Uncovering File Relationships using Association Mining and Topic Modeling

Namita Dave, Delmar Davis, Karen Potts, Hazeline U. Asuncion
Computing and Software Systems
University of Washington, Bothell
Bothell, WA, USA
{namitad, davisdb1, pottsk2, hazeline}@u.washington.edu

***Abstract*—Software maintenance tasks, such as feature enhancements and bug fixes, require familiarity with the entire software system. A modification task could become very time consuming if there is no prior knowledge of the system. Association mining has been used to identify the files that frequently change together in a software repository, and this information can aid a software engineer locate relevant files for a maintenance task. However, association mining techniques are limited to the amount of project history stored in a software repository. We address this difficulty by using a technique that combines association mining with topic modeling, referred to as Frequent Pattern Growth with Latent Dirichlet Allocation (FP-LDA). Topic modeling aims to uncover file relationships by learning semantic topics from source files. We validated our technique via case studies on two open source projects. Our results indicate that topic modeling can increase the effectiveness of association mining in uncovering the file relationships.**

Keywords-Association mining; Topic Modeling; Software Engineering

I. INTRODUCTION

Software maintenance is the largest cost contributor in the lifecycle of a product [1]. This may be due to an engineer's unfamiliarity with the software to modify, requiring more time to understand the source code [2]. Maintenance tasks also become more difficult as the complexity of the code increases and as code degradation occurs over time due to patches and workarounds [3].

Techniques to find related source code files include static and dynamic analysis, recommendation systems, and code search techniques. Static analysis techniques, more specifically, dependency analysis, provide file relationships based on call graphs [4]. Dynamic analysis tools, meanwhile, are able to identify relationships between files based on execution traces [5]. These techniques, however, are generally language-specific. Recommendation systems, meanwhile, provide possible files of interest based on a developer's past activities, textual similarity, check-in records, or email records [6, 7]. These systems generally use information retrieval techniques along with user context to provide files of interest. Code search techniques find related code based on syntactic or structural matches [8].

Association mining (AM) is another technique to find related files. AM uncovers relationship between files based on files that are modified together in the past. This technique generates rules, which specify which files are frequently changed together. Unlike the other techniques, AM is not specific to the programming language used or restricted to syntactic or structural matches of a query.

Most commonly used algorithms for association mining are Apriori [9] and Frequent Pattern Growth (FP-Growth) [10]. While these algorithms provide some level of accuracy, they are highly dependent on the project history. If there are not enough modifications in the software project, or if the modifications are sparse throughout the software system, there are fewer chances that AM will result in correct rules.

Meanwhile, machine learning techniques such as Latent Dirichlet Allocation (LDA), allows us to detect relationships between files based on semantic similarity. LDA is an unsupervised statistical approach for learning semantic topics from a set of documents [11]. It is a fully automated approach that does not require training labels. It only requires a set of documents and number of topics to learn.

Thus, we aim to address the challenges of AM by combining AM with LDA. Our technique, Frequent Pattern Growth with Latent Dirichlet Allocation (FP-LDA), allows us to achieve better recall results than solely using AM. By combining these two techniques, we are able to overcome the limitations of each technique. LDA allows us to find file associations even with limited modification history. AM, meanwhile, allows us to find associations among files where semantic similarities may not be readily apparent.

The contributions of this research paper are as follows: (1) combination of AM and topic modeling to find file relationships in a software project, and (2) case studies on two open source projects. We also created a set of tools that automates the entire process—from pre-processing the data to querying related files.

The rest of the paper is organized as follows. The next two sections provide background on our combined approach. Section 2 covers AM and Section 3 covers topic modeling. In Section 4, we present our combined approach, FP-LDA. We then validate our approach in Section 5. Related work is discussed in Section 6. We conclude with future work.

II. ASSOCIATION MINING

A. Background

Association rule mining is a method to discover patterns in large data sets. Initially, it was used in Market Basket Analysis to find how items bought by customers are related [12]. Rules are mined from the dataset, such as "Customers who bought item A also bought item B". In the case of software projects, rules such as "Developers who modified

 ISBN: 978-1-61208-329-2

file A also modified file B" are mined [13]. In order to mine these rules, patterns must be analyzed in the dataset.

We now discuss the main concepts of AM [9], as applied to software development.

Let $I = \{i_1, i_2, \ldots\ldots i_m\}$ (Eq.1) represent total set of items. In this paper, the files in the repository are items. T represents a set of transactions $T = \{t_1, t_2, \ldots\ldots t_n\}$ (Eq. 2), which are in the software repository being mined. Each transaction t is a set of items such that $t \subseteq I$. In this paper, t represents one atomic commit.

Given the set of transactions T (see Eq.2), the goal of AM is to find all the association rules that have support and confidence greater than the user specified threshold values. An itemset is a collection of items. The support is defined as the fraction of transactions that contain the itemset and from which the rule is derived. The confidence denotes the strength of a rule. An association rule is represented as

$X \rightarrow Y$ [support = 20%, confidence = 80%] (Eq.3)

In this notation, itemset X is called the antecedent and itemset Y is called consequent such that $X, Y \subseteq I$. Both antecedent and consequent comprise of one or more items. Assume that both X and Y consist of one file each namely x and y respectively. Then, this rule says that in 20% of the check-in transactions, both x and y files are modified and the transactions which changed file x also changed file y 80% of the time.

The threshold support value specified by the user is called minimum support. This is an important element that makes AM practical. It reduces the search space by limiting the number of rules generated. The threshold confidence value specified by the user is the minimum confidence [14].

B. Selection of Association Mining Technique

Two commonly used association-mining techniques are Apriori algorithm [9] and Frequent Pattern Growth Algorithm, or FP-Growth [10].

Apriori is a classic algorithm for learning association rules over transactional databases [9]. The essential idea behind Apriori algorithm is that it iteratively generates candidate itemsets of length (k + 1) from frequent itemsets of length k and then tests their corresponding frequency in the database. Apriori is not efficient when used with large data sets as generation of candidate item sets and support counting is very expensive, as confirmed in [15].

FP-Growth is a faster and scalable approach to mine a complete set of frequent patterns by pattern fragment growth using a compact prefix tree structure for storing transaction dataset [10]. In the first step, it creates a compact Frequent Pattern tree to encode the database. The construction of an FP-tree begins with pre-processing the input data with an initial scan of the database to count support for single items. The single items that do not meet the threshold support values are eliminated. The database is then scanned for the second time to produce an initial FP-tree. The second step runs a depth first recursive procedure to mine the FP-tree for frequent itemsets with increasing cardinality. The FP tree stores a single item at each node. The root node of an FP tree is empty. The path from root to a node in the FP tree is a subset of transactions database. The items in the path are in decreasing order of support. In the second step, the algorithm examines conditional-pattern base for each itemset starting with length 1 and then constructs its own conditional FP-tree. Unlike Apriori algorithm, it avoids generating expensive candidate itemsets. Each conditional FP-tree is recursively mined to generate frequent itemsets. The algorithm uses divide and conquer approach to decompose mining task into smaller tasks of mining the confined conditional databases. Interested readers can refer to [10] for more information.

C. Limitations

AM is useful in finding patterns in the data that satisfy minimum support and minimum confidence constraints. However, some researchers have shown that AM often results in redundant and unimportant rules. A drawback is that it is difficult to eliminate insignificant rules [16].

In this research, the number of association rules generated depends on the amount of modification history of a project. Also, there is a possibility that not all modules or files may be changed during a software maintenance phase. This can affect the number of rules generated.

III. TOPIC MODELING

A. Background

LDA is an unsupervised statistical approach for learning semantic topics from a set of documents [11]. Since it is an unsupervised machine learning technique, no training labels are necessary. This is a fully automated approach that only requires a set of documents and the number of topics to learn. Here are some concepts used in LDA:

- A word is a basic unit of discrete data.
- A document is characterized by a vector of word counts.
- A corpus has a total of W words in its vocabulary.
- D documents placed side by side, gives W x D matrix of counts.
- A topic is a probability distribution over W words.
- Each document is associated with a probability distribution over T topics.

LDA is a generative Bayesian topic model for a corpus of documents. The basic concept behind LDA is that it discovers topics from a collection of documents [11]. Then it learns a distribution over words for each topic. To obtain a semantic interpretation of a topic, we simply examine the highest-probability words in that topic. For example, if a topic has high probability words "window", "dialog", "height", "width", "button", we can infer the topic to be related to the user interface of the software. Lastly, it defines each document as a probabilistic mixture of these topics. Each document can belong to multiple topics. Additional details regarding LDA's generative process are in [11].

As we discuss in the next section, we use LDA to determine possible relationships between source code files through their topic distributions. Each source code file equates to a document in LDA.

 ISBN: 978-1-61208-329-2

B. Selection of Topic Modeling Technique

Topic modeling algorithms generally fall under two categories: sampling-based and variational methods [11]. Sampling-based algorithms collect samples to approximate the posterior with an empirical distribution. Variational methods, meanwhile, use a parameterized family of distributions and then find the member of the family that is closest to the posterior. In this paper, we use a fast version of Collapsed Variational Inference (CVB0) for LDA [17], which has been shown to be among the fastest and most accurate methods for learning topic models.

C. Limitations

LDA has generally been applied to unstructured text [18]. Meanwhile, source code is a highly structured text that has a limited range of semantic concepts. The results are also subject to parameters used in LDA. As a result, researchers have examined ways to fine-tune the parameters [19].

We processed the source code prior to running LDA such that reserved words are removed and only semantically meaningful words are used. Our pre-processing technique is similar to the pre-processing technique described here [20].

IV. COMBINED APPROACH

Our technique, FP-LDA, aims to uncover possible file relationships regardless of the amount of project history available and regardless of the programming language used. (Some knowledge of the language used in the source code is needed to eliminate language-reserved words from the source code. See the next section for more details.) FP-LDA consists of the following steps: (1) data extraction and pre-processing, (2) association data mining, (3) topic modeling, and (4) result querying. Fig. 1 shows a high level process of our technique. Each layer in the figure corresponds to each of these steps. All the processes represented by a rectangle have been implemented.

A. Data Extraction and Pre-processing

Pre-processing version history for data mining. This first step involves extracting the version history of an open source project and preparing the data to be fed as input to the mining algorithm (Step A2). We created a tool that accesses the version history of the project, processes it, and stores the history data in MySQL database. For projects using Subversion (SVN), we used SVNKit application programming interfaces (APIs) [21] to access the version history of the project. SVNKit is an open source Java-based SVN library. For projects using Git, we used JavaGit [22] API to access the version history.

Data pre-processing is an important step in that it removes all unwanted data that may impact data mining (A2). In our technique, our goal is to create a generic pre-processing step to support different open source projects. Thus, we used the following conditions when determining the type of transactions to include in our AM. Similar to [13], we do not include transactions with more than one hundred files since these transactions may contribute to noise. Such commits may be due to specialized tasks, such as formatting all source code files and then checking-in all files together. We also removed transactions that do not assist in identifying relationships between files, such as single file commits, non-source code commits (e.g., graphic files), and commits of deleted files. The remaining valid transactions are then stored in a database (A3). This is the dataset that will be analyzed by the mining algorithm. We then transform this dataset into a file format that conforms to expected format of the mining algorithm (A4).

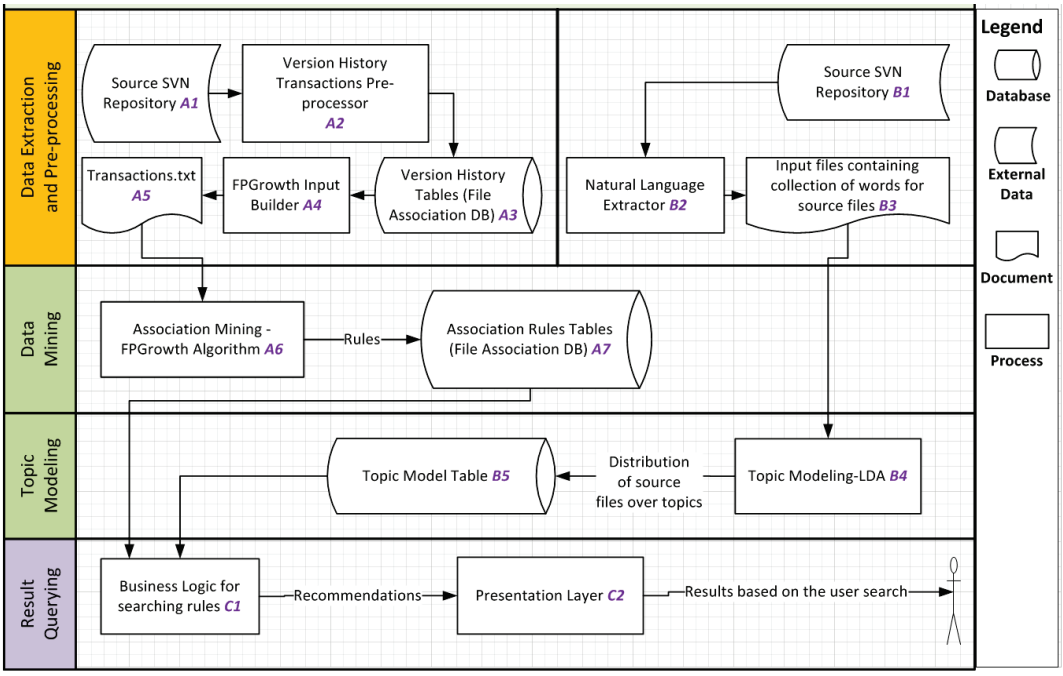

Figure 1. FP-LDA data flow to find file dependencies

Pre-processing source files for LDA. While the mining algorithm examines the entire commit history, we use topic modeling to extract topics from the latest version of the source code. We extracted from the source code semantically meaningful text, such as comments, identifier names and string literals. These words provide clues on the purpose or functionality of the code (B2).

To extract these words, we run each file through a tokenizer. The tokenizer aids in splitting words with underscore or in camel case to obtain the name of objects or variables. We also specified a set of stop words that are programing language-reserved words, commonly occurring words (e.g., the, was), and common terms in a software project. We also removed words like "get" and "set" since source files contain methods that start with these words. This requires some knowledge of the programming language syntax. Another option is to generate the Abstract Syntax Tree using tools like ANTLR [23] to support multiple languages. The generated tree can then be explored to extract the comments and identifiers inside the source code.

B. Association Data Mining

Once the data is preprocessed, we run the data mining algorithm (A6). We used Frequent Pattern Growth (FP-Growth) algorithm for AM, more specifically, the Liverpool University Computer Science – Knowledge Discovery in Data (LUCS-KDD) implementation of FP-Growth. This Java implementation uses tree structures for AM [24]. In our current case study, we restrict the length of the frequent itemsets generated to 2. (In the future, we will use the rules with more than one item in antecedent and consequent to uncover more complex relationships. For example, which files change given that two input files are modified.) Since we currently store the frequent 2-itemsets in database, we can compute the union of consequents using an SQL query. We store the generated frequent itemsets in a database (A7).

C. Topic Modeling

Once the source files are pre-processed, we extract semantic topics using LDA (B4). We used the CVB0 implementation of LDA [17]. Our implementation of LDA has the following parameters: number of topics and number of iterations. Number of topics is the number of topics we specify. The greater the number of topics, the more fine-grained will be the generated topics. Number of iterations is the number of times the algorithm will run. The higher number of iterations increases the likelihood that the topics will converge. For our case studies, we observed that the topics converged by 5000 iterations.

D. Result Querying

The last step is to query the results of both the rules generated from AM and the document relationship to topics (C1). We assume that the user is aware of at least one file that has to be modified for a given modification task. This file is used as the input. The output will show all the files that are recommended or predicted to change along with the input file.

V. VALIDATION

In this section, we discuss how we assessed our technique. We cover the setup of our case study, the results, and the limitations of our approach.

A. Case Study Setup

We conducted a case study on two open source projects: ArgoUML and EclipseFP. ArgoUML is a UML editor that also performs model checks [25]. This project uses the SVN repository, has around 6600 files, 14,000 commit transactions, and has a modification history of more than ten years. The second project we used is EclipseFP [26]. This is an Eclipse plugin for Haskel programming. This project uses Git version control system. This project consists of around 2000 files, has 1,796 commit transactions, and has a modification history of eight years. We selected these projects because these are active projects.

To measure the effectiveness of our approach, we used precision and recall. Precision measures the conciseness of a recommendations provided by the approach. Recall measures how many relevant recommendations are made by using this approach. We followed the same approach as used by Ying et al [13]. In this case study, we have assumed that developer is aware of at least one file for a given modification task. Therefore, we specified only one file f_s for generating recommendations for a modification task m. As explained in [13], the precision $precision(m, f_s)$ of a recommendation $recom(f_s)$ is the fraction of files that are predicted correctly and are part of the solution $f_{sol}(m)$ for the modification task m. The recall $recall(m, f_s)$ of a recommendation $recom(f_s)$ is the fraction of files recommended out of $f_{sol}(m)$.

For example, let us consider a modification task that requires changing files {a, b, c, d}. In addition, let us assume that the recommendations obtained for file b using our approach are files {a, c}. In this case, the precision for file b in this modification task is 100% as the approach recommended correct files. The recall value for file b for same modification task is 66.67% because the approach could predict only two files {a, c} out of {a, c, d}.

In order to determine the effectiveness of our prediction algorithm, we generated FP-Growth rules using 90% of the commit transactions. We then calculated the precision and recall rates of the generated rules on the remaining 10% of the commit transactions. We split the dataset based on time, since this simulates actual practice. Then, we compared these precision and recall rates with the precision and recall rates of FP-LDA.

The parameters we used are as follows. Minimum support for AM for ArgoUML is 10 and EclipseFP is 15. Confidence value for both projects is 25%. The number of topics for LDA was 20 topics. We measured the precision and recall of FP-LDA approach for 10%, 40% and 75% topic distribution values.

 ISBN: 978-1-61208-329-2

B. Results

The FP-growth resulted in 401 rules for ArgoUML and 42 for EclipseFP. The average precision and recall values obtained for ArgoUML with just FP-Growth are 0.48 and 0.06 respectively. The average precision and recall values for EclipseFP using AM are 0.32 and 0.13 respectively. FP-LDA results in lower precision and higher recall using the topic distribution cutoffs. Fig. 2 shows these values for both projects at various distribution cutoffs for topic modeling.

C. Discussion

The calculation of precision and recall gives a general understanding of how the approach fares in finding relationships. We assumed that each of these transactions was a task presented to a developer. For each file in the test transaction, we calculated precision and recall values to see if the tool can predict the remaining files.

We see that average precision reduces with the use of FP-LDA. This is due to the fact that the number of total recommendations increases due to topic modeling. However, a higher recall value shows that there is an increase in the number of relevant files predicted. This proves that number of correct recommendations increases with LDA. Although the overall precision is lower with combined approach, the utility of the approach lies in the fact that a developer needs to search only the set of recommended files, and not the entire source code base. Moreover, since we solely base our precision and recall on actual check-in records in the latter 10% of the history record, it is entirely possible that two files are related, but they may not have been checked-in together within this subset of the data. Thus, one can consider our precision number as a lower bound (e.g., FP-LDA precision for EclipseFP is at least 32% for 20 topics).

In addition to calculating precision and recall, we examined certain transactions to see how the combined FP-LDA fares over using AM alone. For example, in the ArgoUML dataset, AM fails to predict any relationship between files CrSingletonViolatedMissingStaticAttr.java and CrConsiderFacade.java (see Fig. 3). However, topic-modeling results show that these files are related. Upon analyzing these files, we found these two files have an indirect inheritance relationship. Similarly, in EclipseFP dataset, the classes DeltaVisitor.java and FullBuildVisitor.java implement interfaces IResourceDeltaVisitor.java and IResourceVisitor.java respectively. IResourceDeltaVisitor extends from IResourceVisitor.java. Topic Model shows that these files are related. However, since there was only one transaction where these files were changed together, AM does not show these files as related. These examples illustrate that FP-LDA technique can correctly identify related files.

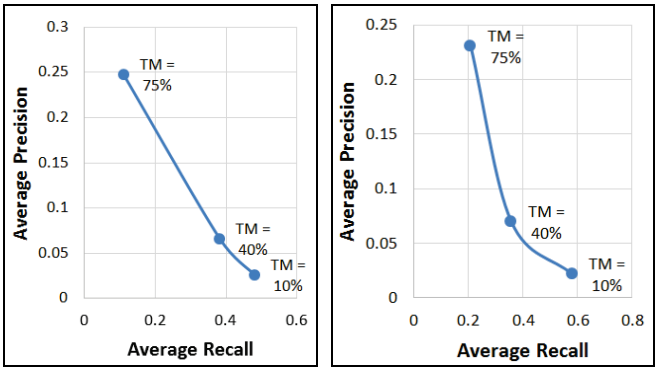

Figure 2. Precision and recall for ArgoUML, minimum support = 10 (left) and EclipseFP (right) minimum support of 15

D. Limitations of the study

Our precision and recall numbers may be subject to the specific datasets we selected. However, even though these datasets are different (e.g., the ratio between transactions and number of files is much greater in ArgoUML than in EclipseFP), we still observe a general increase in recall rates when using FP-LDA.

The number of topics used in LDA may also affect precision and recall rates. We ran our technique using different topic numbers and observed that the smaller the topic number, the higher the recall rates and the lower precision rates are generated. The "right" number of topics differs from one dataset to another.

VI. Related Work

Our work is most closely related to previous work in mining frequently changed files from a software repository [13, 27]. We used AM as other software engineering researchers have used this technique in the past. We build on top of this existing work and examine the benefits of combining AM with topic modeling. While others have used collaborative filtering [28], we use topic modeling, which is a probabilistic version of matrix factorization over the word-document matrix. In this paper, we use topic modeling to analyze the semantic content of source code and commit comments. In previous work, we have used topic modeling to identify associations between various software files and architecture components [29]. In the future, we plan to use topic modeling to identify associations between files and authors. Our work is also related to other techniques that seek to identify relationships between software files, such as recommendation systems, code search techniques, and dependency analysis.

Recommendation systems for software engineering may also recommend files for modification. Not all recommendation systems use association rule mining, but eRose a plugin for Eclipse does [7]. The common factor among all recommendation systems for software engineering is that they rely on the user's context in order to provide recommendations. While recommendation systems may help find related files in source code, the issue of user context is outside of the scope of our work.

 ISBN: 978-1-61208-329-2

Trans#	Name of File	P-noTM	R-noTM	P-TM75	R-TM75
17831	CrConsiderSingleton.java	0	0	0.13	0.75
17831	InitPatternCritics.java	0	0	0	0
17831	CrSingletonViolatedOnlyPrivateConstructors.java	0	0	0.13	0.75
17831	CrSingletonViolatedMissingStaticAttr.java	0	0	0.13	0.75
17831	CrConsiderFacade.java	0	0	0.13	0.75

Figure 3. FP-LDA is able to predict more files that are related to each other (P-TM75 and R-TM75) than FP-Growth alone (P-noTM, R-noTM)

Code search techniques may also be used to find source files that are related to one another. These techniques have their roots in traditional information retrieval methods [30]. An equivalency study was undertaken to the compare various IR methods in the area of traceability recovery [31]. The results of this study showed that while Latent Semantic Indexing (LSI), Jensen-Shannon (JS), and Vector Space Model (VSM) provided higher accuracy in identifying related files, LDA was able to capture associations, which the other methods could not. Recent work in code search has been performed to enhance the accuracy of these methods by allowing the user to specify both the syntactic and semantic properties of a search [30]. Code search techniques, however, fall short in finding relationships between project files, which are not semantically or syntactically related. Meanwhile, our technique finds these relationships based on the change history of the project and semantic relationship.

Dependency analysis tools may be used to find relationships between source files based on call graphs [4]. By making use of the project histories, we can mine relationships between any files that are checked-in together, as opposed to simply analyzing the code structure. As discussed previously, we also have the ability to find relationships between source code written in different languages. Most importantly, this approach helps to detect cross cutting concerns in which there may be a relationship between two files, but no relationship in a call-graph. For example, a project created for multiple operating systems may contain two source files, which accomplish the same task, but have no relationship in the calling tree. In this case, dependency analysis cannot detect these relationships, but our approach can because of the semantic similarity between files.

VII. Conclusion and Future Work

In this paper, we used AM and topic modeling together to assist developers in software maintenance task. These techniques were used to uncover the source file dependencies within a software project. We applied AM on version history of a project to find files that frequently change together. We complemented this technique by using topic modeling on the source code documents. We showed that using topic modeling could uncover file dependencies that are not captured due to lack of version history for those files. Our evaluation indicates that this combination of techniques increases recall rates by more than double, based on the open source projects we analyzed.

In the future, we would like to explore various options that can measure the usefulness of this approach. We also plan to examine other means to pre-process our data (e.g., aggregating the transactions based on time interval and committer to obtain a logical grouping of transactions). Finally, we plan to analyze more open source projects as well as conduct user studies to determine whether our approach reduces the time required for impact analysis or any maintenance task.

Acknowledgment

We thank Arthur U. Asuncion for his insights on LDA and providing the CVB0 implementation of LDA. We also thank Eamon Maguire for his assistance in extracting version histories and running the mining algorithm. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1218266. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] S. S. Yau and J. S. Collofello, “Some stability measures for software maintenance,” Trans. on Software Engineering (TSE), vol. SE-6, Nov. 1980, pp. 545–552, doi:10.1109/TSE.1980.234503.

[2] A. J. Ko, B. A. Myers, M. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” TSE, vol. 32, Dec 2006, pp. 971–987, doi:10.1109/TSE.2006.116.

[3] R. N. Taylor, N. Medvidovic, and E. Dashofy, Software Architecture: Foundations, Theory, and Practice. John Wiley & Sons, 2010.

[4] M. Sharp and A. Rountev, “Static analysis of object references in RMI-based Java software,” in Proc of the Int’l Conf on Software Maintenance (ICSM), Sep. 2005, pp. 101–110, doi:10.1109/ICSM.2005.84.

[5] M. Eaddy, A. V. Aho, G. Antoniol, and Y. G. Gueheneuc, “Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis,” in Proc of the 16th Int’l Conf on Program Comprehension (ICPC), Jun. 2008, pp. 53–62, doi:10.1109/ICPC.2008.39.

[6] D. Cubranic, G. C. Murphy, J. Singer, and S. Booth Kellogg, “Hipikat: a project memory for software development,” Trans. on Software Engineering (TSE), vol. 31, Jun. 2005, pp. 446–465, doi:10.1109/TSE.2005.71.

[7] M. P. Robillard, R. J. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” IEEE Software, vol. 27, Jul-Aug. 2010, pp. 80–86, doi:10.1109/MS.2009.161.

[8] S. Bajracharya, J. Ossher, and C. V. Lopes, “Sourcerer - an infrastructure for large-scale collection and analysis of open-source code,” Science of Computer Programming, vol. 79, Jan. 2014, pp. 241–259, doi:10.1016/j.scico.2012.04.008.

 ISBN: 978-1-61208-329-2

[9] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in Proc of 20th Int’l Conf on Very Large Data Bases, Sep. 1994, pp. 487–499.

[10] J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” in Proc of the 2000 Int’l Conf on Mgmt of Data, May 2000, pp. 1–12, doi:10.1145/342009.335372.

[11] D. M. Blei, “Probabilistic topic models,” Comunications of the ACM, vol. 55, Apr 2012, pp. 77–84, doi:10.1145/2133806.2133826.

[12] R. Agrawal, T. Imielinski, and A. Swami, “Mining association rules between sets of items in large databases,” SIGMOD Rec., vol. 22, Jun. 1993, pp. 207–216, doi:10.1145/170036.170072.

[13] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, “Predicting source code changes by mining change history,” TSE, vol. 30, Sep. 2004, pp. 574–586, doi:10.1109/TSE.2004.52.

[14] B. Liu, W. Hsu, and Y. Ma, “Mining association rules with multiple minimum supports,” in Proc of the Fifth ACM SIGKDD Int’l Conf on Knowledge Discovery and Data Mining, Aug. 1999, pp. 337–341, doi:10.1145/312129.312274.

[15] J. Pei et al., “Mining sequential patterns by pattern-growth: the PrefixSpan approach,” Trans. on Knowledge and Data Engineering, vol. 16, Nov. 2004, pp. 1424–1440, doi:10.1109/TKDE.2004.77.

[16] B. Liu, W. Hsu, and Y. Ma, “Identifying non-actionable association rules,” in Proc of Int’l Conf on Knowledge Discovery and Data Mining, Aug. 2001, pp. 329–334, doi:10.1145/502512.502560.

[17] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh, “On smoothing and inference for topic models,” in Proc of Conf. on Uncertainty in Artificial Intelligence, Jun. 2009, pp. 27–34.

[18] B. Gretarsson et al., “TopicNets: Visual analysis of large text corpora with topic modeling,” Trans. on Intelligent Systems and Technology, vol. 3, Feb. 2012, pp. 23:1–23:26, doi:10.1145/2089094.2089099.

[19] A. Panichella et al., “How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms,” in Proc of the Int’l Conf on Software Engineering (ICSE), May 2013, pp. 522–531.

[20] T. Savage, B. Dit, M. Gethers, and D. Poshyvanyk, “TopicXP: Exploring topics in source code using latent dirichlet allocation,” in Proc of the Int’l Conf on Software Maintenance (ICSM), Sep. 2010, pp. 1–6, doi:10.1109/ICSM.2010.5609654.

[21] TMate Software, “SVNKit.” http://svnkit.com/, retrieved: Jan. 2014.

[22] “JavaGit.” http://javagit.sourceforge.net/, retrieved: Jan. 2014.

[23] “ANTLR.” http://www.antlr.org/, retrieved:Jan. 2014.

[24] F. Coenen, G. Goulbourne, and P. Leng, “Tree structures for mining association rules,” Data Mining and Knowledge Discovery, vol. 8, Jan. 2004, pp. 25–51, doi:10.1023/B:DAMI.0000005257.93780.3b.

[25] CollabNet, “ArgoUML.” http://argouml.tigris.org/, retrieved:Jan. 2014.

[26] “EclipseFP, the Haskell plug-in for Eclipse.” http://eclipsefp.github.io/, retrieved: Jan. 2014.

[27] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” TSE, vol. 31, Jun. 2005, pp. 429–445, doi:10.1109/TSE.2005.72.

[28] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, “Item-based collaborative filtering recommendation algorithms,” in Proc of the 10th International Conference on World Wide Web, May 2001, pp. 285–295, doi:10.1145/371920.372071.

[29] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, “Software traceability with topic modeling,” in Proc of ICSE, vol. 1, May 2010, pp. 95–104, doi:10.1145/1806799.1806817.

[30] S. P. Reiss, “Semantics-based code search,” in Proc of ICSE, May 2009, pp. 243–253, doi:10.1109/ICSE.2009.5070525.

[31] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, “On the equivalence of information retrieval methods for automated traceability link recovery,” in Proc of ICPC, Jun-Jul. 2010, pp. 68–71, doi:10.1109/ICPC.2010.20.

 ISBN: 978-1-61208-329-2