Mining Software Repositories with Topic Models

Stephen W. Thomas
sthomas@cs.queensu.ca

Technical Report 2012-586

School of Computing
Queen's University
Kingston, Ontario, Canada

Completed: December 2010
Published online: February 2012

Contents

Glossary

α:	in LDA, a smoothing parameter for document-topic distributions
β:	in LDA, a smoothing parameter for topic-term distributions
θ:	the document-topic matrix of a corpus
θ_d:	the topic membership vector of document d
ϕ:	the topic-term matrix of a corpus
A:	the term-document matrix of a corpus
C:	a corpus
d:	a document
N:	the number of terms in a document
V:	the vocabulary of a corpus
w:	a term
z:	a topic
AOP:	Aspect Oriented Computing
ICA:	Independent Component Analysis
IR:	Information Retrieval
LDA:	Latent Dirichlet Allocation
LSA:	Latent Semantic Analysis
LSI:	Latent Semantic Indexing
MSR:	Mining Software Repositories
NLP:	Natural Language Processing
PCA:	Principal Component Analysis
PLSA:	Probabilistic Latent Semantic Analysis
PLSI:	Probabilistic Latent Semantic Indexing
SVD:	Singular Value Decomposition
UML:	Unified Modeling Language
VSM:	Vector Space Model

1 Introduction

Mining Software Repositories (MSR) is a technique used in the field of software engineering (SE) focused on analyzing and understanding the data repositories related to a software development project [43, 50–52]. The main goal of MSR is to make intelligent use of these software repositories to help in the decision process of the software project. Specific examples of successes in MSR include predicting the locations of bugs in source code, discovering traceability links between requirements documents and source code, indexing the repositories to provide instant search functionality to project stakeholders, and inferring the social network of developers based on email communication patterns [127, 137].

The repositories in a software project typically contain unstructured text. Source code, while structured with the syntax of a programming language, contains a series of comments, identifier names, and string literals that capture the semantics and developer intentions of the document. Since developers tend to use meaningful identifier names, provide comments that describe the functionality of the code, and write error messages to the screen with string literals, looking at these three sources of text alone yields a good approximation of *what* is happening in the source code, without too much of the *how*. Other repositories, such as email archives, requirements documents, and bug reports, contain unstructured text in the more traditional sense.

An increasingly popular way to analyze unstructured text in other domains is by using *topic models*, which aim to uncover relationships between words and documents [3,20,124, 135]. Topic models, such as latent semantic indexing (LSI) and latent Dirichlet allocation (LDA), are attractive because (i) they require no training data, (ii) they are completely automated, and (iii) they can scale to thousands or millions of documents. Originally from the fields of Natural Language Processing (NLP) and Information Retrieval (IR), topic models have recently been discovered by the SE community.

In this report, we survey the software engineering field to determine how topic models have thus far been applied to one or more software repositories. Our primary goals are to characterize and quantify

– which topics models are being used,
– which SE tasks are being accomplished using topic models,
– which repositories are mined with topic models,
– how researchers are evaluating the results,
– what preprocessing steps are being performed on the data, and
– which tools and input parameter values are typical.

We evaluate a total of 71 articles from the software engineering literature that use topic models in some way. After providing a general overview of the literature, we collect and present 37 attributes on each article that help quantify and distinguish it from the others. We use the attributes to present aggregated findings, current research trends, and future research opportunities.

Structure of Report Section 2 gives an overview of software repositories, followed by an overview of topic modeling concepts. Section 3 presents a critical summary of the relevant articles from the literature. Section 4 outlines the research trends in the literature. Section 5 discusses repeatability issues and Section 6 presents future research opportunities. Finally, Section 7 concludes.

2 Background

In this section we provide background information on the field of Mining Software Repositories, describing which repositories are typically mined, and what information is desired from such repositories. We then briefly introduce the literature on topic modeling, starting with the seminal work in the 1980s on the Vector Space model and concluding with cutting-edge variants of LDA.

2.1 Mining Software Repositories

Mining Software Repositories is a technique used in the field of software engineering focused on analyzing and understanding the data repositories related to a software development project. The main goal of MSR is to make intelligent use of these software repositories to help in the decision process of the software project [43,50–52].

Software development projects produce several types of repositories during its lifetime, described in the following paragraphs. Such repositories are a result of the daily interactions between the stakeholders, as well as the evolutionary changes to the source code, test cases, bug reports, requirements documents, and other documentation. These repositories offer a rich, detailed view of the path taken to realize a software system. Analyzing such repositories provides a wealth of historical information that can directly benefit the future of a project [127,137].

Source Code The *source code* of a software project is the executable specification of the software system's behavior [67]. It is generally regarded by the developers as the single most important repository, since it is ultimately what creates the executable that gets delivered to the customer. The source code repository consists of a number of *source code documents* written in one or more programming languages. Source code documents are generally grouped into logical entities called *packages* or *modules*. The entire set of source code documents is usually referred to as the *system*.

For text mining tasks, the focus is usually placed only on the *identifiers* (i.e., variable names), *comments*, and *string literals* within the source code—the programming language keywords and symbols are discarded.

Bug and Vulnerability Databases A *bug database* (or *bug-tracking system*) is used to maintain information about the creation and resolution of bugs, feature enhancements, and other project maintenance tasks [119]. Typically, when developers or users experience a bug in a software system, they make a note of the bug in the bug database in the form of an *issue*, which includes such information as what task they were performing when the bug occurred, whether the bug is repeatable, and how critical the bug is on the functionality of the system. Then, one or more maintainers of the system investigate the issue, and if it becomes resolved, close the issue. All of these tasks are captured in the bug database. Popular bug database systems include Bugzilla [98] and Trac [37], although many exist.

In the literature, there is a difference in usage between the terms "bug", "defect", and "fault". We treat these terms as synonyms for the purpose of this report.

A *vulnerability database* stores a list of information security vulnerabilities and exposures [99]. The goal of a vulnerability database is to share data (e.g., current vulnerabilities being found in large systems) between communities and applications, as well as to give names to known vulnerabilities.

Mailing Lists and Chat Logs The *mailing list* (or *discussion archives*), along with the *chat logs* (or *chat archives*) of a project is an archival of the textual communication between developers, managers, and other project stakeholders [120]. The mailing list is usually comprised of a set of time-stamped email messages, which contain a *header* (containing the sender, receiver(s), and time stamp), a *message body* (containing the text content of the email), and a set of *attachments* (additional documents sent with the email). The chat logs contain the record of the instant-messaging conversations between project stakeholders, and typically contain a series of time-stamped, author-stamped text messages [14, 121, 122].

Source Control Database A *source control database* (or *revision control database*) is a system used to maintain and record the history of changes (or edits) to a repository of documents. Developers typically use source control databases to maintain the edits to the source code of the system. Popular source control databases (such as Concurrent Versions System (CVS) [13] and Subversion (SVN) [105]) allow developers to:

- *Checkout* a copy of the global repository to their local file system (i.e., make a local copy of the documents in the global repository);
- make local changes to existing documents, add new documents, delete existing documents, or alter the directory structure of the repository; and
- *commit* these local changes to the global repository.

Such revision control has two main advantages. First, it allows developers to change files on their own machine, independent of others who can access the repository. After the changes are made, they can be committed so that others can see the changes. This independence allows a parallel working cycle without having to email versions and copies back and forth. The second advantage of this versioning scheme is that it allows the history of each document to be automatically maintained. If a previous version of a document is desired, a developer can simply query the source control database to *revert* back to a previous version.

Requirements and Design Documents *Requirements documents*, usually written in conjunction with (or with approval from) the customer, are documents that list the required behavior of the software system [67]. The requirements can be categorized as either *functional*, which speicfy the "*what*" of the behavior of the program, and *non-functional*, which describe the qualities of the software (e.g., reliability or accessibility).

Design documents are documents that describe the overall design of the software system, including architectural descriptions, important algorithms, and use cases. Design documents can take the form of diagrams (especially UML diagrams [39]) or free-flowing text.

Execution Logs An *execution log* is a document that logs the output of an application during its execution of one or more predefined test cases. It generally contains a listing of which methods were called at which times, the values of certain variables, and other details about the state of the execution. Execution logs are useful when debugging the performance of large-scale systems with thousands or millions of concurrent users, since individual bugs are difficult to recreate on demand.

Software System Repositories A *software system repository* is a collection of (usually open source) software systems. These collections often contain hundreds or thousands of systems whose source code can easily be searched and downloaded for use by interested parties. Popular repositories include SourceForge [41] and Google Code [44].

These repositories contain a vast array of information about different facets of the software development project, from human communication to source code evolution. Thus, mining these repositories promises to bring rich information that can be used during the various phases of the project.

2.2 Topic Models

A *topic model* (or *latent topic model* or *statistical topic model*) is a model designed to automatically extract *topics* from a corpus of text documents [3, 20, 124, 135]. Here, a topic is a collection of terms that co-occur frequently in the documents of the corpus, for example {*mouse, click, drag, right, left*} and {*user, account, password, authentication*}. Due to the nature of language use, the terms that constitute a topic are often semantically related [22].

Topic models were originally developed in the field of natural language processing (NLP) and information retrieval (IR) as a means of automatically indexing, searching, clustering, and structuring large corpora of unstructured and unlabeled documents. Using topic models, documents can be represented by the topics within them, and thus the entire corpus can be indexed and organized in terms of this discovered semantic structure. Topic models enable a low-dimensional representation of text, which (i) uncovers *latent* semantic relationships and (ii) allows faster analysis on text [135].

2.2.1 Common Terminology

The topics models that we explore in this section share a similar terminology, which we present below. To make the discussion more concrete, we use the following running example of a corpus of three simple documents (excerpts of abstracts from SE articles):

Predicting the incidence of faults in code has been commonly associated with measuring complexity. In this paper, we propose complexity metrics that are based on the code change process instead of on the code.

Bug prediction models are often used to help allocate software quality assurance efforts (for example, testing and code reviews). Mende and Koschke have recently proposed bug prediction models that are effort-aware.

There are numerous studies that examine whether or not cloned code is harmful to software systems. Yet, few of these studies study which characteristics of cloned code in particular lead to software defects (or faults).

term (word) w**:** a string of one or more alphanumeric characters.

In our example, we have a total of 101 terms. For example, *predicting, bug, there, have, bug* and *of* are all terms. Terms might not be unique in a given document.

document d**:** an ordered set of N terms, $w_1, \ldots, w_N$.

In our example, we have three documents: d_1, d_2, and d_3. d_1 has $N = 34$ terms, d_2 has $N = 35$ terms, and d_3 has $N = 32$ terms.

corpus C**:** an unordered set of n documents, $d_1, \ldots, d_n$.

In our example, there is one corpus, which consists of $n = 3$ documents: d_1, d_2, and d_3.

vocabulary V**:** the unordered set of m unique terms that appear in a corpus.

In our example, the vocabulary consists of $m = 71$ unique terms across all three documents: *code, of, are, that, to, the, software, ...*.

term-document matrix A**:** an $m \times n$ matrix whose i^{th}, j^{th} entry is the weight of term w_i in document d_j (according to some weighting function, such as term-frequency).

In our example, we have

$$A = \begin{array}{l|ccc|} & d_1 & d_2 & d_3 \\ \hline \textit{code} & 3 & 1 & 2 \\ \textit{of} & 2 & 0 & 2 \\ \textit{are} & 1 & 2 & 1 \\ \ldots & \ldots & & \end{array}$$

indicating that, for example, the term *code* appears in document d_1 three times and the term *are* appears in document d_2 two times.

topic (concept) z**:** an m-length vector of probabilities over the vocabulary of a corpus.

In our example, we might have a topic

$$z_1 = \begin{array}{cccccccc} \textit{code} & \textit{of} & \textit{are} & \textit{that} & \textit{to} & \textit{the} & \textit{software} & \ldots \\ \hline 0.25 & 0.10 & 0.05 & 0.01 & 0.10 & 0.17 & 0.30 & \ldots \end{array}$$

indicating that, for example, when a term is drawn from topic z_1, there is a 25% chance of drawing the term *code* and a 30% chance of drawing the term *software*. (This example assumes a generative model, such as PLSI or LDA. See Sections 2.2.3 and 2.2.6 for the full definitions.)

topic membership vector θ_d**:** For document d, a K-length vector of probabilities of the K topics.

In our example, we might have a topic membership vector

$$\theta_{d_1} = \begin{array}{ccccc} z_1 & z_2 & z_3 & z_4 & \ldots \\ \hline 0.25 & 0.0 & 0.0 & 0.70 & \ldots \end{array}$$

indicating that, for example, when a topic is selected for document d_1, there is a 25% chance of selecting topic z_1 and a 70% chance of selecting topic z_3 .

document-topic matrix θ (document-topic matrix D): an n by K matrix whose i^{th}, j^{th} entry is the probability of topic z_j in document d_i. Row i of θ corresponds to θ_{d_i}.

In our example, we might have a document-topic matrix

$$\theta = \begin{array}{c|ccccc|} & z_1 & z_2 & z_3 & z_4 & \ldots \\ \hline d_1 & 0.25 & 0.0 & 0.0 & 0.70 & \ldots \\ d_2 & 0.0 & 0.0 & 0.0 & 1.0 & \ldots \\ d_3 & 0.1 & 0.4 & 0.2 & 0.0 & \ldots \end{array}$$

indicating that, for example, document d_3 contains topic z_3 with probability 20%.

topic-term matrix ϕ (topic-term matrix T): an K by m matrix whose i^{th}, j^{th} entry is the probability of term w_j in topic z_i. Row i of ϕ corresponds to z_i.

In our example, we might have a topic-term matrix:

$$\phi = \begin{array}{c|cccccccc} & \textit{code} & \textit{of} & \textit{are} & \textit{that} & \textit{to} & \textit{the} & \textit{software} & \ldots \\ \hline z_1 & 0.25 & 0.10 & 0.05 & 0.01 & 0.10 & 0.17 & 0.30 & \ldots \\ z_2 & 0.0 & 0.0 & 0.0 & 0.05 & 0.2 & 0.0 & 0.05 & \ldots \\ z_3 & 0.1 & 0.04 & 0.2 & 0.0 & 0.07 & 0.10 & 0.12 & \ldots \\ \ldots & \ldots & & & & & & & \end{array}$$

Armed with the above terminology, we now present a definition of topic models.

topic model: a model that takes as input a term-document matrix A and outputs a document-topic matrix θ as well as a topic-term matrix ϕ.

This definition describes the minimum requirements of a topic model. As we shall see, some topic models provide more than just the document-topic matrix θ and topic-term matrix ϕ—some also provide relationships and correlations between topics, multi-lingual topics, and labeled topics.

Some common issues arise with any language model:

synonymy: Two terms w_1 and w_2, $w_1 \neq w_2$, are *synonyms* if they possess similar semantic meanings.

polysemy: A term w is a *polyseme* if it has multiple semantic meanings.

We understand that the term *semantic* is hard to define and takes on different meanings in different contexts. In the field of IR, often a manually-created oracle is used (e.g., WordNet [96]) to determine the semantics of a term (i.e., relationships with other terms).

2.2.2 The Vector Space Model

While not a topic model itself, the *Vector Space Model* (VSM) is the basis for many advanced IR techniques and topic models. The VSM is a simple algebraic model directly based on the term-document matrix (Figure 1) [117]. In the VSM, a document is represented by its corresponding column vector in A. For example, if the term vector for a document d was [0 1 1 0 0], then according to the VSM, d contains two terms, namely those with indices 2

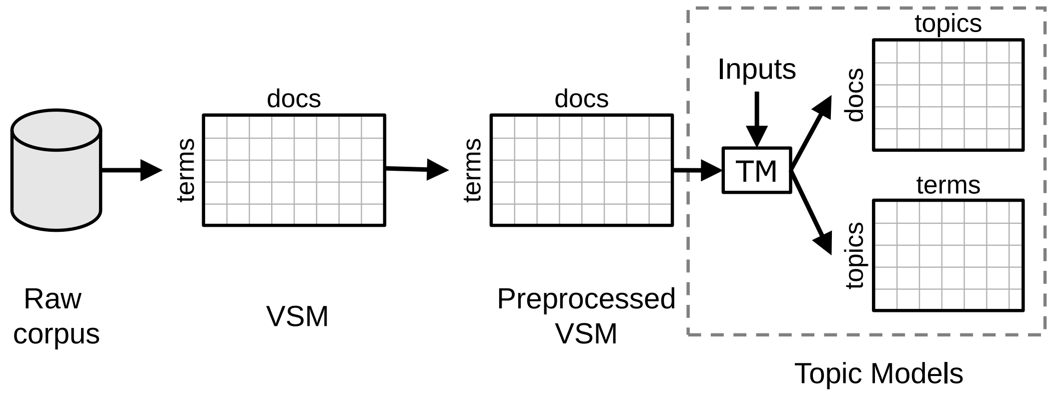


Figure 1: The path from a raw corpus to a topic model (TM). Here, "Topic Models" includes LSI, ICA, PLSI, LDA, and all LDA variants.

and 3. Likewise, it is possible to determine which documents contain a given term w by simply selecting the non-zero elements of w's vector in A.

With this representation, one can perform a query against the corpus as follows. First, compute the m-length term vector q for the query, as if it were another document in the corpus. Then, compute the semantic-relatedness (often the cosine-similarity) between q and each column of A. Finally, sort the semantic-relatedness results to obtain a ranked list of documents that are similar to q. In a similar way, it is possible to determine which of the original documents in the corpus are most similar to one another.

2.2.3 Latent Semantic Indexing

Latent Semantic Indexing (LSI) (or *Latent Semantic Analysis* (LSA)) is an information retrieval model that extends the VSM by reducing the dimensionality of the term-document matrix by means of *Singular Value Decomposition* (SVD) [35]. During the dimensionality reduction phase, terms that are related (in terms of co-occurrence) will be grouped together into topics[1]. This noise-reduction technique has been shown to provide increased performance over VSM in terms of dealing with polysemy and synonymy [6].

SVD is a factorization of the original term-document matrix A that reduces the dimensionality of A by isolating the *singular values* of A [116]. Since A is likely to be very sparse, SVD is a critical step of the LSI approach. SVD decomposes A into three matrices: $A = TSD^T$, where T is an m by $r = rank(A)$ term-topic matrix, S is the r by r singular value matrix, and D is the n by r document-topic matrix.

LSI augments the reduction step of SVD by choosing a reduction factor, K, which is typically much smaller than the rank of the original term-document matrix r. Instead of reducing the input matrix to r dimensions, LSI reduces the input matrix to K dimensions. There is no perfect choice for K, as it is highly data- and task-dependent. In the literature, typical values are between 50–300.

As in VSM, terms and documents are represented by row and column vectors in the term-document matrix. Thus, two terms (or two documents) can be compared by some distance measure between their vectors (e.g., cosine similarity) and queries can by formulated and evaluated against the matrix. However, because of the reduced dimensionality

[1]The original authors of LSI call these reduced dimensions "concepts", not "topics". However, to be consistent with other topic modeling approaches, we will use the term "topics".

of the term-document matrix after SVD, these measures are more equipped to deal with noise in the data.

2.2.4 Independent Component Analysis

Independent Component Analysis (ICA) [29] is a statistical technique used to decompose a random variable into statistically independent components (dimensions). Although not generally considered a topic model, it has been used in similar ways to LSI to model source code documents in a K-dimensional conceptual space.

Like LSI, ICA reduces the dimensionality of the term-document matrix to help reduce noise and associate terms. However, unlike LSI, the resulting dimensions in ICA are statistically independent of one another, which helps capture more variation in the underlying data [45].

2.2.5 Probabilistic LSI

Probabilistic Latent Semantic Indexing (PLSI) (or *Probabilistic Latent Semantic Analysis* (PLSA)) [56, 57] is a generative model that addresses the statistical unsoundness of LSI. Hofmann argues that since LSI uses SVD in its dimension-reduction phase, LSI is implicitly making the unqualified assumption that term counts will follow a Gaussian distribution. Since this assumption is not verified, LSI is "*unsatisfactory and incomplete*" [56].

To overcome this assumption, PLSI defines a generative latent-variable model, where the latent variables are topics in documents. At a high level, a generative model has the advantages of being able to be evaluated with standard statistical techniques, such as model checking, cross-validation, and complexity control; LSI could not be evaluated with any of these techniques. And since the latent variables are topics in documents, PLSI is also well-equipped to more readily handle polysemy and synonymy.

The generative model for each term in the corpus can be summarized with the following steps.

1. Select a document d_i with probability $P(d_i)$.

2. Select a topic z_k with probability $P(z_k|d_i)$.

3. Generate a term w_j with probability $P(w_j|z_k)$.

Given the observations in a dataset (i.e., terms), one can perform inference against this model to uncover the topics $z_i, ..., z_k$. We refer interested readers to the original articles [56, 57].

As was shown in subsequent articles (e.g., Blei et al. [22] and Zhai [135]), the generative model of PLSI suffers from at least two important problems. First, since d is used as an index variable in the first step, the number of parameters that need to be estimated grows linearly with the size of the corpus, which can lead to severe over-fitting issues. Second, since the z_k vectors are only estimated for documents in the training set, they cannot be easily applied to new, unseen documents.

2.2.6 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a popular probabilistic topic model [22] that has largely replaced PLSI. One of the reasons it is so popular is because it models each document as a multi-membership mixture of K corpus-wide topics, and each topic as a multi-membership mixture of the terms in the corpus vocabulary. This means that there are a set of topics that describe the entire corpus, each document can contain more than one of these topics, and each term in the entire repository can be contained in more than one of these topic. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus [20].

LDA is based on a fully generative model that describes how documents are created. Intuitively, this generative model makes the assumption that the corpus contains a set of K corpus-wide topics, and that each document is comprised of various combinations of these topics. Each term in each document comes from one of the topics in the document. This generative model is formulated as follows:

1. Choose a topic vector $\theta_d \sim \text{Dirichlet}(\alpha)$ for document d.
2. For each of the N terms w_i:
   (a) Choose a topic $z_k \sim \text{Multinomial}(\theta_d)$.
   (b) Choose a term w_i from $\text{p}(w_i|z_k, \beta)$.

Here, $\text{p}(w_i|z_k, \beta)$ is a multinomial probability function, α is a smoothing parameter for document-topic distributions, and β is a smoothing parameter for topic-term distributions.

The two levels of this generative model allow three important properties of LDA to be realized: documents can be associated with multiple topics, the number of parameters to be estimated does not grow with the size of the corpus, and, since the topics are global and not estimated per document, unseen documents can easily be accounted for.

Like any generative model, the task of LDA is that of *inference*: given the terms in the documents, what topics did they come from (and what are the topics)? LDA performs inference with *latent variable models* (or *hidden variable models*), which are machine learning techniques devised for just this purpose: to associate observed variables (here, terms) with latent variables (here, topics). A rich literature exists on latent variable models [10,15,78]; for the purposes of this report, we omit the details necessary for computing the posterior distributions associated with such models. It is sufficient to know that such methods exist and are being actively researched.

For the reasons presented above, it is argued that LDA's generative process gives it a solid footing in statistical rigor—much more so than previous topic models [22, 48, 124]. As such, LDA is better suited to discover the latent relationships between documents in a large text corpus.

Several variants and offshoots of LDA have been proposed. All of these variants apply additional constraints on the basic LDA model in some way. Although promising, the software engineering literature has yet to make use of many of these variants, and therefore we omit a detailed presentation of each.

– Hierarchical Topic Models (HLDA) [16, 17]. HLDA discovers a tree-like hierarchy of topics within a corpus, where each additional level in the hierarchy is more specific than the previous. For example, a super-topic "user interface" might have sub-topics "toolbar" and "mouse events".

- Cross-Collection Topic Models (ccLDA) [103]. ccLDA discovers topics from multiple corpora, allowing the topics to exhibit slightly different behavior in each corpus. For example, a "food" topic might contain the words *{food cheese fish chips}* in a British corpus and the words *{food cheese taco burrito}* for a Mexican corpus.
- Supervised Topic Models (sLDA) [21]. sLDA considers documents that are already marked with a response variable (e.g., movie reviews with a numeric score between 1 and 5), and provides a means to automatically discover topics that help with the classification (i.e., predicting the response variable) of unseen documents.
- Labeled LDA (LLDA) [38, 111]. LLDA takes as input a text corpus in which each document is labeled with one or more labels (such as Wikipedia) and discovers the term-label relations. LLDA discovers a set of topics for each label and allows documents to only display topics from one of its labels.
- Correlated Topic Models (CTM) [19]. CTM discovers the correlation between topics and uses the correlation when assigning topics to documents. For example, a document about the "genetics" topic is more likely to also contain the "disease" topic than the "X-ray astronomy" topic.
- Networks Uncovered by Bayesian Inference (Nubbi) [26]. Nubbi discovers relationships between pairs of entities in a corpus, where entities are specified as inputs into the model (e.g., people or places). For example, if the entities *George W. Bush* and *Gerald Ford* were input into Nubbi as entities, along with a corpus of political documents, then Nubbi might connect *George W. Bush* to *Gerald Ford* through a "republican" topic.
- Author-Topic Model [114]. The author-topic model considers one or more authors for each document in the corpus. Each author is then associated with a probability distribution over the discovered topics. For example, the author *Stephen King* would have a high probability with the "horror" topic and a low probability with the "dandelions" topic.
- Polylingual Topic Models (PLTM) [97]. PLTM can handle corpora in several different languages, discovering aligned topics in each language. For example, if PLTM were run on English and German corpora, it might discover the aligned "family" topics *{child parent sibling}* and *{kind eltern geschwister}*.
- Relational Topic Models (RTM) [25]. RTM models documents as does LDA, as well as discovers links between each pair of documents. For example, if document 1 contained the "planets" topic, document 2 contained the "asteroids" topic, and document three contained the "Michael Jackson" topic, then RTM would assign a stronger relationship between documents 1 and 2 than between documents 1 and 3 or documents 2 and 3, because topics 1 and 2 are more closely related to each other.
- Markov Topic Models (MTM) [131]. Similar to the Cross-Collection Topic Model, MTM discovers topics from multiple corpora, allowing the topics to exhibit slightly different behavior in each corpus.
- Pachinko Allocation Model (PAM) [68]. PAM provides connections between discovered topics in an arbitrary directed acyclic graph. For example, PAM might connect the "language processing" topic to the "speech recognition" topic, but not to the "snorkeling" topic.

– Topic Modeling with Network Regularization (TMN) [94]. TMN can model corpora which have networks defined between documents, such as social networks or call-graph dependencies. TMN discovers topics that overlay on the network in a meaningful way. For example, if a network was formed from author-coauthor relationships in academic articles, then topics that are assigned to author A have a high likelihood of being assigned to one of the coauthors of author A.

2.2.7 Topic Evolution Models

Several techniques have been proposed to extract the *evolution* of a topic in a time-stamped corpus—how the usage of a topic (and sometimes the topic itself) changes over time as the terms in the documents are changed over time. Such a model is usually an extension to a basic topic model that accounts for time in some way. We call such a model a *topic evolution model*.

Initially, the Dynamic Topic Model [18] was introduced. This model represents time as a discrete Markov process, where topics themselves evolve according to a Gaussian distribution. This model thus penalizes abrupt changes between successive time periods, discouraging rapid fluctuation in the topics over time.

The Topics Over Time (TOT) [132] model represents time as a continuous beta distribution, effectively removing the penalty on abrupt changes from the Dynamic Topic Model. However, the beta distribution is still rather inflexible in that it assumes that a topic evolution will have only a single rise and fall during the entire corpus history.

The Hall model [49] applies LDA to the entire collection of documents at the same time and performs post hoc calculations based on the observed probability of each document in order to map topics to versions. Linstead et al. [73] and Thomas et al. [125] also used this model on a software system's version history. The main advantage of this model is that no constraints are placed on the evolution of topics, providing the necessary flexibility for describing large changes to a corpus.

The Link model, proposed by Mei et al. [95] and first used on software repositories by Hindle et al. [54], applies LDA to each version of the repository *separately*, followed by a post-processing phase to link topics across versions. Once the topics are linked, the topic evolutions can be computed in the same way as in the Hall model. The post-processing phase must iteratively link topics found in one version to the topics found in the previous. This process inherently involves the use of similarity thresholds to determine whether two topics are similar enough to be called the same, since LDA is a probabilistic process and it is not guaranteed to find the exact same topics in different versions of a corpus. As a result, at each successive version, some topics are successfully linked while some topics are not, causing past topics to "die" and new topics to be "born". Additionally, it is difficult to allow for gaps in the lifetime of a topic.

2.2.8 Applying Topic Models to Source Code

Before topic models are applied to source code, several preprocessing steps are generally taken in an effort to reduce noise and improve the resulting topics.

– Characters related to the syntax of the programming language (e.g., "`&&`", "`->`") are removed; programming language keywords (e.g., "`if`", "`while`") are removed.
– Identifier names are split into multiple parts based on common naming conventions (e.g., "`oneTwo`", "`one_two`").

- Common English-language stopwords (e.g., “the”, “it”, “on”) are removed.
- Word stemming is applied to find the root of each word (e.g., “changing” becomes “chang”).
- In some cases, the vocabulary of the resulting corpus is pruned by removing words that occur in, for example, over 80% of the documents or under 2% of the documents.

The main idea behind these steps is to capture the semantics of the developers’ intentions, which are thought to be encoded within the identifier names and comments in the source code [107]. The rest of the source code (i.e., special syntax, language keywords, and stopwords) are just noise and will not be beneficial to the results of topic models.

2.2.9 Open Issues in Topic Models

Despite being actively researched for over a decade, there are still a number of important issues concerning topic models that are yet unsolved.

Choosing the Values of Input Parameters In LSI, the value of the dimensionality reduction factor used by SVM is an input to the model—large values will generate more specific, tightly-coupled topics whereas low values will generate broader, looser topics. There is currently no automated technique to infer a correct value for the dimensionality reduction factor, and instead so-called *typical values* are used (usually in the range of 25–300).

Similarly, the number of topics K to be discovered is also an input into the LDA model and its variants, along with document and topic smoothing parameters α and β. Again, there is no agreed-upon standard method for choosing the values for these input parameters a priori, although some heuristics have been proposed [46,48,130].

Evaluating Results Related to the issue of choosing the correct values for input parameters is the issue of evaluating a given set of parameters—how well do the current values work, compared to other possible values? For example, a researcher or practitioner using LDA may want to experiment with different values of K on a corpus. However, there is no general technique to automatically determine which value of K performed *better* than the others. This leaves users in a state of flux, because one is never sure which values should be used.

Although some researchers argue that model fit measures (such as the *posterior probability*, which is the likelihood of the data given the model, or the *perplexity*, which is the predictive power of the model on unseen documents) can be used to assess the effectiveness of a set of parameter values [130], it has recently been shown that these measures do not correlate with human judgment as to what corresponds to a good topic model [27]. Thus, the topic modeling community remains in a state of uncertainty.

3 Research Overview

In this section, we describe and evaluate articles that use topic models to perform software engineering tasks. Appendix A provides the details of our article selection process. We organize the articles by software engineering task. We provide a brief description of each task, followed by a chronological presentation of the relevant articles.

3.1 Concept/Feature Location

The task of *concept location* (or *feature location*) is to identify the parts (e.g., documents or methods) of the source code that implement a given feature of the software system [110]. This is useful for developers wishing to debug or enhance a given feature. For example, if the so-called *file printing* feature contained a bug, then a concept location technique would attempt to automatically find those parts of the source code that implement file printing (i.e., parts of the source code that are executed when the system prints a file).

Related to concept location is *aspect-oriented programming* (AOP), which aims at providing developers with the machinery to easily implement aspects of functionality whose implementation spans over multiple source code documents.

3.1.1 LSI-based Approaches

LSI was first used for the concept location problem by Marcus et al. (2004) [91], who developed an approach to take a developer query and return a list of related source code documents. The authors showed that LSI provides better results than existing methods (i.e., regular expressions and dependency graphs) and is easily applied to source code, due to the flexibility and light-weight nature of LSI. The authors also noted that since LSI is applied only to the comments and identifiers of the source code, it is language-independent and thus accessible for any system.

Marcus et al. (2005) [90] demonstrated that concept location is needed in the case of Object-Oriented (OO) programming languages, contrary to previous beliefs. The authors compared LSI with two other approaches, namely regular expressions and dependency graphs, for locating concepts in OO source code. The authors concluded that all techniques are beneficial and necessary, and each possesses its own strengths and weaknesses.

Poshyvanyk et al. (2006) [109] combined LSI and Scenario Based Probabilistic ranking of execution events for the task of feature location in source code. The authors demonstrated that using the two techniques, when applied together, outperform either of the techniques individually.

Poshyvanyk and Marcus (2007) [108] combined LSI and Formal Concept Analysis (FCA) to locate concepts in source code. LSI is first used to map developer queries to source code documents, then FCA is used to organize the results into a concept lattice. The authors found that this technique works well, and that concept lattices are up to four times more effective at grouping relevant information than simple ranking methods.

Cleary et al. (2008) [28] compared several IR (e.g., VSM, LSI) and NLP approaches for concept location. After an extensive experiment, the authors found that NLP approaches do not offer much of an improvement over IR techniques, which is contrary to results in other domains.

van der Spek et al. (2008) [129] used LSI to find concepts in source code. The authors considered the effects of various preprocessing steps, such as stemming, stopping, and term weighting. The authors manually evaluated the resulting concepts with the help of domain experts.

Grant et al. (2008) [47] used ICA, a conceptually similar model to LSI, to locate concepts in source code. The authors argued that since ICA is able to identify statistically independent signals in text, it can better find independent concepts in source code. The authors showed the viability of ICA to extract concepts through a case study on a small system.

Revelle and Poshyvanyk (2009) [113] used LSI, along with static and dynamic analysis, to tackle the problem of feature location. The authors combined the different techniques in novel ways. For example, textual similarity was used to traverse the static program dependency graphs, and dynamic analysis removed textually-found methods that were not executed in a scenario. The authors found that no approach outperformed all others across all case studies.

Revelle et al. (2010) [112] performed data fusion between LSI, dynamic analysis, and web mining algorithms (i.e., HITSand PageRank) to tackle the problem of feature location. The authors found that combining all three approaches significantly outperforms any of the individual methods, and outperforms the state-of-the-art in feature location.

3.1.2 LDA-based Approaches

Linstead et al. (2007) [75] were the first to use LDA to locate concepts in source code in the form of LDA topics. Their approach can be applied to individual systems or large collections of systems to extract the concepts found within the identifiers and comments in the source code. The authors demonstrated how to group related source code documents based on comparing the documents' topics.

Linstead et al. (2007) [74] applied a variant of LDA, the Author-Topic model [114], to source code to extract the relationship between developers (authors) and source code topics. Their technique allows the automated summarization of "who has worked on what", and the authors provided a brief qualitative argument as to the effectiveness of this approach.

Maskeri et al. (2008) [92] applied LDA to source code to extract the business concepts embedded in comments and identifier names. The authors applied a weighting scheme for each keyword in the system, based on where the keyword is found (e.g., class name, parameter name, method name). The authors found that their LDA-based approach is able to successfully extract business topics, implementation topics, and cross-cutting topics from source code.

Baldi et al. (2008) [9] proposed a theory that software concerns are equivalent to the latent topics found by statistical topic models. Further, they proposed that aspects are those latent topics that have a high scattering metric. The authors applied their approach to a large set of open-source projects to identify the global set of topics, as well as perform a more detailed analysis of a few specific projects. The authors found that latent topics with high scattering metrics are indeed those that are typically classified as aspects in the AOP community.

Savage et al. (2010) [118] introduced a topic visualization tool, called Topic$_{XP}$, which supports interactive exploration of discovered topics located in source code.

3.2 Traceability Recovery

An often asked question during a software development project is: "*Which source code document(s) implement requirement X?*" *Traceability recovery* aims to automatically uncover links between pairs of software artifacts, such as source code documents and requirements documents. This allows a project stakeholder to trace a requirement to its implementation, for example to ensure that it has been implemented correctly (or at all!). Traceability recovery between pairs of source code documents is also important for developers wishing to learn which source code documents are somehow related to the current source code file being worked on.

3.2.1 LSI-based Approaches

Marcus et al. (2003) [88] were the first to use LSI to recover traceability links between source code and documentation (e.g., requirements documents). The authors applied LSI to the source code identifiers and comments and the documentation, then computed similarity scores between each pair of documents. A user could then specify a similarity threshold to determine the actual links. The authors compared their work to a VSM-based recovery technique and found that LSI performs at least as good as VSM in all case studies.

De Lucia et al. (2004) [32] integrated a traceability recovery tool, based on LSI, into a software artifact management system called ADAMS. The authors presented several case studies that use their LSI-based technique to recover links between source code, test cases, and requirements documents. In subsequent work, De Lucia et al. (2006) [33] proposed an incremental approach for recovering traceability links. In this approach, a user semi-automatically interacts with the system to find an optimal similarity threshold between documents (i.e., a threshold that properly discriminates between related and unrelated documents). The authors claimed that a better threshold results in fewer links for the developer to consider, and thus fewer chances for error, making human interaction a necessity.

Hayes et al. (2006) [53] evaluated various IR techniques for generating traceability links between various high- and low-level requirements, concentrating on the tf-idf and LSI models. The authors implemented a tool called RETRO to aid a requirements analyst in this task. The authors concluded that, while not perfect, IR techniques provide value to the analyst.

Lormans et al. (2006) [81] evaluated different linking strategies (i.e., thresholding techniques) for traceability recovering using LSI by performing three case studies. The authors concluded that LSI is a promising approach for recovering links between source code and requirements documents and that different linking strategies result in different results. However, the authors observed that no linking strategy is optimal under all scenarios. In subsequent work, Lormans et al. (2007) [79] introduced a framework for managing evolving requirements (and their traceability links) in a software development cycle. Their approach uses LSI to suggest candidate links between artifacts.

Lormans et al. (2006) [80] used LSI for constructing *requirement views*, which are different views of requirements. For example, one requirement view might display only requirements that have been implemented. The authors implemented their tool, called ReqAnalyst, and used it on several real-world case studies.

De Lucia et al. (2007) [34] were the first to perform a human case study, which evaluated the effectiveness of using LSI for recovering traceability links during the software development process. The authors concluded that LSI is certainly a helpful step for developers, but that its main drawback is the inevitable trade off between precision and recall.

Jiang et al. (2008) [58] proposed an incremental approach to maintaining traceability links as a software system evolves over time. The authors' approach, called incremental LSI, uses links (and the LSI matrix) from previous versions when computing links for the current version, thus saving computation effort.

de Boer and van Vliet (2008) [31] developed a tool to support auditors in locating documentation of interest. The tool, based on LSI, suggests to the auditor documents that are related to a given query, as well as documents that are semantically related to a given document. Such a process gives the auditor, who is unfamiliar with the documentation, a guide to make it easier to explore and understand the documentation of a system.

Antoniol et al. (2008) [4] introduced a tool called Reuse or Rewrite (ReORe) to help stakeholders decide if they should update existing code (for example, to introduce new functionalities) or completely rewrite from scratch. ReORe achieves this by using a combination of static (LSI), manual, and dynamic analysis to create traceability links between existing requirements and source code. The stakeholders can then review the recovered traceability links to decide how well the current system implements the requirements.

McMillan et al. (2009) [93] used both textual (via LSI) and structural (via Evolving Inter-operation Graphs) information to recover traceability links between source code and requirements documents. The authors performed a case study on a small but well-understood system, `CoffeeMaker`. The authors demonstrated that combining textual and structural information modestly improves traceability results in most cases.

3.2.2 Comparison Studies

While the majority of researchers only evaluate their approach with respect to a single topic model, four have directly compared the performance of multiple topic models.

Lukins et al. (2008) [82, 83] used LDA to find traceability links between bug reports and source code (known as *bug localization*). The authors first build an LDA model on the source code at the method level, using the standard preprocessing steps. Then, given a bug report, the authors compute the similarity of the text content of the bug report to all source code documents. They then return the top ranked source code documents. The authors compared their approach to an LSI-based approach and showed that LDA often significantly outperforms LSI, and never under-performs.

Capobianco et al. (2009) [24] compared the ability of four different techniques (Vector Space Model, LSI, Jenson-Shannon, and B-Spline) to recover traceability links between source code, test cases, and UML diagrams. The authors found that the B-Spline method outperforms VSM and LSI, and is comparable to the Jenson-Shannon method.

Oliveto et al. (2010) [101] compared the effectiveness of four IR techniques for traceability recovery: Jenson-Shannon, VSM, LSI, and LDA. The authors showed that LDA provides unique dimensionality compared to the other four techniques.

Asuncion et al. (2010) [5] introduced a tool called TRASE that uses LDA for prospectively, as opposed to retrospectively, recovering traceability links amongst diverse artifacts in software repositories. This means that developers can create and maintain traceability links as they work on the project. The authors demonstrated that LDA outperforms LSI in terms of precision and recall.

3.3 Source Code Metrics

Bug prediction (or *defect prediction* or *fault prediction*) tries to automatically predict which parts (e.g., documents or methods) of the source code are likely to contain bugs. This task is often accomplished by collecting metrics on the source code, training a statistical model to the metrics of documents that have known bugs, and using the trained model to predict whether new documents will contain bugs.

Often, the state of the art in bug prediction is advanced either by the introduction of new metrics or by the use of a previously unexplored statistical model (e.g., Kamei et al. [60], Nguyen et al. [100], Shihab et al. [123]). An entire suite of metrics have thus far been introduced, counting somewhere in the hundreds. Additionally, dozens or hundreds of statistical models have been applied with varying degrees of success.

The majority of metrics are measured directly on the code (e.g., code complexity, number of methods per class) or on the code change process (methods that are frequently changed together, number of methods per change). However, researchers have used topic models to introduce *semantic* or conceptual metrics, which are mostly based on the comments and keywords in the source code.

3.3.1 LSI-based Metrics

Marcus et al. (2008) [89] introduced a new class cohesion metric, called the Conceptual Cohesion of Classes (C3), for measuring the cohesion of a program entity. The metric is based on the semantic information in the class, such as identifier names and comments, and is computed using LSI. Highly cohesive entities are thought to follow better design principles and are shown to correlate negatively with program faults. Bavota et al. (2010) [11] used the C3 metric in developing 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). Kagdi et al. (2010) [59] used a similar metric, the conceptual similarity between pairs of source code methods, as a part of a novel change impact analysis technique.

Gall et al. (2008) [40] extensively evaluated a suite of semantic metrics that are computed on the design and requirements documents and on the source code of a system throughout the development process. Some of the metrics are based on LSI. Through three case studies, the authors found significant correlation between metrics measured on design and requirements documents and the same metrics measured source code, providing strong evidence of the semantic similarity of these documents. The authors argued that tracking such metrics can help in the detection of problematic or suspect design decisions early in the software development process.

Ujhazi et al. (2010) [128] defined two new conceptual metrics that measure the coupling and cohesion of methods in software systems. Both metrics are based on a method's representation in an LSI subspace. The authors compared their new metrics to an existing suite of metrics (including those of Marcus et al. [89]) and found that the new metrics provide statistically significant improvements compared to previous metrics.

3.3.2 LDA-based Metrics

Linstead et al. (2009) [71] applied LDA to the bug reports in the GNOME project with the goal of measuring the coherence of a bug report, i.e., how easy to read and how focused a bug report is. This coherence metric is defined as the tangling of LDA topics within the report, i.e., how many topics are found in the report (fewer is better).

Liu et al. (2009) [77] applied LDA to source code methods in order to compute a novel class cohesion metric called Maximum Weighted Entropy (MWE). MWE is computed based on the occupancy and weight of a topic in the methods of a class. The authors demonstrated that this metric captures novel variation in models that predict software faults.

Gethers et al. (2010) [42] introduced a new coupling metric, the Relational Topic-based Coupling (RTC) metric, based on a variant of LDA called Relational Topic Models (RTM). 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 demonstrated that their proposed metric provides value because it is statistically different from existing metrics.

3.4 Statistical Debugging and Root Cause Analysis

Andrzejewski et al. (2007) [2] performed statistical debugging with the use of Delta LDA, a variant of LDA. *Statistical debugging* is the task of identifying a problematic piece of code, given a log of the execution of the code. Delta LDA is able to model two types of topics: usage topics and bug topics. Bug topics are those topics that are only found in the logs of failed executions. Hence, the authors were able to identify the pieces of code that likely caused the bugs.

Bose et al. (2008) [23] used LSI as a tool for root cause analysis (RCA), i.e., identifying the root cause of a software failure. The authors built and executed a set of test scenarios that excersized the system's methods in various sequences. Then, the authors used LSI to build a method-to-test co-occurrence matrix, which clustered tests that execute similar functionalities, helping to characterize the different manifestations of a fault.

Zawawy et al. (2010) [134] presented a framework for reducing the size and complexity of execution logs so that the manual work performed by a log analyst is reduced during RCA. The reduction is accomplished by filtering the log by performing SQL queries and LSI queries. The authors demonstrated that LSI leads to fewer false positives and higher recall during the filtering process.

3.5 Software Evolution and Trend Analysis

Analyzing and characterizing how a software system changes over time, or the *software evolution* [66] of a project, has been of interest to researchers for many years. Both *how* a software system changes (e.g., it grows rapidly every twelfth month) and *why* a software system changes (e.g., a bug fix) can help yield insights into the processes used by a specific software project as well as software development as a whole.

Linstead et al. (2008) [73] applied LDA to several versions of the source code of a project in an effort to identify the trends in the topics over time. Trends in source code histories can be measured by changes in the probability of seeing a topic at specific version. When documents pertaining to a particular topic are first added to the system, for example, the topics will experience a spike in overall probability.

In a similar effort, Thomas et al. (2010) [125] evaluated the effectiveness of topic evolution models for detecting trends in the software development process. The authors applied LDA to a series of versions of the source code and calculated the popularity of a topic over time. The authors manually verified that spikes or drops in a topic's popularity indeed coincided with developer activity mentioned in the release notes and other project documentation, providing evidence that topic evolution models provide a good summary of the software history.

Hindle et al. (2009) [54,55] applied LDA to commit log messages in order to see what topics are being worked on by developers at any given time. The authors applied LDA to the commit logs in a 30 day period, and then linked successive periods together using a topic similarity score (i.e., two topics are linked if they share 8 out of their top 10 terms). The authors found LDA to be useful in identifying developer activity trends.

Neuhaus and Zimmerman (2010) [99] used LDA to analyze the Common Vulnerabilities and Exposures (CVE) database, which archives vulnerability reports from many different sources. The authors' goal was to find the trends of each vulnerability, in order to see which are increasing and which are decreasing. The authors found that their results are mostly comparable to an earlier manual study on the same dataset.

3.6 Document Clustering

Document clustering is the task of grouping related documents together, usually to enhance program understanding or reduce a developer's searching effort [62,63]. Documents can be clustered using any of several possible attributes, including their semantic similarity or dependency graphs.

Maletic et al. (1999, 2001) [84,85] first applied LSI to cluster source code documents. The authors claimed that such a clustering can improve program comprehension during the maintenance and evolutionary phases of the software development cycle. The authors found that LSI produces useful clusters and, since LSI is automated, can be of significant value to developers.

In a similar effort, Kuhn et al. (2005, 2007) [62,63] introduced a tool named HAPAX for clustering source code documents. The authors extended the work by Maletic et al. by visualizing the resulting clusters and providing each cluster with a name based on all the words in the class, not just the class names.

Lin et al. (2006) [69] introduced a tool called Prophecy that allows developers to search the Java API for groups of related functionalities. The authors applied LSI to the Javadocs of the Java API to find similarities in their functionalities. A developer can then search the LSI index to yield a cluster of related classes.

Kuhn et al. (2008) [64,65] built a two dimensional map of a software system, where the positions of entities and distances between entities are based on their vocabularies. LSI is used to reduce the dimensionality of the document-term matrix so that similar documents can be closely aligned on the map. This *software cartography* can help developers understand the layout and relationships of their source code.

3.7 Organizing and Searching Software Repositories

Kawaguchi et al. (2006) [61] presented a tool called MUDABlue for automatically organizing large collections of open-source software systems (e.g., SourceForge and Google Code) into related groups, called software categories. MUDABlue applies LSI to the identifier names found in each software system. The authors demonstrated that MUDABlue can achieve recall and precision scores above .80, compared with manually created tags of the systems.

Tian et al. (2009) [126] developed LACT, a technique to categorize systems based on their underlying topics. This work is similar in nature to Kawaguchi et al., except this work employs LDA instead of LSI. The authors compared their approach to that of Kawaguchi et al. and concluded that the approaches are comparable in effectiveness.

Linstead et al. (2008) [70,76] introduced and used an Internet-scale repository crawler, Sourcerer, to analyze a large set of software projects. The authors applied LDA and the

Author-Topic model to extract the concepts in source code and the developer contributions in source code, respectively. The authors also defined new techniques for searching for code, based on the extracted topic model. Sourcerer can be used to analyze existing projects (i.e., view most popular identifier names and LDA topics) as well as search for modules which contain desired functionality.

Poshyvanyk and Grechanik (2009) [106] proposed an approach called S^3 for searching, selecting, and synthesizing existing applications. The approach is intended for developers wishing to find code snippets from an online repository matching their current development needs. The approach builds a dictionary of available API calls and related keywords, based on online documentation. Then, developers can search this dictionary to find related code snippets. LSI is used in conjunction with Apache Lucene to provide the search capability.

3.8 Other Tasks

Marcus and Maletic (2001) [87] were the first to detect high-level clones [12,115] of source code methods by computing the semantic similarity between pairs of methods. The authors used LSI to cluster related methods together in *concept space* (i.e., a K-dimensional representation of a document, based on the document's topic memberships), and tight clusters represents code clones. Despite low levels of precision, the authors argued that this technique is cheap and can therefore be used in conjunction with existing clone detection techniques to enhance the overall results.

Grant and Cordy (2009) [45] used ICA to detect method clones. The authors argued that since ICA can identify more distinct signals (i.e., topics) than LSI, then the conceptual space used to analyze the closeness of two methods will be of higher effectiveness. The authors performed a small case study on the Linux `kernal` package, but do not compare their results to LSI.

Ahsan et al. (2009) [1] aimed to create an automatic bug triaging system, which determines which developer should address a given bug report. The authors extracted the textual content from the titles and summaries of a project's bug reports and applied LSI to obtain a reduced term-document matrix. Then, various classifiers mapped each bug report to a developer, trained on previous bug reports and related developers. In the best case, this approach achieved 45% classification accuracy.

Bajracharya et al. (2009, 2010) [7,8] apply LDA to a usage log of a popular code search engine (Koders) to analyze the user queries over time. Their goal was to determine which topics are the most popular search topics, and whether the search engine provides users with the features that they need to identify the code they want. They found LDA to be an effective tool for such a problem.

Dit et al. (2008) [36] measured the cohesion of the content of a bug report by applying LSI to the entire set of bug reports and then calculating a similarity measure on each comment within a single bug report. The authors compared their metrics to human-generated analysis of the comments and find a high similarity.

Grant and Cordy (2010) [46] tackled the challenge of choosing the optimal number of topics to input into LDA when analyzing source code. The authors' approach varied the number of topics and used a heuristic to determine how well a particular choice is able to identify two pieces of code located in the same document. The authors concluded with general guidelines and case studies.

Wu et al. (2008) [133] tackled the challenge of building a semantic-based Web-service discovery tool. Their approach, built on LSI, allows the automatic discovery of Web services based on concepts, rather than keywords.

4 Research Trends

In this section, we identify and describe a set of attributes that allow us to characterize the research trends of the the articles just presented. We define six *facets* of related attributes, summarized in Table 1.

First and foremost, we are interested in which *topic model* was primarily used in the study—LSI, LDA or some other topic model. (Note that if an article evaluates its proposed approach, which uses topic model X, against another approach, which uses topic model Y, we only mark the article as using model X. However, if the main purpose of an article is to compare various topic models, we mark the article with all topic models considered.) Second, we are interested in the *SE activity* that was being performed. We include a range of activities to allow a fine-grained view of that literature. Third, we document the *repository* being used in the article. Fourth, we are interested in how the authors of the article *evaluated* their approach, as topic models are known to be difficult to objectively evaluate. Fifth, we are interested in how the corpus was *preprocessed*, as there are several proposed techniques. And finally, we are interested in which topic modeling *tool* was used in the article, along with which parameter values were chosen, and how they were chosen.

We processed each of the articles in our article set and assigned attribute sets to each. The results allow the articles to be summarized and compared along our six chosen facets.

The results are shown in Tables 2 and 3. Table 2 shows our first four facets: which topic model was used, which software engineering activity was being performed, which repository was used, and how the authors evaluated their approach. Table 3 shows our last two facets: which preprocessing steps were taken, and what topic modeling tools and parameters were used.

We now analyze the research trends of each facet.

Facet	Attribute	Description
Topic Model	LSI	uses LSI
	LDA	uses standard LDA
	Other	uses ICA, PLSI, or a variant of LDA
SE Activity	doc. clustering	performs a clustering of documents
	concept loc.	concept/feature location or aspect-oriented programming
	metrics	derives source code metrics (usually, but not always, for bug prediction)
	trend/evolution	analyzes/predicts source code evolution
	traceability	uncovers traceability links between pairs of artifacts (including bug localization)
	bug predict./debug	predicts bugs/faults/defects in source code, uses statistical debugging techniques, or performs root cause analysis
	org./search coll.	operates on collections of systems (search, organize, analyze)
	other	any other SE task, including bug triaging and clone detection
Repository	source code	uses source code, revision control repository, or software system repository
	email	uses email, chat logs, or forum postings
	req./design	uses requirements or design documents
	logs	uses execution logs or search engine logs
	bug reports	uses bug reports or vulnerability reports
Evaluation	statistical	uses topic modeling statistics, like log likelihood or perplexity
	task specific	uses an task specific method (e.g., classification accuracy)
	manual	performs a manual evaluation
	user study	conducts a user study
Preprocessing	identifiers	includes source code identifiers
	comments	includes source code comments
	string literals	includes string literals in source code
	tokenize	splits camelCase and under_scores
	stem	stems the terms in the corpus
	stop	performs stop word removal
	prune	removes overly common or overly rare terms from vocabulary
Tool Use	tool	name of the used topic model implementation
	K value	value chosen for the number of topics, K
	K justif.	justification given for the chosen K
	iterations	number of sampling iterations run (if LDA or LDA variant)

Table 1: The final set of attributes we collected on each article.

	topic model			activity								repository					evaluation			
	LSI	LDA	other	doc. clustering	concept loc.	metrics	trend/evolution	traceability	bug pred./debug	org./search coll.	other	source code	email	req./design	logs	bug reports	statistical	task specific	manual	user study
Ahsan, 2009 [1]	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	o	.	o	.	.
Andrzejewski, 2007 [2]	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Antoniol, 2008 [4]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	.	o	.
Asuncion, 2010 [5]	.	o	.	.	.	.	.	o	.	.	.	o	o	o	.	o	.	o	.	.
Bajracharya, 2009 [7]	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Bajracharya, 2010 [8]	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Baldi, 2008 [9]	.	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Bavota, 2010 [11]	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Bose, 2008 [23]	o	.	.	o	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Capobianco, 2009 [24]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Cleary, 2008 [28]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
de Boer, 2008 [31]	o	.	.	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	.
De Lucia, 2004 [32]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
De Lucia, 2006 [33]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
De Lucia, 2007 [34]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Dit, 2008 [36]	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	o	.	o	o	.
Gall, 2008 [40]	o	.	.	.	.	o	.	.	.	.	.	o	.	o	.	.	.	.	.	.
Gethers, 2010 [42]	.	o	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Grant, 2008 [47]	.	.	o	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Grant, 2009 [45]	.	.	o	.	.	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Grant, 2010 [46]	.	o	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.
Hayes, 2006 [53]	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	.	o	.	.
Hindle, 2009 [54]	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.
Hindle, 2010 [55]	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.
Jiang, 2008 [58]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Kagdi, 2010 [59]	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Kawaguchi, 2006 [61]	o	.	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Kuhn, 2005 [62]	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Kuhn, 2007 [63]	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Kuhn, 2008 [65]	o	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	o	.	.
Kuhn, 2010 [64]	o	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.
Lin, 2006 [69]	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	o	o	.
Linstead, 2007 [75]	.	o	.	.	o	o	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Linstead, 2007 [74]	.	o	.	.	o	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Linstead, 2008 [76]	.	o	.	o	o	o	.	.	.	o	o	o	.	.	.	.	.	.	.	.
Linstead, 2008 [70]	.	o	.	o	o	o	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Linstead, 2008 [73]	.	o	.	.	o	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.
Linstead, 2009 [71]	.	o	.	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.
Linstead, 2009 [72]	.	o	.	.	o	.	o	.	.	.	o	o	.	.	.	.	.	.	.	.
Liu, 2009 [77]	.	o	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.
Lormans, 2006 [81]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Lormans, 2006 [80]	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	.	o	.	.
Lormans, 2007 [79]	o	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.	.	o	.	.
Lukins, 2008 [82]	.	o	.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	o	.	.
Lukins, 2010 [83]	.	o	.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	o	.	.

continued on next page

continued from previous page

	topic model			activity								repository					evaluation			
	LSI	LDA	other	doc. clustering	concept loc.	metrics	trend/evolution	traceability	bug pred./debug	org./search coll.	other	source code	email	req./design	logs	bug reports	statistical	task specific	manual	user study
Maletic, 1999 [85]	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Maletic, 2001 [84]	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Marcus, 2001 [87]	o	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	.	o	.
Marcus, 2003 [88]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.
Marcus, 2004 [91]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	o	.
Marcus, 2004 [86]	o	.	.	o	o	o	.	o	.	.	.	o	.	o	.	.	.	.	.	.
Marcus, 2005 [90]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Marcus, 2008 [89]	o	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.
Maskeri, 2008 [92]	.	o	.	.	o	.	.	.	.	.	.	o	.	.	.	.	o	.	.	.
McMillan, 2009 [93]	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	.	o	o	o
Neuhaus, 2010 [99]	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.
Oliveto, 2010 [101]	o	o	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.	o	.	.
Ossher, 2009 [102]	.	o	.	.	o	.	o	.	.	o	.	o	.	.	.	.	.	.	.	.
Poshyvanyk, 2006 [109]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk, 2007 [108]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk, 2007 [107]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Poshyvanyk, 2009 [106]	o	.	.	.	o	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.
Revelle, 2009 [113]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Revelle, 2005 [112]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.
Savage, 2010 [118]	.	o	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.
Thomas, 2010 [125]	.	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	o	.
Tian, 2009 [126]	.	o	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.
Ujhazi, 2010 [128]	o	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o	.	.
van der Spek, 2008 [129]	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.
Wu, 2008 [133]	o	.	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.
Zawawy, 2010 [134]	o	.	.	.	.	.	.	.	o	.	.	.	.	.	o	.	.	o	.	.
Percentage ‘o’	62	37	3	15	31	17	14	25	8	8	6	77	1	23	11	10	1	51	14	1

Table 2: Article characterization results of facets 1–4. The attributes are described in Table 1.

	preprocessing								tools			
	identifiers	comments	strings	granularity	tokenize	stem	stop	prune	tool	*K* value	justif. of *K*	iterations
Ahsan, 2009 [1]	?	?	?	bug report	?	N	Y	Y	MATLAB	50-500	vary	-
Andrzejewski, 2007 [2]	-	-	-	?	?	?	?	?	own	?	expert	2000
Antoniol, 2008 [4]	?	?	?	method	Y	Y	Y	?	own	?	?	-
Asuncion, 2010 [5]	?	?	?	?	?	Y	Y	?	own	?	?	?
Bajracharya, 2009 [7]	-	-	-	query	Y	?	N	?	Dragon	50-500	vary	?
Bajracharya, 2010 [8]	-	-	-	query	Y	?	N	?	Dragon	50-500	vary	?
Baldi, 2008 [9]	Y	N	N	class	Y	?	Y	?	?	125	vary	?
Bavota, 2010 [11]	?	?	?	method	?	?	?	?	?	?	?	-
Bose, 2008 [23]	-	-	-	?	?	?	?	?	?	?	?	-
Capobianco, 2009 [24]	?	?	?	class	?	?	Y	?	?	?	?	-
Cleary, 2008 [28]	Y	Y	Y	?	Y	Y	Y	Y	?	300	?	-
de Boer, 2008 [31]	-	-	-	req.	N	?	Y	?	?	5	?	-
De Lucia, 2004 [32]	Y	?	?	?	Y	?	Y	?	own	?	?	-
De Lucia, 2006 [33]	?	?	?	?	?	N	?	?	own	?	?	-
De Lucia, 2007 [34]	Y	N	N	?	Y	?	Y	Y	own	?	?	-
Dit, 2008 [36]	?	?	?	bug report	Y	?	Y	?	?	300	?	-
Gall, 2008 [40]	Y	Y	?	class	?	?	?	?	own	?	?	-
Gethers, 2010 [42]	Y	Y	?	class	Y	?	?	?	lda-r	75, 125, 225	vary	?
Grant, 2008 [47]	Y	Y	Y	method	?	?	?	Y	own	10	?	-
Grant, 2009 [45]	Y	N	Y	method	?	?	?	Y	?	?	?	-
Grant, 2010 [46]	Y	N	?	method	Y	?	?	?	GibbsLDA	50-300	vary	?
Hayes, 2006 [53]	-	-	-	?	?	Y	Y	?	own	10-100	vary	-
Hindle, 2009 [54]	-	-	-	commit msg	?	?	Y	Y	lda-c	20	vary	?
Hindle, 2010 [55]	-	-	-	commit msg	?	?	?	?	?	?	?	?
Jiang, 2008 [58]	?	?	?	?	?	?	?	?	own	?	?	-
Kagdi, 2010 [59]	Y	Y	?	method	?	?	?	?	?	?	?	-
Kawaguchi, 2006 [61]	Y	N	N	system	?	?	?	Y	own	?	?	-
Kuhn, 2005 [62]	Y	Y	?	class/method	Y	Y	Y	?	?	200-500	?	-
Kuhn, 2007 [63]	Y	Y	?	class	Y	Y	Y	?	own	15	?	-
Kuhn, 2008 [65]	?	?	?	class	?	?	?	?	own	?	?	-
Kuhn, 2010 [64]	Y	Y	Y	class	?	?	?	?	own	50	?	-
Lin, 2006 [69]	N	Y	N	class	?	Y	Y	?	own	?	?	-
Linstead, 2007 [75]	?	?	?	class	Y	?	Y	?	own	100	vary	3000
Linstead, 2007 [74]	?	?	?	?	Y	?	Y	?	TMT	100	vary	3000
Linstead, 2008 [76]	?	?	?	?	Y	?	Y	?	?	100	vary	3000
Linstead, 2008 [70]	?	?	?	?	Y	?	Y	?	?	100	vary	3000
Linstead, 2008 [73]	Y	?	?	class	Y	?	Y	?	?	100	vary	?
Linstead, 2009 [71]	?	?	?	bug report	Y	?	Y	?	?	100	vary	3500
Linstead, 2009 [72]	?	?	?	?	?	?	?	?	?	?	?	?
Liu, 2009 [77]	Y	Y	?	method	Y	?	Y	?	GibbsLDA	100	?	1000
Lormans, 2006 [81]	?	?	?	?	?	Y	Y	?	TMG	?	?	-
Lormans, 2006 [80]	-	-	-	?	?	?	?	?	own	?	?	-
Lormans, 2007 [79]	-	-	-	?	?	?	?	?	own	?	?	-
Lukins, 2008 [82]	Y	Y	Y	method	Y	Y	Y	?	GibbsLDA	100	?	?
Lukins, 2010 [83]	Y	Y	?	method	N	N	N	?	GibbsLDA	100	vary	?

continued on next page

continued from previous page

	preprocessing								tools			
	identifiers	comments	strings	granularity	tokenize	stem	stop	prune	tool	K value	justif. of K	iterations
Maletic, 1999 [85]	Y	Y	Y	class/method	?	?	?	?	?	250	vary	-
Maletic, 2001 [84]	Y	Y	Y	class/method	?	?	?	?	?	350	?	-
Marcus, 2001 [87]	Y	Y	Y	class/method	?	?	?	?	own	350	?	-
Marcus, 2003 [88]	Y	Y	Y	class	Y	?	?	?	?	?	?	-
Marcus, 2004 [91]	Y	Y	N	?	Y	?	?	?	?	?	?	-
Marcus, 2004 [86]	Y	Y	?	class/method	Y	?	?	?	?	?	?	-
Marcus, 2005 [90]	Y	Y	N	?	?	?	?	?	?	?	?	-
Marcus, 2008 [89]	Y	Y	N	method	?	?	?	?	?	?	?	-
Maskeri, 2008 [92]	Y	Y	?	class	Y	Y	Y	?	own	30	?	?
McMillan, 2009 [93]	Y	?	?	method	Y	Y	Y	?	own	25-75	?	-
Neuhaus, 2010 [99]	-	-	-	report	N	Y	Y	?	?	40	?	?
Oliveto, 2010 [101]	?	?	?	?	?	Y	Y	Y	?	250	vary	?
Ossher, 2009 [102]	?	?	?	?	?	?	?	?	?	?	?	?
Poshyvanyk, 2006 [109]	Y	Y	?	method	?	?	?	?	?	?	?	-
Poshyvanyk, 2007 [108]	Y	Y	N	method	Y	N	N	N	?	?	?	-
Poshyvanyk, 2007 [107]	Y	Y	N	method	Y	N	N	N	?	500	?	-
Poshyvanyk, 2009 [106]	?	?	?	?	?	?	?	?	?	?	?	-
Revelle, 2009 [113]	Y	?	?	method	?	?	?	?	?	?	?	-
Revelle, 2005 [112]	Y	Y	Y	method	Y	Y	?	?	?	?	?	-
Savage, 2010 [118]	Y	Y	?	class	Y	Y	Y	?	JGibbLDA	input	?	input
Thomas, 2010 [125]	Y	Y	?	class	Y	Y	Y	?	MALLET	45	previous	?
Tian, 2009 [126]	Y	Y	?	system	Y	?	Y	?	GibbsLDA	40	vary	?
Ujhazi, 2010 [128]	Y	Y	N	method	Y	Y	Y	?	?	?	?	-
van der Spek, 2008 [129]	Y	Y	N	method	Y	N	Y	?	SVDLIBC	input	vary	-
Wu, 2008 [133]	-	-	-	log	Y	Y	Y	?	JAMA	?	?	-
Zawawy, 2010 [134]	-	-	-	log	?	Y	Y	?	?	?	?	-
Percentage 'Y'	54	42	14	-	48	27	49	11	-	-	-	-
Percentage 'N'	1	7	15	-	4	8	7	3	-	-	-	-
Percentage '?'	27	32	52	30	48	65	44	86	48	49	70	25

Table 3: Article characterization results of facets 5 and 6. The attributes are described in Table 1. A 'Y' means the article stated that it included this attribute or performed this step; a 'N' means the article stated that it did not include this attribute or perform this activity; a '?' means the article did not state either way; and a '-' means this attribute is not applicable to this article.

4.1 Facet 1: Topic Model

The majority of articles that we surveyed (62%) employed LSI. This majority is likely due to LSI's earlier introduction than LDA (1990 vs. 2003) as well as its relative simplicity, speed, and ease of implementation.

Still, 37% of the articles used LDA or an LDA variant, indicating that LDA is indeed a popular choice. In fact, as Figure 2 illustrates, the use of LDA is increasing rapidly since its introduction into the software engineering field in 2006.

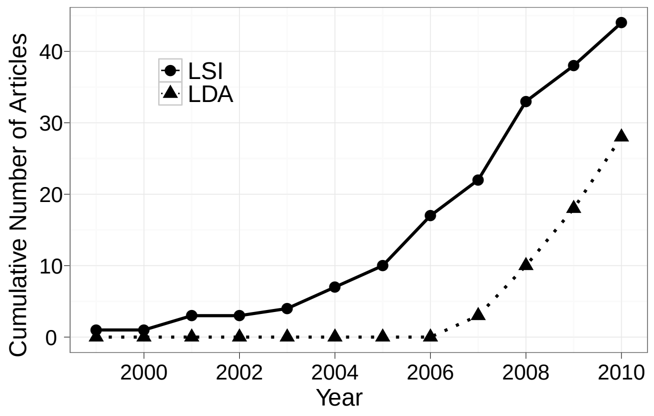


Figure 2: Trends of LSI and LDA use.

4.2 Facet 2: Software Engineering Activity

The most popular SE activities in the articles we surveyed are concept location (31% of articles) and traceability link recovery (25% of articles). Concept location is an ideal task for topic models, since many researchers (e.g., Baldi et al. [9]) believe that the topics discovered by a topic model are essentially equivalent (or can be directly mapped) to the programming concepts in the source code.

Traceability link recovering is another task well-suited for topic models, since the goal of traceability recovering is to find the textual similarity between pairs of documents. Thus, using the document similarity metrics defined on the topic membership vectors of two documents is a direct implementation of traceability link recovery.

The activities in the *"other"* category include bug triaging (Ahsan et al. (2009) [1]), search engine usage analysis (Bajracharya et al. (2009) [7], Bajracharya et al. (2010) [8]), auditor support for exploring the implementation of requirements (de Boer and van Vliet (2008) [31]), analyzing bug report quality (Dit et al. (2008) [36]), estimating the number of topics in source code (Grant and Cordy (2010) [46]), clone identification (Marcus and Maletic (2001) [87], Grant et al. [45]), finding related code on the web (Poshyvanyk et al. (2009) [106]), and web service discovery (Wu et al. (2008) [133]).

4.3 Facet 3: Repository

The overwhelming majority (77%) of the articles use the source code repository, with the second most being requirements or design documents (23%). One possible reason for the popularity of using source code is that source code is usually the only repository that is (easily) available to researchers who study open-source projects. Requirements and design documents are not created as often during such projects, and if they are, they are rarely accessible to researchers. Email archives are usually only available for large projects, and when they are available, the archives are not usually in a form that is suited for analysis without complicated preprocessing efforts. Execution logs are most useful for analyzing ultra-large scale systems under heavy load, which is difficult for researchers to simulate. Bug reports, although gaining in popularity, are typically only kept for large, well-organized projects.

4.4 Facet 4: Evaluation

The most typical evaluation method used is task-specific (51% of articles surveyed). This result makes sense, because most researchers are using topic models as a tool to accomplish some SE task. Hence, the researchers evaluate how well their approach performed at the given task, as opposed to how well the topic model fit the data, as is typically done in the text mining community.

Perhaps surprising is that 14% percent of articles performed a manual evaluation of the topic modeling results—an evaluation technique that is difficult and time consuming. This may be due to the seemingly "black-box" nature of topic models—documents are input, topics are output, and the rest is unknown. In these articles, manual evaluation is deemed to be the only way to be sure that the discovered topics make sense for software repositories.

Although articles in the IR and natural language processing domains tend to only use statistical means to evaluate topic models, only a single article we surveyed used statistical evaluation techniques (Maskeri et al. [92]). This is perhaps a result of the SE community being task-focused, as opposed to model-focused. For example, a traceability recovery technique is often evaluated on its accuracy of connecting two documents that are known to be related, as opposed to being evaluated on the model fit on the corpus, which does not directly indicate how accurate the discovered traceability links are.

4.5 Facet 5: Preprocessing

Of the articles that analyzed a source code repository, 54% mention that they include the identifiers, 42% mention that they include the comments, and 14% mention that they include string literals. This seems to follow the trend that the semantics of the developers' intention is captured by their choice of identifier names and comments (e.g., Poshyvanyk et al. [107]).

The majority of articles that analyzed source code created "documents" at the method level (33%), with a close second being the class level (25%). This means that researchers tend to consider the method as a cohesive unit of information, as opposed to an entire class. 9% of the articles left the choice of method or class as an input to their tool and reported results on both. 29% of the articles did not specify the level of granularity used.

The majority of articles that analyzed source code chose to tokenize terms (53%). One reason for this is that identifier names in source code are often written in a form that lends itself to tokenization (e.g., `camelCase` and `under_score`). By tokenizing these identifier names, the terms are being broken down into their base form and generating a larger likelihood of finding meaningful topics.

To further reduce the size of the vocabulary and increase the effectiveness of topic modeling, 27% of articles report stemming words, 49% report removing stop words, and 11% report pruning the vocabulary by removing overly- and/or underly-used terms.

4.6 Facet 6: Tool Use and Parameter Choices

For LDA-based articles, GibbsLDA [104] was the most frequently reported tool used (6 times). The only other tool that was used in more than one article was Dragon [136].

Not reporting the value of K was the norm, with 49% of articles giving no indication of their choice. Of those that did, values ranged between 5 and 500, with the most frequent values being between 100 and 200.

43% of the articles that did specify the value of K did not specify *why* that particular value was used. Of the articles that did specify why, 54% came to an optimal value by testing a range of K values and evaluating each in some way (usually task-specific). A single article used an expert's opinion on what the number of topics should be (*"The number of topics for each program are chosen according to domain expert advice."* [2]) (although the actual value was not reported!).

The reporting of other input parameters, such as α, β, and the number of sampling iterations (in the case of LDA) was even more scarce. 77% of the articles that used LDA did not indicate the number of iterations sampled. Of those that did, values ranged between 1000 and 3500, with 3000 being the norm.

5 Repeatability

Our findings indicate a need for improvement in the software engineering community in communicating key study design decisions. For instance,49% of the articles we studied did not report the value of K (topics or reduction factor) used in the topic model, even though it is well known that this choice greatly effects the output of the topic model, and thus the study being performed.

In addition, many articles were unclear as to how they preprocessed the textual data, even though this step also has a large impact on the results of topic modeling. For example, 30% of the articles we surveyed did not mention the document granularity of their approach (e.g., an entire class or an individual method) and 65% of the articles did not indicate whether they used word stemming, a common preprocessing step in the NLP community.

More generally, each of the attributes we collected impact the results of a topic model, and thus should be included in the article for purposes of replicability.

6 Avenues for Future Research

In this section we outline opportunities for future research.

Underused Repositories Several software repositories have been rarely studied, despite containing rich sources of textual data: email archives have only been studied by a single article and bug reports have only been studied by 10% of the articles.

Underperformed SE Tasks Bug prediction, searching collections of software systems, and measuring the evolutionary trends of repositories are all under-represented tasks in the studied articles. Traceability links are typically established between requirements documents and source code, although it would be also be useful to find links between emails and source code, and between source code documents themselves.

LDA Variants The variants of LDA listed in Section 2.2.6 have promising features that may directly improve the results of several SE activities. For example, the correlated topic model, which models dependencies between topics, may allow sets of dependent topics in need of refactoring to be found in the source code. Additionally, the cross-collection topic model might allow similar topics to be discovered from the source code of related systems, such as Mozilla Firefox and Google Chrome.

Preprocessing Steps There is currently little consensus in the community as to which preprocessing steps are necessary to achieve optimal results with topic models. It would be useful to experimentally determine which steps have the largest impact on the results.

Comparing Topic Models Very few articles directly compared the results of multiple topic models, even though there are currently claimed-advantages to both. It would be useful to experimentally determine which topic model works best for a particular software repository, and under which conditions.

7 Conclusions

In this report, we have surveyed the past decade of software engineering literature to determine how topic models have helped researchers in their software repository mining efforts. We analyzed 71 highly-relevant articles and collected dozens of attributes on each. After presenting a critical summary of each article, we broadly characterized the articles according to our six facets of interest: which topic model was used, which software engineering activity was performed, which software repository was mined, how the results were analyzed, which preprocessing steps were performed, and which tools and parameter values were employed.

Our results summarize the literature by software engineering task and along the six facets, allowing interested readers to quickly find articles of interest. We have characterized which attributes are most and least popular in each facet and offered possible explanations. Finally, we have presented future research opportunities in this area.

References

[1] S. N. Ahsan, J. Ferzund, and F. Wotawa. Automatic software bug triage system (BTS) based on Latent Semantic Indexing and Support Vector Machine. In *Proceedings of the 4th International Conference on Software Engineering Advances*, pages 216–221, 2009.

[2] D. Andrzejewski, A. Mulhern, B. Liblit, and X. Zhu. Statistical debugging using latent topic models. In *Proceedings of the 18th European Conference on Machine Learning*, pages 6–17, 2007.

[3] G. Anthes. Topic models vs. unstructured data. *Communications of the ACM*, 53:16–18, Dec. 2010.

[4] G. Antoniol, J. H. Hayes, Y. G. Gueheneuc, and M. Di Penta. Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date. In *Proceedings of the 24th International Conference on Software Maintenance*, pages 147–156, 2008.

[5] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 95–104, 2010.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval*, volume 463. ACM press New York, 1999.

[7] S. Bajracharya and C. Lopes. Mining search topics from a code search engine usage log. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 111–120, 2009.

[8] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, pages 1–43, Sept. 2010.

[9] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. *ACM SIGPLAN Notices*, 43(10):543–562, 2008.

[10] D. J. Bartholomew. *Latent variable models and factors analysis*. Oxford University Press, Inc. New York, NY, USA, 1987.

[11] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *Proceedings of the 25th International Conference on Automated Software Engineering*, pages 151–154, 2010.

[12] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, pages 577–591, 2007.

[13] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX Winter 1990 Technical Conference*, volume 341, page 352, 1990.

[14] N. Bettenburg, E. Shihab, and A. E. Hassan. An empirical study on the risks of using off-the-shelf techniques for processing mailing list data. In *Proceedings of the 25th International Conference on Software Maintenance*, 2009.

[15] C. M. Bishop. Latent variable models. *Learning in graphical models*, 1998.

[16] D. Blei, T. L. Griffiths, M. I. Jordan, and J. B. Tenenbaum. Hierarchical topic models and the nested Chinese restaurant process. *Advances in neural information processing systems*, 16:106, 2004.

[17] D. M. Blei, T. L. Griffiths, and M. I. Jordan. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *Journal of the ACM*, 57(2):1–30, 2010.

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

[19] D. M. Blei and J. D. Lafferty. A correlated topic model of science. *Annals of Applied Statistics*, 1(1):17–35, June 2007.

[20] D. M. Blei and J. D. Lafferty. Topic models. In *Text Mining: Classification, Clustering, and Applications*, pages 71–94. Chapman & Hall, London, UK, 2009.

[21] D. M. Blei and J. McAuliffe. Supervised topic models. *Advances in Neural Information Processing Systems*, 20:121–128, 2008.

[22] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[23] J. C. Bose and U. Suresh. Root cause analysis using sequence alignment and Latent Semantic Indexing. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 367–376, 2008.

[24] G. Capobianco, A. De Lucia, R. Oliveto, A. Panichella, and S. Panichella. Traceability recovery using numerical analysis. In *Proceedings of the 16th Working Conference on Reverse Engineering*, pages 195–204, 2009.

[25] J. Chang and D. M. Blei. Relational topic models for document networks. *Proceedings of the 12th International Conference on Artificial Intelligence and Statistics*, 9:81–88, 2009.

[26] J. Chang, J. Boyd-Graber, and D. M. Blei. Connections between the lines: Augmenting social networks with text. In *Proceedings of the 15th International Conference on Knowledge Discovery and Data Mining*, pages 169–178, 2009.

[27] J. Chang, J. Boyd-Graber, S. Gerrish, C. Wang, and D. M. Blei. Reading tea leaves: How humans interpret topic models. In *Neural Information Processing Systems*, 2009.

[28] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2008.

[29] P. Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[30] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, pages 684–702, 2009.

[31] R. C. de Boer and H. van Vliet. Architectural knowledge discovery with Latent Semantic Analysis: constructing a reading guide for software product audits. *Journal of Systems and Software*, 81(9):1456–1469, 2008.

[32] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 306–315, 2004.

[33] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Can information retrieval techniques effectively support traceability link recovery? In *Proceedings of the 14th International Conference on Program Comprehension*, pages 307–316, 2006.

[34] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4), 2007.

[35] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[36] B. Dit, D. Poshyvanyk, and A. Marcus. Measuring the semantic similarity of comments in bug reports. In *Proceedings 1st International Workshop on Semantic Technologies in System Maintenance*, 2008.

[37] Edgewall Software. The Trac project. http://trac.edgewall.org/, 2012.

[38] P. Flaherty, G. Giaever, J. Kumm, M. I. Jordan, and A. P. Arkin. A latent variable model for chemogenomic profiling. *Bioinformatics*, 21(15):3286, 2005.

[39] M. Fowler and K. Scott. *UML distilled: A brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

[40] C. S. Gall, S. Lukins, L. Etzkorn, S. Gholston, P. Farrington, D. Utley, J. Fortune, and S. Virani. Semantic software metrics computed from natural language design specifications. *Software, IET*, 2(1):17–26, 2008.

[41] Geeknet. SourceForge. http://sourceforge.net/, 2010.

[42] M. Gethers and 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*, pages 1–10, 2010.

[43] M. W. Godfrey, A. E. Hassan, J. Herbsleb, G. C. Murphy, M. Robillard, P. Devanbu, A. Mockus, D. E. Perry, and D. Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67–70, 2008.

[44] Google. Google code. http://code.google.com/, 2012.

[45] S. Grant and J. R. Cordy. Vector space analysis of software clones. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 233–237, 2009.

[46] S. Grant and J. R. Cordy. Estimating the optimal number of latent concepts in source code analysis. In *Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation*, pages 65–74, 2010.

[47] S. Grant, J. R. Cordy, and D. Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 138–142, 2008.

[48] T. L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[49] D. Hall, D. Jurafsky, and C. D. Manning. Studying the history of ideas using topic models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 363–371. ACL, 2008.

[50] A. E. Hassan. *Mining Software Repositories to Assist Developers and Support Managers.* PhD thesis, University of Waterloo, Waterloo, ON, Canada, 2004.

[51] A. E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance*, pages 48–57, 2008.

[52] A. E. Hassan and R. C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, pages 263–272, 2005.

[53] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, pages 4–19, 2006.

[54] A. Hindle, M. W. Godfrey, and R. C. Holt. What's hot and what's not: Windowed developer topic analysis. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 339–348, 2009.

[55] A. Hindle, M. W. Godfrey, and R. C. Holt. Software process recovery using recovered unified process views. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–10, 2010.

[56] T. Hofmann. Probabilistic Latent Semantic Indexing. In *Proceedings of the 22nd International Conference on Research and Development in Information Retrieval*, pages 50–57, 1999.

[57] T. Hofmann. Unsupervised learning by probabilistic Latent Semantic Analysis. *Machine Learning*, 42(1):177–196, 2001.

[58] H. Jiang, T. N. Nguyen, I. Chen, H. Jaygarl, and C. Chang. Incremental Latent Semantic Indexing for automatic traceability link evolution management. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 59–68, 2008.

[59] H. Kagdi, M. Gethers, D. Poshyvanyk, and 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*, pages 119–128, 2010.

[60] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–10, 2010.

[61] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006.

[62] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142, 2005.

[63] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[64] A. Kuhn, D. Erni, P. Loretan, and O. Nierstrasz. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 191–210, 2010.

[65] A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 209–218, 2008.

[66] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[67] T. C. Lethbridge, R. Laganiere, and C. King. *Object-oriented software engineering: practical software development using UML and Java.* McGraw-Hill, 2005.

[68] W. Li and A. McCallum. Pachinko allocation: DAG-structured mixture models of topic correlations. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 577–584, 2006.

[69] M. Y. Lin, R. Amor, and E. Tempero. A Java reuse repository for Eclipse using LSI. In *Proceedings of the 2006 Australian Software Engineering Conference*, pages 351–362, 2006.

[70] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18(2):300–336, 2008.

[71] E. Linstead and P. Baldi. Mining the coherence of GNOME bug reports with statistical topic models. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 99–102, 2009.

[72] E. Linstead, L. Hughes, C. Lopes, and P. Baldi. Software analysis with unsupervised topic models. In *NIPS Workshop on Application of Topic Models: Text and Beyond*, 2009.

[73] E. Linstead, C. Lopes, and P. Baldi. An application of latent Dirichlet allocation to analyzing software evolution. In *Proceedings of the 7th International Conference on Machine Learning and Applications*, pages 813–818, 2008.

[74] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 461–464, 2007.

[75] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining Eclipse developer contributions via author-topic models. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, pages 30–33, 2007.

[76] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining internet-scale software repositories. In *Advances in Neural Information Processing Systems*, volume 2007, pages 929–936, 2008.

[77] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimothy, and N. Chrisochoides. Modeling class cohesion as mixtures of latent topics. In *Proceedings of the 25th International Conference on Software Maintenance*, pages 233–242, 2009.

[78] J. C. Loehlin. *Latent variable models*. Erlbaum Hillsdale, NJ, 1987.

[79] M. Lormans. Monitoring requirements evolution using views. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 349–352, 2007.

[80] M. Lormans, H. G. Gross, A. van Deursen, and R. van Solingen. Monitoring requirements coverage using reconstructed views: An industrial case study. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 275–284, 2006.

[81] M. Lormans and A. Van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Proceedings of 10th European Conference on Software Maintenance and Reengineering*, pages 47–56, 2006.

[82] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent Dirichlet allocation. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 155–164, 2008.

[83] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[84] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, 2001.

[85] J. I. Maletic and N. Valluri. Automatic software clustering via Latent Semantic Analysis. In *Proceeding of the 14th International Conference on Automated Software Engineering*, pages 251–254, 1999.

[86] A. Marcus. Semantic driven program analysis. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 469–473, 2004.

[87] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering*, pages 107–114, 2001.

[88] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using Latent Semantic Indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, 2003.

[89] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[90] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42, 2005.

[91] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, 2004.

[92] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent Dirichlet allocation. In *Proceedings of the 1st conference on India software engineering conference*, pages 113–120, 2008.

[93] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, 2009.

[94] Q. Mei, D. Cai, D. Zhang, and C. X. Zhai. Topic modeling with network regularization. In *Proceeding of the 17th international conference on World Wide Web*, pages 101–110, 2008.

[95] Q. Mei and 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*, pages 198–207, 2005.

[96] G. A. Miller. WordNet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[97] D. Mimno, H. M. Wallach, J. Naradowsky, D. A. Smith, and A. McCallum. Polylingual topic models. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 880–889, 2009.

[98] Mozilla Foundation. Bugzilla. http://www.bugzilla.org/, 2010.

[99] S. Neuhaus and T. Zimmermann. Security trend analysis with CVE topic models. In *Proceedings of the 21st International Symposium on Software Reliability Engineering*, pages 111–120, 2010.

[100] T. H. Nguyen, B. Adams, and A. E. Hassan. A case study of bias in Bug-Fix datasets. In *Proceedings of the 17th Working Conference on Reverse Engineering*, pages 259–268, 2010.

[101] R. Oliveto, M. Gethers, D. Poshyvanyk, and 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*, pages 68–71, 2010.

[102] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects. In *Proceedings of the 6th Working Conference on Mining Software Repositories*, pages 183–186, 2009.

[103] M. Paul. *Cross-Collection Topic Models: Automatically Comparing and Contrasting Text*. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 2009.

[104] X. H. Phan, L. M. Nguyen, and S. Horiguchi. Learning to classify short and sparse text & web with hidden topics from large-scale data collections. In *Proceeding of the 17th International Conference on World Wide Web*, pages 91–100, 2008.

[105] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008.

[106] D. Poshyvanyk and M. Grechanik. Creating and evolving software by searching, selecting and synthesizing relevant source code. In *Proceedings of the 31st International Conference on Software Engineering*, pages 283–286, 2009.

[107] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, pages 420–432, 2007.

[108] D. Poshyvanyk and 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*, pages 37–48, 2007.

[109] D. Poshyvanyk, A. Marcus, V. Rajlich, et al. Combining probabilistic ranking and Latent Semantic Indexing for feature identification. In *Proceedings of the 14th International Conference on Program Comprehension*, pages 137–148, 2006.

[110] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278, 2002.

[111] D. Ramage, D. Hall, R. Nallapati, and C. D. Manning. Labeled LDA: a supervised topic model for credit attribution in multi-labeled corpora. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 248–256, 2009.

[112] M. Revelle, B. Dit, and D. Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Proceedings of the 18th International Conference on Program Comprehension*, pages 14–23, 2010.

[113] M. Revelle and D. Poshyvanyk. An exploratory study on assessing feature location techniques. In *Proceedings of the 17th International Conference on Program Comprehension*, pages 218–222, 2009.

[114] M. Rosen-Zvi, T. Griffiths, M. Steyvers, and P. Smyth. The author-topic model for authors and documents. In *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 487–494, 2004.

[115] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[116] G. Salton and M. J. McGill. Introduction to modern information retrieval. *New York*, 1983.

[117] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):620, 1975.

[118] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk. TopicXP: exploring topics in source code using latent dirichlet allocation. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–6, 2010.

[119] N. Serrano and I. Ciordia. Bugzilla, ITracker, and other bug trackers. *Software, IEEE*, 22(2):11–13, 2005.

[120] E. Shihab, N. Bettenburg, B. Adams, and A. Hassan. On the central role of mailing lists in open source projects: An exploratory study. *New Frontiers in Artificial Intelligence*, 6284:91–103, 2010.

[121] E. Shihab, Z. M. Jiang, and A. E. Hassan. On the use of IRC channels by developers of the GNOME GTK+ open source project. In *Proceedings of the 6th IEEE Working Conference on Mining Software Repositories*, 2009.

[122] E. Shihab, Z. M. Jiang, and A. E. Hassan. Studying the use of developer irc meetings in open source projects. In *Proceedings of the 25th International Conference on Software Maintenance*, 2009.

[123] E. Shihab, Z. M. Jiang, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: a case study on the eclipse project. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2010.

[124] M. Steyvers and T. Griffiths. Probabilistic topic models. In *Latent Semantic Analysis: A Road to Meaning*. Laurence Erlbaum, 2007.

[125] S. W. Thomas, B. Adams, A. E. Hassan, and 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*, pages 55–64, 2010.

[126] K. Tian, M. Revelle, and D. Poshyvanyk. Using latent Dirichlet allocation for automatic categorization of software. In *Proceedings of the 6th International Working Conference on Mining Software Repositories*, pages 163–166, 2009.

[127] W. Tichy. An interview with Prof. Andreas Zeller: Mining your way to software reliability. *Ubiquity*, 2010, Apr. 2010.

[128] B. Ujhazi, R. Ferenc, D. Poshyvanyk, and 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*, pages 33–42, 2010.

[129] P. van der Spek, S. Klusener, and P. van de Laar. Towards recovering architectural concepts using Latent Semantic Indexing. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 253–257, 2008.

[130] H. M. Wallach, I. Murray, R. Salakhutdinov, and D. Mimno. Evaluation methods for topic models. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1105–1112, 2009.

[131] C. Wang, B. Thiesson, C. Meek, and D. Blei. Markov topic models. In *The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 583–590, 2009.

[132] X. Wang and 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*, pages 424–433. ACM, 2006.

[133] C. Wu, E. Chang, and A. Aitken. An empirical approach for semantic web services discovery. In *Proceedings of the 19th Australian Conference on Software Engineering*, pages 412–421, 2008.

[134] H. Zawawy, K. Kontogiannis, and J. Mylopoulos. Log filtering and interpretation for root cause analysis. In *Proceedings of the 26th International Conference on Software Maintenance*, pages 1–5, 2010.

[135] C. X. Zhai. Statistical language models for information retrieval. *Synthesis Lectures on Human Language Technologies*, 1(1):1–141, 2008.

[136] X. Zhou, X. Zhang, and X. Hu. Dragon toolkit: Incorporating auto-learned semantic knowledge into large-scale text retrieval and mining. In *Proceedings of the 19th International Conference on Tools with Artificial Intelligence*, volume 2, pages 197–201, 2007.

[137] T. Zimmermann et al. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, pages 429–445, 2005.

A Article Selection

We focus our attention to articles written between January 2000 and November 2010, almost a full decade of research results, starting with the first use of topic models in software engineering research. We consider in our search a wide range of journals and conferences in order to recover as many relevant articles as possible. Similar to previous surveys (e.g., Cornelissen et al. [30]), we perform a *systematic review* of the literature, that is, we describe our methodology for selecting the set of articles used in the analysis of the literature.

To select our list of articles, we first compile a list of highly relevant venues to search. Then, we perform a series of keyword searches at each venue, producing a list of candidate articles. For each of the candidate articles, we read the abstract (and, in some cases, the introduction) of the article to determine if the article is indeed relevant to our interests. This yields an *initial set* of related articles. Finally, for each of the articles in the initial set, we consider the citations contained in the article for additional relevant articles. We finally arrive at our *final set* of articles, which we read and characterize in detail.

A.1 Venues Considered

Table 4 lists the journals and conference venues that we included in our initial search for articles. We consider this list of six journals and nine conferences to be the most closely related venues for software engineering, empirical software engineering, program comprehension, and mining software repositories.

A.2 Keyword Searches and Filtering

We collected the initial set of articles by performing keyword searches at the publisher websites for each of our considered venues. The keywords we used for our searches are

Type	Acronym	Description
Journal	ESE	Empirical Software Engineering
	JSME	Journal of Software Maintenance and Evolution
	JSS	Journal of Systems and Software
	SP&E	Software – Practice & Experience
	TSE	IEEE Transactions on Software Engineering
	TOSEM	ACM Transactions on Software Engineering & Methodology
Conference	ASE	International Conference on Automated Software Engineering
	ESEC/FSE	European Software Engineering Conference / Symposium on the Foundations of Software Engineering
	FASE	International Conference on Fundamental Approaches to Software Engineering
	ICSE	International Conference on Software Engineering
	ICSM	International Conference on Software Maintenance
	IWPC/ICPC	International Workshop/Conference on Program Comprehension
	MSR	International Workshop/Working Conference on Mining Software Repositories
	SCAM	International Workshop/Working Conference on Source Code Analysis and Manipulation
	WCRE	Working Conference on Reverse Engineering

Table 4: The fourteen venues that we considered in our initial article selection process.

listed in Appendix B. We also searched using aggregate search engines, such as ACM Digital Library and IEEE Xplore.

In general, we found that keyword searches resulted in many irrelevant results. We manually filtered the search results by reading an article's abstract (and sometimes introduction) to determine if the article solved an software engineering task by applying one or more topic modeling techniques. The articles that were determined to be relevant were added to our initial set.

A.3 Reference Checking

For each article in the initial set, we followed its citations to obtain another list of potentially relevant articles. (We did not recursively follow the citations of these potentially relevant articles.) Again, we filtered this list by reading the abstracts and introductions. The articles that we determined to be relevant were added to our final set of articles.

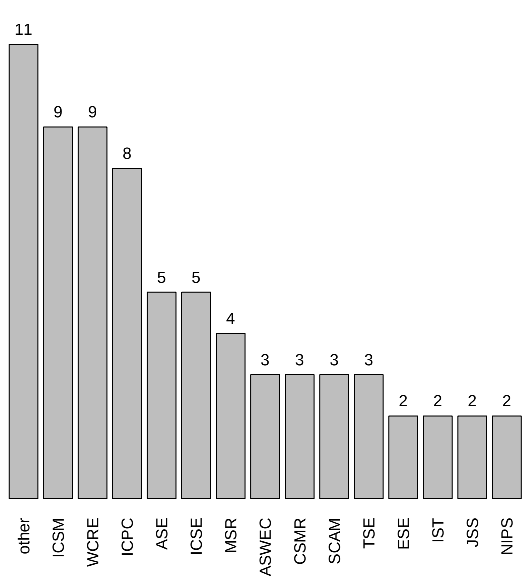
A.4 Article Selection Results

We finally arrive at 69 articles published between 1999 and 2010. Figure 3 shows the distribution of venues and years for the articles.

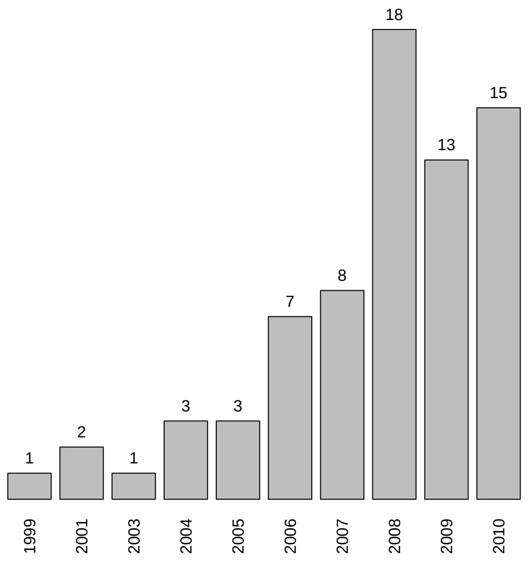
B Search Queries Used

IEEE Xplore

```
("topic models" OR "topic model"
OR "lsi" OR "lda" OR "plsi"
```



(a) Venues of articles



(b) Years of articles

Figure 3: Article selection results.

```
OR "latent dirichlet allocation" OR "latent semantic")
AND
(  "Publication Title":"Source Code Analysis and Manipulation"
OR "Publication Title":"Software Engineering, IEEE Transactions on"
OR "Publication Title":"Reverse Engineering"
OR "Publication Title":"Software Maintenance"
OR "Publication Title":"Software Engineering"
OR "Publication Title":"Program Comprehension"
OR "Publication Title":"Mining Software Repositories")
```

Software Practice and Experience

```
lsi or lda or "topic model" or "topic models" or
"latent dirichlet allocation" or "latent semantic"
AND publication title="Software Practice and Experience"
```

Journal of Software Maintenance and Evolution

```
lsi or lda or "topic model" or "topic models" or
latent dirichlet allocation" or "latent semantic"
AND publication title="Software Maintenance and Evolution"
```