshyvanyk, T. Gyimothy, New conceptual coupling and cohesion metrics for object-oriented systems, in: Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation, 2010, pp. 33–42.
[51] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, A. De Lucia, Identifying method friendships to remove the feature envy bad smell, in: Proceeding of the 33rd International Conference on Software Engineering (NIER Track), 2011, pp. 820–823.
[52] R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, On the equivalence of information retrieval methods for automated traceability link recovery, in: Proceedings of the 18th International Conference on Program Comprehension, 2010, pp. 68–71.
[53] K. Tian, M. Revelle, D. Poshyvanyk, Using latent Dirichlet allocation for automatic categorization of software, in: Proceedings of the 6th International Working Conference on Mining Software Repositories, 2009, pp. 163–166.
[54] T. Savage, B. Dit, M. Gethers, D. Poshyvanyk, TopicXP: exploring topics in source code using latent Dirichlet allocation, in: Proceedings of the 26th International Conference on Software Maintenance, 2010, pp. 1–6.
[55] M. Gethers, T. Savage, M. Di Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, CodeTopics: which topic am I coding now? in: Proceedings of 33rd International Conference on Software Engineering, Formal Research Tool Demonstration, 2011, pp. 1034–1036.
[56] R. Ihaka, R. Gentleman, R: a language for data analysis and graphics, Journal of Computational and Graphical Statistics (1996) 299–314.