

Chapter 5

Mining Unstructured Software Repositories

Stephen W. Thomas, Ahmed E. Hassan and Dorothea Blostein

Summary. Mining software repositories, which is the process of analyzing the data related to software development practices, is an emerging field of research which aims to improve software evolutionary tasks. The data in many software repositories is unstructured (for example, the natural language text in bug reports), making it particularly difficult to mine and analyze. In this chapter, we survey tools and techniques for mining unstructured software repositories, with a focus on information retrieval models. In addition, we discuss several software engineering tasks that can be enhanced by leveraging unstructured data, including bug prediction, clone detection, bug triage, feature location, code search engines, traceability link recovery, evolution and trend analysis, bug localization, and more. Finally, we provide a hands-on tutorial for using an IR model on an unstructured repository to perform a software engineering task.

5.1 Introduction

Researchers in software engineering have attempted to improve software development by mining and analyzing software repositories, such as source code changes, email archives, bug databases, and execution logs [329, 371]. Research shows that interesting and practical results can be obtained from mining these repositories, allowing developers and managers to better understand their systems and ultimately increase the quality of their products in a cost effective manner [847]. Particular success has been experienced with *structured* repositories, such as source code, execution traces, and change logs.

Software repositories also contain *unstructured* data, such as the natural language text in bug reports, mailing list archives, requirements documents, and source code comments and identifier names. In fact, some researchers estimate that between 80% and 85% of the data in software repositories is unstructured [118, 351].

Unstructured data presents many challenges because the data is often unlabeled, vague, and noisy [371]. For example, the Eclipse bug database contains the following bug report titles:

- “NPE caused by no spashscreen handler service available” (#112600)
- “Provide unittests for link creation constraints” (#118800)
- “jaxws unit tests fail in standalone build” (#300951)

This data is *unlabeled* and *vague* because it contains no explicit links to the source code entity to which it refers, or even to a topic or task from some pre-defined ontology. Phrases such as “link creation constraints,” with no additional information or pointers, are ambiguous at best. The data is *noisy* due to misspellings and typographical errors (“spashscreen”), unconventional acronyms (“NPE”), and multiple phrases used for the same concept (“unittests”, “unit tests”). The sheer size of a typical unstructured repository (for example, Eclipse receives an average of 115 new bug reports per day), coupled with its lack of structure, makes manual analysis extremely challenging and in many cases impossible. Thus, there is a real need for automated or semi-automated support for analyzing unstructured data.

Over the last decade, researchers in software engineering have developed many tools and techniques for handling unstructured data, often borrowing from the natural language processing and information retrieval communities. In fact, this problem has led to the creation of many new academic workshops and conferences, including NaturaLiSE (International Workshop on Natural Language Analysis in Software Engineering), TEFSE (International Workshop on Traceability in Emerging Forms of Software Engineering), and MUD (Mining Unstructured Data). In addition, premier venues such as ICSE (International Conference on Software Engineering), FSE (Foundations of Software Engineering), ICSM (International Conference on Software Maintenance), and MSR (Working Conference on Mining Software Repositories), have shown increasing interest in techniques for mining unstructured software repositories.

In this chapter, we examine how to best use unstructured software repositories to improve software evolution. We first introduce and describe common unstructured

software repositories in Section 5.2. Next, we present common tools and techniques for handling unstructured data in Section 5.3. We then explore the state-of-the-art in software engineering research for combining the tools and unstructured data to perform some meaningful software engineering tasks in Section 5.4. To make our presentation concrete, we present a hands-on tutorial for using an advanced IR model to perform the task of bug localization in Section 5.5. We offer concluding remarks in Section 5.6.

5.2 Unstructured Software Repositories

The term “unstructured data” is difficult to define and its usage varies in the literature [105, 557]. For the purposes of this chapter, we adopt the definition given by Manning [557]:

“Unstructured data is data which does not have clear, semantically overt, easy-for-a-computer structure. It is the opposite of structured data, the canonical example of which is a relational database, of the sort companies usually use to maintain product inventories and personnel records.”

Unstructured data usually refers to natural language text, since such text has no explicit data model. Most natural language text indeed has latent structure, such as parts-of-speech, named entities, relationships between words, and word sense, that can be inferred by humans or advanced machine learning algorithms. However, in its raw, unparsed form, the text is simply a collection of characters with no structure and no meaning to a data mining algorithm.

Structured data, on the other hand, has a data model and a known form. Examples of structured data in software repository include: source code parse trees, call graphs, inheritance graphs; execution logs and traces; bug report metadata (e.g., author, severity, date); source control database commit metadata (e.g., author, date, list of changed files); and mailing list and chat log metadata (e.g., author, date, recipient list).

We now describe in some detail the most popular types of unstructured software repositories. These repositories contain a vast array of information about different facets of software development, from human communication to source code evolution.

5.2.1 Source Code

Source code is the executable specification of a software system’s behavior [514]. The source code repository consists of a number of *documents* or *files* written in one or more programming languages. Source code documents are generally grouped into logical entities called *packages* or *modules*.

While source code contains structured data (e.g., syntax, program semantics, control flow), it also contains rich unstructured data, collectively known as its *linguistic data*:

- Comments in the form of developer messages and descriptions and Javadoc-type comments.
- Identifier names, including class names, method names, and local and global variable names.
- String literals found in print commands and functions.

This unstructured portion of source code, even without the aid of the structured portion, has been shown to help determine the high-level functionality of the source code [482].

5.2.2 Bug Databases

A *bug database* (or *bug-tracking system*) maintains information about the creation and resolution of bugs, feature enhancements, and other software maintenance tasks [770]. 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 a *bug report* (or *issue*), which includes such information as what task they were performing when the bug occurred, how to reproduce the bug, and how critical the bug is to the functionality of the system. Then, one or more maintainers of the system investigate the bug report, and if they resolve the issue, they close the bug report. All of these tasks are captured in the bug database. Popular bug database systems include Bugzilla¹ and Trac², although many exist [770].

The main unstructured data of interest in a bug report is:

- *title* (or *short description*): a short message summarizing the contents of the bug report, written by the creator of the bug report;
- *description* (or *long description*): a longer message describing the details about the bug;
- *comments*: short messages left by other users and developers about the bug

5.2.3 Mailing Lists and Chat Logs

Mailing lists (or *discussion archives*), along with the *chat logs* (or *chat archives*) are archives of the textual communication between developers, managers, and other project stakeholders [775]. The mailing list is usually comprised of a set of time-stamped email messages, which contain a *header* (containing the sender, receiver(s),

¹ www.bugzilla.org

² trac.edgewall.org

and time stamp), a *message body* (containing the unstructured textual 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 message bodies [103, 776, 777]. The main unstructured data of interest here are the message bodies.

5.2.4 Revision Control System

A *revision control system* maintains and records the history of changes (or edits) to a repository of documents. Developers typically use revision control systems to maintain the edits to source code. Most revision control systems (including CVS [95], Subversion (SVN) [685]), and Git [110]) allow developers to enter a *commit message* when they commit a change into the repository, describing the change at a high level. These unstructured commit messages are of interest to researchers because taken at an aggregate level, they describe how the source code is evolving over time.

5.2.5 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 [514]. The requirements can be categorized as either *functional*, which specify the “*what*” of the behavior of the program, or *non-functional*, which describe the qualities of the software (e.g., reliability or accessibility). Most often, requirements documents take the form of natural language text.

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, such as UML diagrams [303], or natural language text.

5.2.6 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 third parties. Popular repositories include SourceForge³ and Google Code⁴. Each software

³ sourceforge.net

⁴ code.google.com

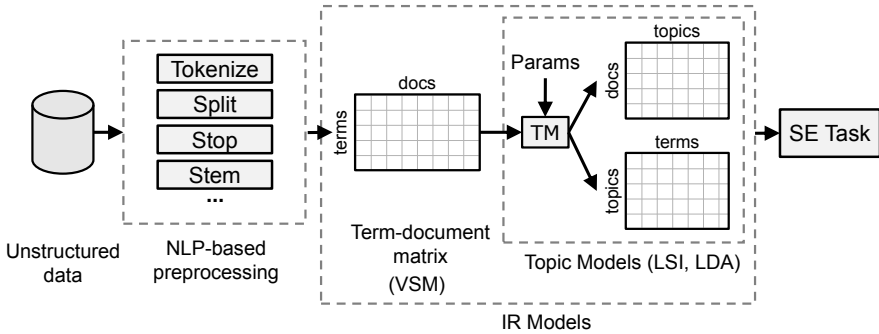


Fig. 5.1: The process of mining unstructured software repositories. First, the unstructured data is preprocessed using one or more NLP techniques. (See Figure 5.2 for an example.) Then, various IR models are built. Finally, the software engineering (SE) task can be performed.

system in the repository may contain any of the above unstructured repositories: source code, bug databases, mailing lists, revision control databases, requirements documents, or design documents.

5.3 Tools and Techniques for Mining Unstructured Data

To help combat the difficulties inherent to unstructured software repositories, researchers have used and developed many tools and techniques. No matter the software repository in question, the typical technique follows the process depicted in Figure 5.1. First, the data is preprocessed using one or more Natural Language Processing (NLP) techniques. Next, an information retrieval (IR) model or other text mining technique is applied to the preprocessed data, allowing the software engineering task to be performed. In this section, we outline techniques for preprocessing data, and introduce common IR models.

5.3.1 NLP Techniques for Data Preprocessing

Preprocessing unstructured data plays an important role in the analysis process. Left unprocessed, the noise inherent to the data will confuse and distract IR models and other text mining techniques. As such, researchers typically use NLP techniques to perform one or more preprocessing steps before applying IR models to the data. We describe *general* preprocessing steps that can be applied to any source of unstruc-

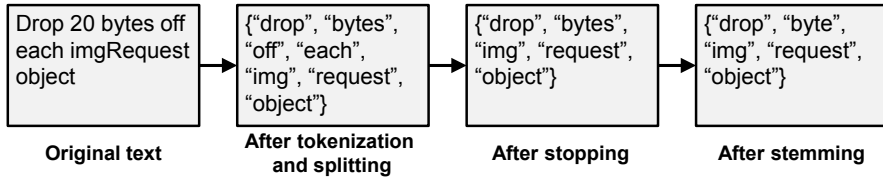


Fig. 5.2: An illustration of common NLP preprocessing steps. All steps are optional, and other steps exist.

tured data, and then outline more specific preprocessing steps for source code and email.

5.3.1.1 General Preprocessing Steps

Several general preprocessing steps can be taken in an effort to reduce noise and improve the resulting text models built on the input corpus. These steps are depicted in Figure 5.2.

- **Tokenization.** The original stream of text is turned into tokens. During this step, punctuation and numeric characters are removed.
- **Identifier splitting.** Identifier names (if present) are split into multiple parts based on common naming conventions, such as camel case (`oneTwo`), underscores (`one_two`), dot separators (`one.two`), and capitalization changes (`ONETwo`). We note that identifier splitting is an active area of research, with new approaches being proposed based on speech recognition [551], automatic expansion [501], mining source code [274], and more.
- **Stop word removal.** Common English-language stop words (e.g., “the”, “it”, “on”) are removed. In addition to common words, custom stop words such as domain-specific jargon, can be removed.
- **Stemming.** Word stemming is applied to find the morphological root of each word (e.g., “changing” and “changes” both become “chang”), typically using the Porter algorithm [692], although other algorithms exist.
- **Pruning.** The vocabulary of the resulting corpus is pruned by removing words that occur in, for example, over 80% or under 2% of the documents [554].

5.3.1.2 Source Code Preprocessing

If the input data is source code, the characters related to the syntax of the programming language (e.g., “&&”, “->”) are often removed, and programming language keywords (e.g., “if”, “while”) are removed. These steps result in a dataset containing only the comments, identifiers, and string literals. Some research also takes

steps to remove certain comments, such as copyright comments or unused snippets of code. 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, comments, and string literals in the source code [695].

5.3.1.3 Email Preprocessing

Preprocessing email is an ongoing research challenge [103, 775], due to the complex nature of emails. The most common preprocessing steps include:

- Detecting and removing noise: header information in replies or forwards; previous email messages; and signature lines [825].
- Isolating source code snippets or stack traces [53, 102], so that they can either be removed or treated specially. We discuss this step in more detail in Section 5.4.7.

5.3.1.4 Tools

Many researchers create their own preprocessing tools, based on commonly available toolkits such as the NLTK module in Python [111] and TXL [198] for parsing source code. The authors of this chapter have released their own tool, called *lscp*⁵, that implements many of the steps described above.

5.3.2 Information Retrieval

“Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).”

— Manning [557, p. 1]

IR is used to find specific documents of interest in a large collection of documents. Usually, a user enters a query (i.e., a text snippet) into the IR system, and the system returns a list of documents related to the query. For example, when a user enters the query “software engineering” into the Google IR system, it searches every web page ever indexed and returns those that are somehow related to software engineering.

IR models—the internal workings of IR systems—come in many forms, from basic keyword-matching models to statistical models that take into account the location of the text in the document, the size of the document, the uniqueness of the matched term, and even whether the query and document contain shared topics of interest [948]. Here, we briefly describe three popular IR models: the Vector Space

⁵ github.com/doofuslarge/lscp

Model, Latent Semantic Indexing, and latent Dirichlet allocation. A full treatment of each of these models is beyond the scope of this chapter. Instead, we aim to capture the most essential aspects of the models.

5.3.2.1 The Vector Space Model

The Vector Space Model (VSM) is a simple algebraic model based on the *term-document matrix* A of a corpus [741]. A is an $m \times n$ matrix, where m is the number of unique terms, or words, across n documents. The i^{th}, j^{th} entry of A is the weight of term w_i in document d_j (according to some weighting function, such as term-frequency). VSM represents documents by their column vector in A : a vector containing the weights of the words present in the document, and 0s otherwise. The similarity between two documents (or between a query and a document) is calculated by comparing the similarity of the two column vectors of A . Example vector similarity measures include Euclidean distance, cosine distance, Hellinger distance, or KL divergence. In VSM, two documents will only be deemed similar if they contain at least one shared term; the more shared terms they have, and the higher the weight of those shared terms, the higher the similarity score will be.

VSM improves over its predecessor, the Boolean model, in two important ways. First, VSM allows the use of term weighting schemes, such as tf-idf (term frequency, inverse document frequency) weighting. Weighting schemes help to downplay the influence of common terms in the query and provide a boost to documents that match rare terms in the query. Another improvement is that the relevance between the query and a document is based on vector similarity measures, which is more flexible than the strict Boolean model [741].

Tools. Popular tools implementing VSM include Apache Lucene⁶ and gensim⁷.

5.3.2.2 Latent Semantic Indexing

Latent Semantic Indexing (LSI) (or *Latent Semantic Analysis* (LSA)) is an information retrieval model that extends VSM by reducing the dimensionality of the term-document matrix by means of *Singular Value Decomposition* (SVD) [232]. During the dimensionality reduction phase, terms that are related (i.e., by co-occurrence) are grouped together into topics. This noise-reduction technique has been shown to provide increased performance over VSM for dealing with polysemy and synonymy [57], two common issues in natural language.

SVD is a factorization of the original term-document matrix A that reduces its dimensionality by isolating its singular values [740]. Since A is likely to be sparse, SVD is a critical step of the LSI approach. SVD decomposes A into three matrices:

⁶ lucene.apache.org

⁷ radimrehurek.com/gensim

$A = TSD^T$, where T is an m by $r = \text{rank}(A)$ 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 r , the rank of the original term-document matrix. 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 range between 50–300.

As in VSM, terms and documents are represented by row and column vectors, respectively, 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 be formulated and evaluated against the matrix. However, because of the reduced dimensionality of the term-document matrix after SVD, these measures are well equipped to deal with noise in the data.

Tools. For LSI, popular implementations include *gensim* and R's LSA package.

5.3.2.3 Latent Dirichlet Allocation

Latent Dirichlet allocation (LDA) is a popular probabilistic topic model [117] which takes a different approach to representing documents than previous models. The main idea behind LDA is that 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 is a set of topics that describe the entire corpus. Each document is composed of one or more than one of these. Each term in the vocabulary can be contained in more than one of these topics. Hence, LDA is able to discover a set of ideas or themes that well describe the entire corpus [116].

LDA is based on a fully generative model that describes how documents are created. Informally stated, 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:

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

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

Like any generative model, the task of LDA is that of *inference*: given the terms in the documents, which topics did they come from, and what are the topics themselves? LDA performs inference with *latent variable models* (or *hidden variable models*), which are machine learning techniques devised for just this purpose: to



Fig. 5.3: Four example topics (their labels and top words) from JHotDraw 7.5.1. We also show a snippet of the file `SVGCreateFromFileTool.java`, with terms colored corresponding to the topic from which they came. In this example, no terms come from the “Undoable Edit” or “Bezier Path” topics.

associate observed variables (here, terms) with latent variables (here, topics). A rich literature exists on latent variable models [e.g., 74, 113]; for the purposes of this chapter, we omit the details necessary for computing the posterior distributions associated with such models.

Figure 5.3 shows example topics discovered by LDA from version 7.5.1 of the source code of JHotDraw⁸, a framework for creating simple drawing applications. For each example topic, the figure shows an automatically-generated two-word topic label and the top (i.e., highest probable) words for the topic. The topics span a range of concepts, from opening files to drawing Bezier paths.

Tools. Implementations of LDA include MALLET [575], gensim, R’s LDA and topicmodels packages, lucene-lda, lda-c, and Stanford’s topic modeling toolbox⁹, amongst others.

⁸ www.jhotdraw.org

⁹ nlp.stanford.edu/software/tmt

5.4 The State of The Art

Many software engineering tasks can be addressed or enhanced by incorporating unstructured data. These tasks include concept and feature location, traceability linking, calculating source code metrics, statistical debugging, studying software evolution and trends, searching software repositories, managing bug databases and requirements documents. In this section, we describe how these tasks can be performed using IR models, while also pointing the interested reader to further reading material.

5.4.1 Concept/Feature Location and AOP

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 [707]. 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 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.

Concept location is a straightforward application of IR models on source code. The general method is to preprocess the source code as outlined in Section 5.3.1, build an IR model on the preprocessed source code, accept a developer query such as “*file printing*”, and use the IR model to return a list of related source code documents. Table 5.1 summarizes various approaches and studies that have been performed in this area. Many IR models have been considered, including VSM, LSI, LDA, and combinations thereof. While most researchers report some success with their concept location methods, there is not yet a consensus as to which IR model performs best under all circumstances.

Related to concept location is *aspect-oriented programming* (AOP), which aims to provide developers with the machinery to easily implement aspects of functionality whose implementation spans over many source code documents. Recently, a theory has been proposed that says software concerns are equivalent to the latent topics found by statistical topic models, in particular LDA [67]. In particular, aspects are exactly those topics that have a relatively high scatter metric value. After testing this theory on a large set of open-source systems, researchers find that this theory holds true most of the time [67].

5.4.2 Traceability Recovery and Bug Localization

An often-asked question during software development 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. *Bug localization* is a special case of traceability recovery in which developers seek traceability links between bug reports and source code, to help locate which source code files might be related to the bug report.

Typically, an IR model is first applied to the preprocessed source code, as well as the documentation or other textual repository. Next, a similarity score is calculated between each pair of documents (e.g., source code document and documentation documents). A developer then specifies a desired value for the similarity threshold, and any pair of documents with similarity greater than the threshold would be presented.

Table 5.1 outlines related research in this area. LSI has been the IR model of choice for many years, likely due to its success in other domains. Recently, however, multiple IR models have been empirically compared, as outlined in Table 5.1. From this research, we find that LDA is usually reported to achieve better results than LSI, but not in every case. Additional research is required to further determine exactly when, and why, one IR model outperforms another.

5.4.3 Source Code Metrics

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

Often, the state-of-the-art in bug prediction is advanced by the introduction of new metrics. An impressive suite of metrics has thus far been introduced, counting somewhere in the hundreds. For example, the *coupling* metric measures how interdependent two entities are to each other, while the *cohesion* metric measure how related the elements of an entity are to each other. Highly coupled entities make maintenance difficult and bug-prone, and thus should be avoided. Highly cohesive entities, on the other hand, are thought to follow better design principles.

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). Recently, researchers have used IR models to introduce *semantic* or *conceptual* metrics, which are mostly based on the linguistic data in the source code. Table 5.1 lists research that uses IR models to measure metrics on the linguistic data. Overall, we find that LSI-metrics

have been used with success, although more recent work reports that LDA-based metrics can achieve better results.

5.4.4 Software Evolution and Trend Analysis

Analyzing and characterizing how a software system changes over time, or the *software evolution* [506] of a system, has been of interest to researchers for many years. Both *how* and *why* a software system changes can help yield insights into the processes used by a specific software system as well as software development as a whole.

To this end, LDA has been applied to several versions of the source code of a system to identify the trends in the topics over time [525, 834, 835]. 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. Researchers have evaluated the effectiveness of such an approach, and found that spikes or drops in a topic's popularity indeed coincided with developer activity mentioned in the release notes and other system documentation, providing evidence that LDA provides a good summary of the software history [832].

LDA has also been applied to the commit log messages in order to see which topics are being worked on by developers at any given time [401, 402]. LDA is applied to all the commit logs in a 30 day period, and then successive periods are linked together using a topic similarity score (i.e., two topics are linked if they share 8 out of their top 10 terms).

LDA has also been used to analyze the Common Vulnerabilities and Exposures (CVE) database, which archives vulnerability reports from many different sources [641]. Here, the goal is to find the trends of each vulnerability, in order to see which are increasing and which are decreasing. Research has found that using LDA achieves just as good results as manual analysis on the same dataset.

Finally, LDA has recently been used to analyze the topics and trends present in Stack Overflow¹⁰, a popular question and answer forum [75]. Doing so allows researchers to quantify how the popularity of certain topics and technologies (e.g.: Git vs. SVN; C++ vs. Java; iPhone vs. Android) is changing over time, bringing new insights for vendors, tool developers, open source projects, practitioners, and other researchers.

¹⁰ www.stackoverflow.com

5.4.5 Bug Database Management

As bug databases grow in size, both in terms of the number of bug reports and the number of users and developers, better tools and techniques are needed to help manage their work flow and content.

For example, a semi-automatic bug triaging system would be quite useful for determining which developer should address a given bug report. Researchers have proposed such a technique, based on building an LSI index on the the titles and summaries of the bug reports [7, 41]. After the index is built, various classifiers are used to map each bug report to a developer, trained on previous bug reports and related developers. Research reports that in the best case, this technique can achieve 45% classification accuracy.

Other research has tried to determine how easy to read and how focused a bug report is, in an effort to measure the overall quality of a bug database. Here, researchers 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 [253, 524]. The researchers compared their metrics to human-generated analysis of the comments and find a high correlation, indicating that their automated method can be used instead of costly human judgements.

Many techniques exist to help find duplicate bug reports, and hence reduce the efforts of developers wading through new bug reports. For example, Runeson et al. [737] use VSM to detect duplicates, calling any highly-similar bug reports into question. Developer can then browse the list to determine if any reports are actually duplicates. The authors preprocess the bug reports with many NLP techniques, including synonym expansion and spell correction. Subsequent research also incorporates execution information when calculating the similarity between two bug reports [907]. Other research takes a different approach and trains a discriminative model, using Support Vector Machines, to determine the probability of two bug reports being duplicates of each other [801]. Results are mixed.

Finally, recent work has proposed ways to automatically summarize bug reports, based on extracting key technical terms and phrases [540, 709]. Bug summaries are argued to save developers time, although no user studies have been performed.

5.4.6 Organizing and Searching Software Repositories

To deal with the size and complexity of large-scale software repositories, several IR-based tools have been proposed, in particular tools for organizing and searching such repositories.

MUDABlue is an LSI-based tool for organizing large collections of open-source software systems into related groups, called software categories [454]. MUDABlue applies LSI to the identifier names found in each software system and computes the pairwise similarity between whole systems. Studies show that MUDABlue can achieve recall and precision scores above 80%, relative to manually created tags of

the systems, which are too costly to scale to the size of typical software repositories. LACT, an LDA-based tool similar to MUDABlue, has recently been shown to be comparable to MUDABlue in precision and recall [844].

Sourcerer is an LDA-based, internet-scale repository crawler for analyzing and searching a large set of software systems. Sourcerer applies LDA and the Author-Topic model to extract the concepts in source code and the developer contributions in source code, respectively. Sourcerer is shown to be useful for analyzing systems and searching for desired functionality in other systems [528, 529].

S³ is an LSI-based technique for searching, selecting, and synthesizing existing systems [694]. The technique is intended for developers wishing to find code snippets from an online repository matching their current development needs. The technique 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.

5.4.7 Other Software Engineering Tasks

LSI has been used to detect high-level clones of source code methods by computing the semantic similarity between pairs of methods [563]. Related work has used ICA for the same purpose, arguing 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 [345].

Aside from establishing traceability links from requirements to source code (described previously in Section 5.4.2), researchers have proposed many techniques to help manage and use the natural language text in requirements documents. These techniques include generating UML models from requirements [231], detecting conflicts in aspect-oriented requirements [744], identifying aspects in requirements [742], and assessing the quality of requirements [672].

IR methods require many parameter values to be configured before using. Various methods have been proposed to (semi-)automatically tune the parameters for software engineering datasets [80, 346].

Researchers are beginning to consider how discussion forums and question and answer websites might help developers. For example, new tools include finding relevant answers in formats [343], finding experts for a particular question [519], analyzing the topics and trends in Stack Overflow [75], and semi-automatically extracting FAQs about the source code [389].

Methods that incorporate email are receiving more attention from researchers. For example, lightweight IR methods have been used to link emails to their discussed source code entities [55]. In similar work, more heavy-weight classification techniques are used to extract source code from emails, which can be a useful first step for the aforementioned linking methods [54]. This technique was later revised

to use island grammars [53]. Finally, spell checkers [104] and clone detection techniques [102] have been used to locate and extract source code in emails.

Statistical debugging is the task of identifying a problematic piece of code, given a log of the execution of the code. Researchers have proposed Delta LDA, a variant of LDA, to perform statistical debugging [33]. 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, Delta LDA is able to identify the pieces of code that likely caused the bugs.

LSI has been used as a tool for root cause analysis (RCA), i.e., identifying the root cause of a software failure [132]. The tool builds and executes a set of test scenarios that exercise the system's methods in various sequences. Then, the tool uses LSI to build a method-to-test co-occurrence matrix, which has the effect of clustering tests that execute similar functionalities, helping to characterize the different manifestations of a fault.

Other tasks considered include automatically generating comments in source code [794], web service discovery [931], test case prioritization [833], execution log reduction [946], and analyzing code search engine logs [60, 61].

5.5 A Practical Guide: IR-based Bug Localization

In this section, we present a simple, yet complete tutorial on using IR models on unstructured data to perform bug localization, the task of identifying those source code files that are related to a given bug report. To make the tutorial concrete, we will use the source code and bug reports of the Eclipse IDE¹¹, specifically the JDT submodule.

To make the tutorial easily digestible, we will make the following simplifying assumptions:

- Only a single version of the source code will be analyzed, in our case, the snapshot of the code from July 1, 2009. Methods exist to analyze all previous versions of the code [832], but require some extra steps.
- Only basic source code representations will be considered. More advanced representations exist [836], such as associating a source code document with all the bug reports with which it has been previously associated.
- For simplicity, we will assume that we have access to all the source code and bug reports on the local hard drive. In reality, large projects may have their documents controlled by content management servers on the cloud, or via some other mechanism. In these cases, the steps below still apply, but extra care must be taken when accessing the data.

The basic steps to perform bug localization include (a) collecting the data, (b) preprocessing the data, (c) building the IR model on the source code, (d) and query-

¹¹ www.eclipse.org

Listing 5.1: An example Eclipse bug report (#282770) in XML.

```
<bug>
  <bug_id>282770</bug_id>
  <creation_ts>2009-07-07_23:48:32</creation_ts>
  <short_desc>
    _Dead_code_detection_should_have_specific_@SuppressWarnings
  </short_desc>
  <long_desc>
    _As_far_as_I_can_tell_there_is_no_option_to_selectively_turn_off
    _dead_code_detection_warnings_using_the_@SuppressWarnings
    _annotation._The_feature_either_has_to_be_disabled...
  </long_desc>
</bug>
```

ing the IR model with a particular bug report and viewing the results. We now expand on each step in more detail.

5.5.1 *Collect data*

Let's assume that that source code of Eclipse JDT is available in a single directory, `src`, with no subdirectories. The source code documents are stored in their native programming language, Java. There are 2,559 source code documents, spanning dozens of packages, with a total of almost 500K source lines of code.

We also assume that the bug reports are available in a single directory, `bugs`, with no subdirectories. Each bug report is represented in XML. (See Listing 5.1 for an example.) The Eclipse JDT project has thousands of bug reports, but in this tutorial we will focus on only one, shown in Listing 5.1.

5.5.2 *Preprocess the source code*

Several decisions must be made during the phase of preprocessing the source code. Which parts of the source code do we want to include when building the IR model? Should we tokenize identifier names? Should we apply morphological stemming? Should we remove stopwords? Should we remove very common and very uncommon words?

The correct answers are still an active research area, and may depend on the particular project. In this tutorial, we'll assume that identifiers, comments, and string literals are desired; we will tokenize identifier names; we will stem; and finally, we will remove stopwords.

Listing 5.2: Using `lscp` to preprocess the source code in the `src` directory.

```
#!/usr/bin/perl
use lscp;
my $preprocessor = lscp->new;
$preprocessor->setOption("isCode", 1);
$preprocessor->setOption("doIdentifiers", 1);
$preprocessor->setOption("doStringLiterals", 1);
$preprocessor->setOption("doComments", 0);
$preprocessor->setOption("doTokenize", 1);
$preprocessor->setOption("doStemming", 1);
$preprocessor->setOption("doStopwordsKeywords", 1);
$preprocessor->setOption("doStopwordsEnglish", 1);
$preprocessor->setOption("inPath", "src");
$preprocessor->setOption("outPath", "src-pre");

$preprocessor->preprocess();
```

To do so, we can use any of the tools mentioned in 5.3.1, such as `lscp` or `TXL` [198], or write our own code parser and preprocessor. In this chapter, we'll use `lscp`, and preprocess the source code with the Perl script shown in Listing 5.2. The script specifies the preprocessing options desired, gives the path to the source code (in our case, `src`), and specifies where the resulting files should be placed (here, `src-pre`). After running the Perl script shown Listing 5.2, we'll have one document in `src-pre` for each of the documents in `src`.

5.5.3 Preprocess the bug reports

As with source code, several decisions need to be made when preprocessing a bug report. Should we include its short description only, long description only, or both? If stack traces are present, should we remove them? Should we tokenize, stem, and stop the words?

As with preprocessing source code, the best answers to these design decisions are yet unknown. In this tutorial, we'll include both the short and long description; leave stack traces if present; and tokenize, stem, and stop the words. Note that it is usually a good idea to perform the same preprocessing steps on the queries as we did on the documents, as we have done here, so that the IR model is dealing with a similar vocabulary.

To do the preprocessing, we again have a number of tools at our disposal. Here, we'll again use `lscp` and write a simple Perl script similar to that shown in Listing 5.2. The only differences will be setting the options `isCode` to 0, `inPath` to `bugs`, `outPath` to `bugs-pre`, and removing the `doStringLiterals` and `doComments` options, as they no longer apply. We leave the `doIdentifiers` option, in case the bug report contains snippets of code that contain identifiers.

Before inputting the bug reports into lscp, we must first use a simple XML parser to parse the bug report and strip out the text content from the two elements we desire, `<short_desc>` and `<long_desc>`.

5.5.4 Build the IR model on the source code

In this tutorial, we'll build an LDA model, one of the more advanced IR models. To build the LDA model, we'll use the lucene-lda tool¹², which is a tool to build and query LDA indices from text documents using the Apache Lucene framework.

We use the command prompt to issue the following command, assuming we're in the lucene-tool's base directory:

```
$ ./bin/indexDirectory --inDir src-pre --outIndexDir \
out-lucene --outLDADir out-lda --K 64
```

We pass the `src-pre` directory as the input directory, and specify two output directories, `out-lucene` and `out-lda`, where the Lucene index and LDA output will be stored, respectively. We also specify the number of topics for the LDA model to be 64 in this case, although choosing the optimal number of topics is still an active area of research. We'll use the tool's default behavior for the α and β smoothing parameters of LDA, which will use heuristics to optimize these values based on the data.

The tool will read all the files in the input directory and run LDA using the MALLET toolsuite. MALLET will create files to represent the topic-term matrix (i.e., which terms are in which topics) and the document-topic matrix (i.e., which topics are in which source code documents). The lucene-lda tool will use these files, along with the Lucene API, to build an index that can be efficiently stored and queried, which will be placed in the `out-lucene` directory.

5.5.5 Query the LDA model with a bug report

Now that the index is built, it is ready to be queried. Although we have the ability to query the index with any terms or phrases we desire, in the task of bug localization, the terms come from a particular bug report. Since we've already preprocessed all of the bug reports, we can choose any one and use it as a query to our pre-built index:

```
$ ./bin/queryWithLDA --inIndexDir out-lucene --intLDADir \
out-lda --query bugs-pre/#282770 --resultFile results.dat
```

¹² github.com/doofuslarge/lucene-lda

Listing 5.3: Results of querying the LDA model with Eclipse bug report #282770. Each row shows the similarity score between the bug report and a source code document in the index.

```
6.75, compiler.impl.CompilerOptions.java
5.00, internal.compiler.batch.Main.java
4.57, internal.formatter.DefaultCodeFormatterOptions.java
3.61, jdt.ui.PreferenceConstants.java
3.04, internal.compiler.ast.BinaryExpression.java
2.93, core.NamingConventions.java
2.85, internal.ui.preferences.ProblemSeveritiesConfigBlock.java
2.74, internal.formatter.DefaultCodeFormatter.java
2.59, internal.ui.text.java.SmartSemicolonAutoEditStrategy.java
...
```

Here, we specify the names of the directories holding the Lucene index and supporting LDA information, the name of the query (in this case, the preprocessed version of the bug report shown in Listing 5.1), and the file that should contain the results of the query.

The tool reads in the query and the Lucene index, and uses the Lucene API for execute the query. Lucene efficiently computes a similarity score between the query and each document, in this case based on their shared topics. Thus, the tool infers which topics are in the query, computes the *conditional probability* between the query and each document, and sorts the results.

After running the query, the tool creates the `results.dat` file, which contains the similarity score between the query and each document in the index. Listing 5.3 shows the top 10 results for the this particular query, ordered by the similarity score. These top files have the most similarity with the query, and thus should hopefully be relevant for fixing the given bug report. Indeed, as we know from another study [836], the file `internal.compiler.batch.Main.java` was changed in order to fix bug report #282770. We see this file as appearing second in the list in Listing 5.3.

The above result highlights the promising ability of IR models to help with the bug localization problem. Out of the 2,559 source code documents that *may* be relevant to this bug report, the IR model was able to pinpoint the *most relevant file* on its second try.

IR models are not always this accurate. Similar to issuing a web search and not being able to find what you're looking for, IR-based bug localization sometimes can't pinpoint the most relevant file. However, research research has found that IR models can pinpoint the most relevant file to a bug report within the top 20 results up to 89% of the time [836], a result that is sure to aid developers quickly wade through their thousands of source code files.

5.6 Conclusions

Over the past decade, researchers and practitioners have started to realize the benefits of mining their software repositories: using readily-available data about their software projects to improve and enhance performance on many software evolution tasks. In this chapter, we surveyed tools and techniques for mining the *unstructured* data contained in software repositories.

Unstructured data brings many challenges, such as noise and ambiguity. However, as we demonstrated throughout this chapter, many software evolution tasks can be enhanced by taking advantage of the rich information contained in unstructured data, including concept and feature location, bug localization, traceability linking, computing source code metrics to assess source code quality, and many more.

We focused our survey on natural language processing (NLP) techniques and information retrieval (IR) models, which were originally developed in other domains to explicitly handle unstructured data, and have been adopted by the software engineering community. NLP techniques, such as tokenization, identifier splitting, stop word removal, and morphological stemming, can significantly reduce the noise in the data. IR models, such as the Vector Space Model, Latent Semantic Indexing, and latent Dirichlet allocation, are fast and simple to compute and bring useful and practical results to software development teams. Together, NLP and IR models are an effective approach for researchers and practitioners to mine unstructured software repositories.

Research in the area of mining unstructured software repositories has become increasingly active over the past years, and for good reason. We expect to see continued advances in all major areas in the field: better NLP techniques for preprocessing source code, bug reports, and emails; better tailoring of existing IR models to unstructured software repositories; and novel applications of IR models in to solve software engineering tasks.

Table 5.1: Applications of IR models. A (C) indicates that the work compares the results of multiple IR models. *Continued in Table 5.2.*

Application	Summary
<i>Concept loc.</i> Markus et al. [564, 565]	First to use LSI for concept location. LSI provides better context than existing concept location methods, such as regular expressions and dependency graphs. Using LSI for concept location in object-oriented (OO) code is useful, contrary to previous beliefs.
<i>Concept loc. (C)</i> Cleary et al. [192]	Tests two IR techniques, VSM and LSI, against NLP techniques. NLP techniques do not offer much improvement over the two IR techniques.
<i>Concept loc.</i> van der Spek et al. [791]	Considers the effects of various preprocessing steps, using LSI. Both splitting and stopping improve results, and term weighting plays an important role, but no weighting scheme was consistently best.
<i>Concept loc.</i> Grant et al. [347]	Uses ICA, a model similar to LSI, to locate concepts in source code. The viability of ICA is demonstrated through a case study on a small system.
<i>Concept loc.</i> Compare Models Linstead et al. [526, 527]	Uses LDA to locate concepts in source code. Demonstrates how to group related source code documents based on the documents' topics. Also uses a variant of LDA, the Author-Topic model [731], to extract the relationship between developers (authors) and source code topics. The technique allows the automated answer of "who has worked on what".
<i>Concept loc.</i> Maskeri et al. [571]	Applies LDA to source code, using a weighting scheme for each keyword in the system, based on where the keyword is found (e.g., class name vs. method name). The technique is able to extract business topics, implementation topics, and cross-cutting topics from source code.
<i>Concept loc.</i> Poshyvanyk, Rev- elle et al. [696, 697, 712, 713]	Combines LSI with various other models, such as Formal Concept Analysis, dynamic analysis, and web mining algorithms (HITS and PageRank). All results indicate that combinations of models outperform individual models.
<i>Traceability</i> Marcus et al. [566]	Uses LSI to recover traceability links between source code and requirements documents. Compared to VSM, LSI performs at least as well in all case studies.
<i>Traceability</i> De Lucia et al. [222–224]	Integrates LSI traceability into ADAMS, a software artifact management system. Also proposes a technique for semi-automatically finding an optimal similarity threshold between documents. Finally, performs a human case study to evaluate the effectiveness of LSI traceability recovery. LSI is certainly a helpful step for developers, but that its main drawback is the inevitable trade off between precision and recall.
<i>Traceability (C)</i> Hayes et al. [376]	Evaluates various IR techniques for generating traceability links between various high- and low-level requirements. While not perfect, IR techniques provide value to the analyst.
<i>Traceability</i> Lormans et al. [537–539]	Evaluates different thresholding techniques for LSI traceability recovering. Different linking strategies result in different results; no linking strategy is optimal under all scenarios. Uses LSI for constructing <i>requirement views</i> . For example, one requirement view might display only requirements that have been implemented.
<i>Traceability</i> Jian et al. [440]	Proposes a new technique, incremental LSI, to maintain traceability links as a software system evolves over time. Compared to standard LSI, incremental LSI reduces runtime while still producing high quality links.

Table 5.2: *Continued from Table 5.1.* Applications of IR models.

Application	Summary
<i>Traceability</i> de Boer et al. [220]	Develops an LSI-based tool to support auditors in locating documentation of interest.
<i>Traceability</i> Antoniol et al. [39]	Introduces a tool, ReORE, to help decide whether code should be updated or rewritten. ReORE uses static (LSI), manual, and dynamic analysis to create links between requirements and source code.
<i>Traceability</i> McMillan et al. [580]	Combines LSI and Evolving Inter-operation Graphs to recover traceability links. The combination modestly improves traceability results in most cases.
<i>Traceability (C)</i> Lukins et al. [544, 545]	Compares LSI and LDA for bug localization. After two case studies on Eclipse and Mozilla, the authors find that LDA often outperforms LSI.
<i>Traceability (C)</i> Nguyen et al. [644]	Introduces BugScout, an IR model for bug localization, which explicitly considers past bug reports as well as identifiers and comments. BugScout can improve performance by up to 20% over traditional LDA.
<i>Traceability (C)</i> Rao et al. [708]	Compares several IR models for bug localization, including VSM, LSI, and LDA, and various combinations. Simpler IR models (such as VSM) often outperform more sophisticated models.
<i>Traceability (C)</i> Copabianco et al. [165]	Compares VSM, LSI, Jenson-Shannon, and B-Spline for recovering traceability links between source code, test cases, and UML diagrams. B-Spline outperforms VSM and LSI and is comparable to Jenson-Shannon.
<i>Traceability (C)</i> Oliveto et al. [661]	Compares Jenson-Shannon, VSM, LSI, and LDA for traceability. LDA provides unique insights compared to the other three techniques.
<i>Traceability (C)</i> Asuncion et al. [49]	Introduces TRASE, an LDA-based tool for prospectively, rather than retrospectively, recovering traceability links. LDA outperforms LSI in terms of precision and recall.
<i>Fault detection</i> Marcus et al. [567]	Introduces C3, an LSI-based class cohesion metric. Highly cohesive classes correlate negatively with program faults.
<i>Fault detection</i> Gall et al. [307]	Presents natural-language metrics based on design and requirements documents. The authors argue that tracking such metrics can help detect problematic or suspect design decisions.
<i>Fault detection</i> Ujhazi et al. [864]	Defines two new conceptual metrics that measure the coupling and cohesion of methods, both based on LSI. The new metrics provide statistically significant improvements compared to previous metrics.
<i>Fault detection</i> Liu et al. [533]	Introduces MWE, an LDA-based class cohesion metric, based on the average weight and entropy of a topic across the methods of a class. MWE captures novel variation in models that predict software faults.
<i>Fault detection</i> Gethers et al. [322]	Introduces a new coupling metric, RTC, based on a variant of LDA called Relational Topic Models. RTC provides value because it is statistically different from existing metrics.
<i>Fault detection</i> Chen et al. [175]	Proposes new LDA-based metrics including: NT, the number of topics in a file and NBT, the number of buggy topics in a file. These metrics can well explain defects, while also being simple to understand.



<http://www.springer.com/978-3-642-45397-7>

Evolving Software Systems

Mens, T.; Serebrenik, A.; Cleve, A. (Eds.)

2014, XXIII, 404 p., Hardcover

ISBN: 978-3-642-45397-7