Information and Software Technology 57 (2015) 369–377

Contents lists available at ScienceDirect

Information and Software Technology

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

Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation

Ying Fu [b], Meng Yan [b], Xiaohong Zhang [a,b,*], Ling Xu [b], Dan Yang [b], Jeffrey D. Kymer [b]

[a] Key Laboratory of Dependable Service Computing in Cyber Physical Society Ministry of Education, Chongqing 400044, PR China
[b] School of Software Engineering, Chongqing University, Chongqing 401331, PR China

ARTICLE INFO

Article history:
Received 12 September 2013
Received in revised form 21 May 2014
Accepted 22 May 2014
Available online 6 June 2014

Keywords:
Software repositories mining
Semi-supervised topic modeling
LDA
Change message

ABSTRACT

Context: Topic models such as probabilistic Latent Semantic Analysis (pLSA) and Latent Dirichlet Allocation (LDA) have demonstrated success in mining software repository tasks. Understanding software change messages described by the unstructured nature-language text is one of the fundamental challenges in mining these messages in repositories.
Objective: We seek to present a novel automatic change message classification method characterized by semi-supervised topic semantic analysis.
Method: In this work, we present a semi-supervised LDA based approach to automatically classify change messages. We use domain knowledge of software changes to make labeled samples which are added to build the semi-supervised LDA model. Next, we verify the cross-project analysis application of our method on three open-source projects. Our method has two advantages over existing software change classification methods: First of all, it mitigates the issue of how to set the appropriate number of latent topics. We do not have to choose the number of latent topics in our method, because it corresponds to the number of class labels. Second, this approach utilizes the information provided by the label samples in the training set.
Results: Our method automatically classified about 85% of the change messages in our experiment and our validation survey showed that 70.56% of the time our automatic classification results were in agreement with developer opinions.
Conclusion: Our approach automatically classifies most of the change messages which record the cause of the software change and the method is applicable to cross-project analysis of software change messages.
© 2014 Elsevier B.V. All rights reserved.

1. Introduction

Since the pioneering work of Mockus and Votta [1] has exemplified that a textual description field of a change is useful to understand why that change was performed, classification of change messages corresponding to software changes has become a useful task in software lifecycle management assisting in tracing and monitoring: software evolution [2], software maintenance [3], software state, and development. For example, using these classifications the project manager can adjust the development plan according to the rate of corrective changes versus adaptive changes. A high rate indicates that too much time is spent fixing bugs. This information provides a suggestion to enforce quality control practices to improve the quality of code. The classification performance and automatically classifying speed are naturally critical to the project manager's decision. The main goal of this paper is to provide a novel and good performance method for automatically classifying the large corpus of changes using only the textual descriptions of change messages.

Many nature language processing (NLP) techniques have been extensively utilized for classifying change messages for conducting many software engineering activities, such as keyword based matching [1,4,5], Weka Machine Learning framework [6], naive Bayes classifiers [7], and so on. The successful applications of NLP in software engineering activities result from several facts: (1) Change messages record the purpose of making the changes in nature-language, such as adding a new feature or a bug fix, but not explicitly labeling the change category. (2) There are many software engineering domain terms included in the textual description of change messages, such as "bug", "fix", and "maintain". (3) The description is brief and unformatted. The textural description of change messages is effectively used for conducting many software engineering activities such as quickly classifying

* Corresponding author at: School of Software Engineering, Chongqing University, Huxi Town, Shapingba, Chongqing 401331, PR China. Tel.: +86 15923238399.
E-mail address: xhongz@cqu.edu.cn (X. Zhang).

http://dx.doi.org/10.1016/j.infsof.2014.05.017
0950-5849/© 2014 Elsevier B.V. All rights reserved.

the change messages and filtering out irrelevant versions. This is significantly less expensive and simple [6] than retrieving then analyzing the source code of the change. Among NLP techniques, the topic model provides a powerful tool for discovering and exploiting the hidden thematic structure in large archives of text, which can discover hidden semantic relations between words [8]. It had been introduced into the domain of software engineering to help to solve various software engineering research questions, such as software evolution [9] and software defect prediction [10]. Especially, probabilistic Latent Semantic Analysis (pLSA) [11] and Latent Dirichlet Allocation (LDA) [12] which have demonstrated success in mining software repository tasks [8,9,13–15]. In this paper, our research is motivated by the success mentioned above of applying the statistical topic model to automatically extract topics by mining textual software repositories [1,5,16].

Despite the great success some issues with topic models remain in software engineering. First of all, many works applied the LDA model to extract topics from software repositories [8,9,13,15,17]. However, their works seldom considered using the software engineering domain knowledge to cue the application of the topic model. In fact, the domain knowledge is useful in classification or pattern discovery [18] as Wu and Chien [19] showed the semi-supervised LDA with the domain knowledge added achieved higher classification performance compared to the other methods on text categorization. Secondly, many researchers set the number of topics by their experience or referred to the results of similar studies such that there is no guarantee that the latent topics found by their approaches will necessarily correspond to the goal of the research. For example, Somasundaram and Murphy [20] used the LDA model to automatically categorize bug reports by assuming T to be 20, 70, 120, 170, 220, 270 to compute an estimate of $P(W|T)$, and attained a peak for a particular value of T. The number of topics was set to be that particular value in their research. However, the latent topics found by this method do not correspond to the bug report categories. In this paper, we made two changes to the original LDA based method: (1) In building the semi-supervised LDA model, we added labeled samples with the domain knowledge of software changes to train the model. In this way, we influence the co-occurrence frequency of words which belong to the same category. (2) The latent topics in our model directly correspond to the class labels and we classified the change messages according to a comparison of the similarity between topics and class labels instead of the probability distribution of the topics.

The contributions of this paper are twofold. First, we mitigate the issue of how to set the appropriate number of latent topics. We do not have to choose the number of latent topics because the latent topics in our model directly correspond to the software change class labels. Second, we extended the standard LDA model to make use of the class label information with the domain knowledge in the trained data, and trained the LDA in a semi-supervised fashion. To understand the detail of our work and how our research performs, we focused on the following research questions.

- **RQ1** How to build a semi-supervised LDA based model for change message features?
- **RQ2** Could the iterations affect the results of the classification in our approach?
- **RQ3** How to automatically classify the software change messages by semi-supervised LDA modeling?

The remainder of this paper is structured as follows: Section 2 presents the related work of our research, including the research of software change and LDA modeling in mining software repositories (MSR). Section 3 provides the information of our study projects and the procedure of our research. Sections 4-7 provide our detailed research process and the application of our approach to three open-source projects. Section 8 presents a conclusion of our work and the potential threats to the validity of our study.

2. Related work

This section provides an overview of software change mining, the software change classification rules, and LDA modeling in mining software repositories (MSR).

2.1. The research of software change

Software change is one of the significant characteristics of software evolution [21]. And change messages in VCS are used to record the software changes by developers, including the cause of the changes and all the details about the updated, deleted, or added source code files. Most of the researchers studied software changes through the change messages and have obtained a great number of achievements.

2.1.1. Software change messages mining

In this subsection, we describe the related work conducted on some software engineering activities through software change message mining, such as: identifying the reasons for the software change, guiding software changes, and connecting the change messages to bug reports.

To understand the cause of software changes, Mockus and Votta [1] proposed a lexical analysis approach to automatically classify a change by a textual description of the changes. In their work, they not only showed that a textual description field of a change is useful to understand the cause of the changes but they also showed that difficulty, size, and interval would vary greatly across different types of changes. Hence, they defined every logically distinct change as a Modification Request (MR), where the keywords in the textual descriptions were directly used to classify the changes. For example, if the keywords such as "add", "new", "create", or "change" were present in a change description, the change was classified as adaptive. Their validation surveys showed that 61% of the time their automatic classification results were in agreement with developer opinions. In terms of Mockus's idea, Hassan [5] applied a similar approach to several open source projects as opposed to commercial projects. In this paper, unlike the work of Mockus, we will understand the cause of software changes by classifying change messages based on a semi-supervised LDA method, which mitigates the issue of how to set the appropriate number of latent topics and utilizes the information provided by the label samples in the training set.

After the work of Mockus and Votta [1], Hattori and Lanza [4] and Mauczka et al. [16] proposed some other variations of keyword approaches to classify change messages. The method of Hattori automatically classified each change commit into development or maintenance activities according to the first keywords found in the commit's comment, and the classification results were employed to further investigate the characteristics of each commit category. However, the first keyword found in the commit's comment used to classify change messages may cause some deviation when the first keyword cannot describe the cause of the changes. Mauczka developed an Eclipse plug-in named Subcat to automatically assess whether a change to the software was a bug fix, a refactoring, or others. They further introduced a weighting of keywords and rule sets for ambiguous, yet strongly indicative words. However, the weight of the keywords was defined by the researcher; hence it introduces the problem that different people have different opinions about the weight.

Except the keyword based methods and its variants, there were other machine learning methods used to mine the change

messages. For example, Antoniol et al. [7] discussed alternating decision trees, naive Bayesian classifiers, and logistic regression that automatically divided the change requests in the Bug Tracking System (BTS) into two classes: bugs and non-bugs and proposed an algorithm that accurately distinguished bugs from other issues. Hindle et al. [6] used several machine learners from Weka Machine Learning framework including J48, NativeBayes, SMO, KStar, IBk, JRip, and ZeroR to automatically classify large changes into the extended Swanson Categories of Changes. Zimmermann et al. [22] applied data mining to change messages to guide programmers along related changes and developed a tool called ROSE to discover association rules from a set of existing changes. The discovered association rules are used to predict the change locations and show item coupling that is undetectable by program analysis and can prevent errors due to incomplete changes.

It is important to note that many researchers [23–26] discussed the link between change messages and bug reports for predicting bugs. Bachmann et al. [26] linked the change messages to bug reports through mining the information in change messages, and explored how the linked data bias affected the BUGCACHE algorithm. Although there are some advantages to cross-referencing a bug report in the bug database with a commit in the VCS to assist automatic classification of change messages, it is difficult to find out the corresponding relationship between change messages and bug reports without automatic-assisted tools.

2.1.2. Software change classification rules

The research field of the identification and classification of software changes has evolved for decades. The first classification system of software changes was put forward by Swanson [21] in 1977. Nowadays, Swanson's classification definition is widely used for mining software repositories. Swanson defined a maintenance task as a change that can be classified into one of his classification definitions. His definition is as follows:

- Corrective: These are the changes that are made to fix processing failures, performance failures, or implementation failures.
- Adaptive: These are the activities that focus on the changes in data environment or processing environment.
- Perfective: These are the changes which are made to improve processing efficiency, enhance the performance, or increase the maintainability.

Since the work of Swanson, there have been many variants and extensions of his classification system. For instance, Hassan [5] provided a simpler classification rule than Swanson's. In Hassan's classification rule, software changes are classified into three categories: Bug Fixing change (BF), General Maintenance change (GM), and Feature Introduction changes (FI). In addition, a fourth category Not Sure (NS) was introduced when asking participating developers to classify the change messages. Hindle et al. [9] extended Swanson's categorization into corrective-, adaptive- and perfective changes with two additional changes: implementation changes and non-functional changes. Mauczka et al. [16] also made a slight extension to Swanson's original definition by introduced an additional category "Blacklist".

In our work, we adopted a modified version of Swanson's original maintenance-classification to meet our requirements. In addition to the three categories which were defined by Swanson, a fourth category which is Not Sure (NS) was added. This was also adopted and defined by Hassan. The decision to use Swanson's definition of maintenance tasks has been made because it provides a categorization into a few, well defined categories and is therefore a very good starting-point to construct the keywords-list which is used to classify the change messages. If the change messages do not have sufficient information to be automatically classified into one of the other three categories, they will be classified into the NS category.

2.2. LDA modeling in mining software repositories

Mining software repositories (MSR) is a technique which focuses on analyzing and understanding the data in software repositories which is related to the software development lifecycle. To combat the complexities of MSR, a recent achievement was the use of statistical topic models, such as PLSA [11], LDA [12], CTM [27] and their variants.

Hindle et al. [9] proposed the link Latent Dirichlet Allocation model for discovering a set of topics in change messages over a defined time-window, such as 30 days. The link model with 20 topics per window showed what topics are being worked on by the developers and identify activity trends in a particular time window. Later, Hindle et al. [17] presented another approach to labeling change descriptions by LDA modeling. In their work, they first set the number of topics to 20 to generate topics from change descriptions, then they labeled the topics by using a non-functional requirements (NFR) taxonomy which is based on the ISO quality model [28]. In addition to being used to analysis change messages, LDA is also used to analysis other artifacts in software repositories. Linstead et al. [29] adapted and applied LDA, for the first time, to extract concepts from source code and demonstrated the results on the SourceForge and Apache projects. Their study effectively demonstrated the effectiveness of LDA to extract topics from source code. Liu et al. [30] as well applied LDA to extract the latent topics embedded in comments and identifiers in source code and combined with information entropy measures to quantitatively evaluate the cohesion of classes in software. In addition to methodological advancements, LDA is also implemented in tools. Asuncion et al. [31] introduced a tool called TRASE that uses LDA for prospectively recovering traceability links amongst diverse artifacts in software repositories which include source code, email, requirement design documents, and bug reports. Asuncion demonstrated that LDA outperforms LSI [32] in terms of precision and recall. It is noted that the researchers mentioned above experimented with manual or automatic techniques for choosing an optimal value for the topic number.

The LDA based method is not only used to analysis the unstructured data but also used to classify unstructured data such as done in the work of Somasundaram and Murphy [20]. Somasundaram used the LDA model to automatically categorize bug reports into the appropriate components. They investigated three approaches to automating bug report categorization, and found that LDA combined with the Kullback Leibler divergence (LDA-KL) based approach achieved the most consistent results, which achieved a recall of 70–95%. The combined work of Hindle and Somasundaram provided a preliminary and rough clue that LDA can be used to classify the change messages for us. Since the number of topics in Somasundaram's work [20] was chosen by experiment the discovered topics did not correspond to the bug report categories. So far, the number of topics is a critical parameter and determining the number of topics is still an open problem.

3. Research methodology

3.1. The data under study

We performed our experiments on the change messages repositories of five mature open-source projects (Table 1 lists the details of these projects). Exploring the effect of iterations on Boost and Wireshark and validating our approach on Bugzilla, Firebird, and Python. The five projects belonged to different fields, programmed

Table 1
Five open-source projects.

Projects	Application type	Start date	Devs	Changes	Programming language
Bugzilla	Project management	August 1998	62	27539	Perl
Wireshark	Packet analyzer	September 1998	43	40511	C
Boost	Prog. library	July 2000	294	42208	C++
Firebird	RDBMS	May 2001	43	48622	C++
Python	Interpreter	August 1998	216	45032	C

in different programming languages, and the number of change messages in each is near 30,000 changes. Another feature of these projects is that the number of developers is at least 40. The reasons that we decided to choose these projects as our experimental data is as follows: (1) To validate if our approach can do across-project analysis. (2) To validate if our approach can understand nature-language and the expression diversity of different projects or developers. (3) To validate if the programming language or the application field would affect our approach.

3.2. LDA model

LDA is an unsupervised generative model that categorizes the words that appear in the corpus of documents into clusters which are typically referred to as topics. According to Blei et al. [12], the joint distribution of LDA is as follow:

$$p(w, z/\alpha, \beta) = p(w/z, \beta)p(z/\alpha) \quad (1)$$

The parameters α and β are given model parameters, w is the word, and z is the topic. The $p(z|\alpha)$ is the probability of topic z occurring in document d, and $p(w|z, \beta)$ is the probability of word w occurring in a particular topic z. The direct computation of Eq. (1) is intractable. According to Griffiths and Steyvers [33], we can use a suitable approximation instead of computing it directly. The suitable approximation is given by:

$$p(z_i = k/z_{-i}, w) \propto \frac{n^{w_i}_{-i,k} + \beta}{n^{\cdot}_{-i,k} + W\beta} \cdot \frac{n^{d_i}_{-i,k} + \alpha}{n^{d_i}_{-i,\cdot} + T\alpha} \quad (2)$$

In Eq. (2), $n^{(\cdot)}_{-i}$ is a count that does not include the current assignment of the topic z_i, $n^{w_i}_{-i}$ is the number of times the topic z which is associated with the word w_i, $n^{z_i}_{-i}$ is the number of times the topic z which is associated with the document d_i, W is the number of distinct words which are preprocessed and T is the number of topics. Since Eq. (2) is a straightforward Markov Chain, we use the Gibbs Sampler approach [33] to resolve it using approximation according to Griffiths.

3.3. Study setup

The procedure of our research was mainly divided into six parts. First, we analyzed the characteristic of the change message in order to adapt the LDA. Secondly, we preprocessed the change messages using nature-language text preprocessing. After data preprocessing, we pruned the change messages by removing messages that are too brief. Thirdly, we built three sets of term lists as signifiers which matched Swanson's [21] original definition of maintenance tasks. The construction of the term lists was conducted with the help of Mauczka's et al. [16]. Using these term lists, we converted the text information to numerical information by term frequency calculation. Fourthly, the related prior knowledge was added to the software change classification to build a semi-supervised LDA model. Fifthly, we choose the iterations through an experiment. Finally, we validated the applicability of our approach on three open-source projects.

3.3.1. Preprocessing the change messages

In this research, we preprocessed the software change messages of each project by applying the required nature-language text preprocessing steps used by any information retrieval technique. The change messages were preprocessed by Lucene[1] and Snowball[2] in our research. The preprocessing steps included: sentence splitting, term splitting, stop words filtering, and stemming. We filtered common English language stop words (such as: the, it, and on) to reduce noise. In addition, we stemmed the words (e.g. "fixing" becomes "fix") in order to reduce the vocabulary size and reduce duplication due to the word form.

3.3.2. Prune the change messages

Open-source projects do not enforce how to write a change description; hence, some change messages do not clearly describe the cause of the changes. After reading a lot of changes, we found that many change messages were too brief; hence they did not record the cause of the change clearly. This problem made it difficult to classify the software changes according to the text descriptions of the change messages. These change messages will be noise in the LDA modeling. So we pruned the change messages by removing these messages (the total number of words being less than five after data preprocessing) to reduce the noise of the data.

3.3.3. Constructing the term list

In the process of term frequency calculation, the key point is the construction of three sets of term lists as signifiers. Each set of term lists is associated with a type which matches Swanson's [21] original definition of maintenance tasks. In our work, the term lists which are used for cross-project analysis were derived with the help of Mauczka et al. [16] whose final dictionary was validated in analyzing cross-project data (see Table 2).

3.3.4. Choosing the number of topics

There is no agreed-upon standard method for choosing the value for the input parameter K (the number of topics) in advance, although some heuristics have been proposed [33–35]. The choice of the value of K is a trade-off between coarser topics (smaller K) and finer-grained topics (large K). After long term research, researchers put forward *typical values* of K instead of a correct value.

In our work, we classified software change messages into three categories according to Swanson's original definition of maintenance. Naturally, we made the latent topics in our models directly correspond to the software change class label. We kept $K = 3$ in our method, therefore, the topics correspond to the three categories in Swanson's definition.

3.3.5. The hyper-parameters of LDA

As to the number of topics, there is no agreed-upon standard method for generating the values of α and β. The α is the Dirichlet prior parameter of the topics within each document and the β is the Dirichlet prior parameter of words within each topic. Although

[1] http://lucene.apache.org/.
[2] http://snowball.tartarus.org/.

Table 2
Keywords in three sets of term lists.

Signifier	Keywords
Corrective	Bugfix bug cause error failure fix miss null warn wrong bad correct incorrect problem opps valid invalid fail bad dump except
Adaptive	Add new create feature function appropriate available change compatibility config configuration text current default easier future information internal method necessary old patch protocol provide release replace require security simple structure switch context trunk useful user version install introduce faster init
Perfective	Clean cleanup consistent declaration definition documentation move prototype remove static style unused variable whitespace header include dead inefficient useless

Griffiths's [33] algorithm was easily extended to allow α and β to be sampled, this extension could slow the convergence of the Markov Chain. Therefore we set the values of the hyper-parameters the same as the Griffiths's work. In our research, we kept the hyper-parameters α to be $50/T$ (T is the number of topics) and β to be 0.01, as described by Griffiths.

4. Semi-supervised LDA (RQ1)

The original LDA model is an unsupervised model. Although Sections 1 and 2 provide evidence for the applicability of LDA, directly applying the original LDA based approach to the change messages is not a good idea because of the differences between the change messages and nature-language text. As to the features of change message which are mentioned above, we added the domain knowledge of software changes to train the LDA in a semi-supervised fashion.

In Fig. 1(a), the probabilistic graphical model of LDA shows two processes; $\alpha \to \theta \to z$ and $z \to w$. The process $\alpha \to \theta \to z$ indicates the generation process of topic z. As well, the process $z \to w$ indicates the generation process of word w. In Fig. 1(b), the probabilistic graphical model of the semi-supervised LDA also shows the same two processes. The gray cycle indicates the main difference between LDA and semi-supervised LDA is the generation of topic z. In the semi-supervised LDA model, we added signifier documents to change the unsupervised training process into a semi-supervised fashion. The signifier documents can influence the generation process of the words because they can increase the co-occurrence [36] frequency of the keywords which belong to the same category used by human supervision.

The original LDA only provided the probability of topics within the document and the probability of words within that topic. In our semi-supervised LDA based method, the classification algorithm relies on the similarity comparison between the unclassified documents and the signifier documents. Therefore, the similarity is calculated by calculating the distance between the unclassified documents and the signifier documents. Here we used the Cosine Similarity method as used in previous research [37]. The specific calculation equation is as follows:

$$sim(d_1, d_2) = \frac{\sum_i (d_{1i} * d_{2i})}{\sqrt{\sum_i d_{1i}^2 * \sum_i d_{2i}^2}} \quad (3)$$

where d_1 and d_2 are two documents, and the i-th dimension weight in topic spaces are d_{1i} and d_{2i}.

5. The effect of iterations (RQ2)

In the LDA model the iterations are the Gibbs samplings. There is no agreed-upon standard method for choosing the iterations. As described by Griffiths and Steyvers [33], the iterations can be selected according to the demands of the problem. To explore the best iterations for our study, an experiment was designed to explore the iteration effects on the automatic classification accuracy. We applied the semi-supervised LDA based method to two open-source projects (Boost and Wireshark) which are introduced in Section 3.1. In this experiment, we set the iterations to be 1, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000. Then for all iteration values, we randomly selected 60 change messages of every project to evaluate the accuracy of automatic classification. As Fig. 2 shows, the accuracies of both Boost and Wireshark were sustainably increasing between 1 and 100, and then remained stable between 100 and 700. After 700 iterations the accuracies began to decrease slightly then remained stable. The results suggested that the classification accuracies reached a peak at iterations between 100 and 700.

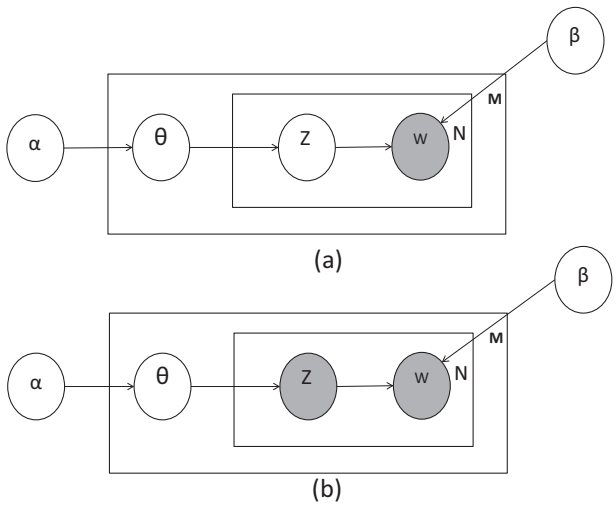

Fig. 1. Graphical representation of: (a) the LDA model, adopted from Blei et al.; (b) the semi-supervised LDA for training.

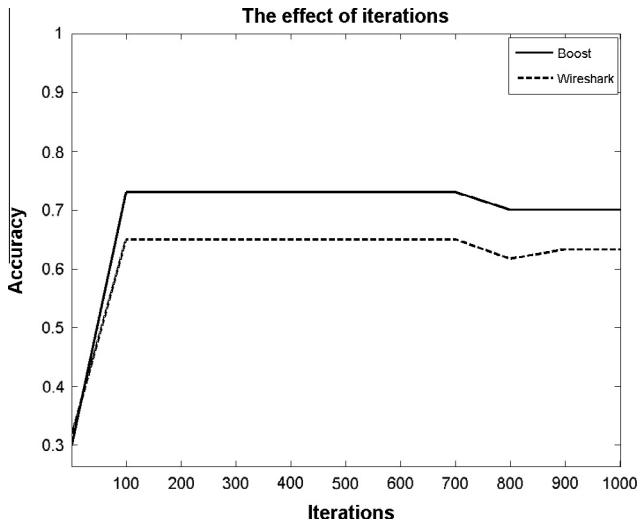

Fig. 2. The effect of iterations on classification accuracy (Boost, Wireshark).

Then according to the results of our first experiment, we narrowed the change granularity and set the iterations to be 1, 5, 10, 15, 20, 25, 30, 35, 40, 45, and 50 to explore the effect of the iterations further. As Fig. 3 shows, the accuracies of both Boost and Wireshark were sustainably increasing between 1 and 30, and then remained stable between 30 and 50. Furthermore the accuracies of 50 iterations and 100 iterations were equal.

According to the results of these two experiments, we decided to set the iterations to be 100 in our approach where the accuracies peaked in both Boost and Wireshark.

6. Applying semi-supervised LDA to open-source projects (RQ3)

In order to verify the cross-project analysis applicability of the semi-supervised LDA based method, we applied the method to the rest of the three open-source projects which were introduced in Section 3.1.

In this experiment, we divided the data of each project into two parts. The 50% data used to test the approach and we combined the other 50% data with 25% signifier documents which are mentioned in Section 3.3.2 as training data. Therefore the amount of the training data is about 60% and the amount of the testing data is about 40%. For all projects, we used $K = 3$, $\alpha = 50/K$, $\beta = 0.01$, and iterations = 100. The detailed procedure of the experiment is shown in Fig. 4. The dotted line represents the training course of the model. The solid line represents the testing procedure. The bold solid line represents the shared part of model training and model testing.

6.1. Results and conclusions

The results of our automatic classification approach are summarized in Table 3. As mentioned above, the change messages with words less than five were not used to classify as we filtered them out from the experiment data. As Table 3 shows, these following conclusions are drawn from the results:

- In all projects, the number of unclassified change messages is acceptable and the amount of NS (Not Sure) is about 14.12%. Therefore, we conclude that our approach automatically classifies most of the change messages which record the cause of the software changes.

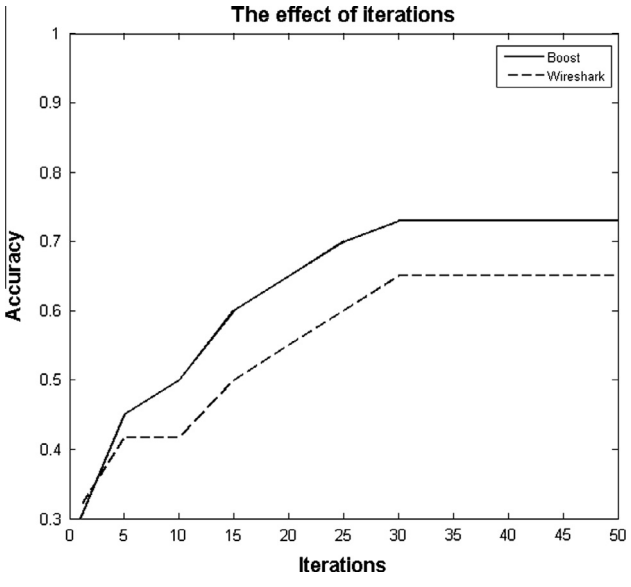

Fig. 3. The effect of iterations on classification accuracy (Boost, Wireshark).

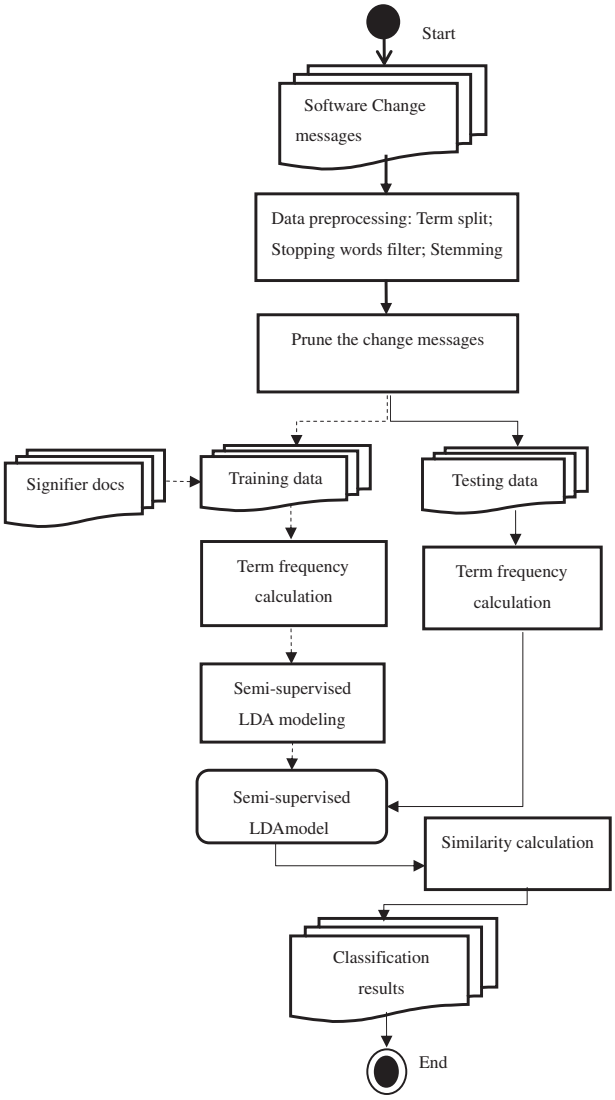

Fig. 4. The procedure of the semi-supervised experiment.

- As Table 3 shows, in Firebird and Python, the number of corrective changes is more than either adaptive or perfective changes and in Bugzilla the number of perfective changes is more than either corrective or adaptive changes. The phenomenon indicates that the activities of project debugging and correction accounted for more time than other development activities in Firebird and Python. According to the automatic classification results of projects, we found the cause of software evolution. For example, by tracing the release history of Bugzilla, we found that feature introduction and perfection were the main cause of Bugzilla evolution which matches the results in Table 3.
- In all projects, the total change messages which were classified were less than the actual number of change messages in the project. The reason is that messages that were too brief did not record the cause of the changes well and introduced noise to our approach. Hence, we filtered them out before classification. There is a little exception in the case of Firebird, the total messages of training and testing is 7675 less than the actual. After checking the data, we found that we filtered out 31,698 changes with the same description of "increment build number", in addition these messages were not entered by developers. They were caused by a repository–converter.

Table 3
Automatic classification results of semi-supervised LDA.

Projects	Corrective	Adaptive	Perfective	Not Sure	Total
Bugzilla	1486(37.35%)	830(20.86%)	1617(40.64%)	46(1.16%)	3979
Firebird	1408(36.70%)	998(26.01%)	605(15.76%)	826(21.53%)	3837
Python	5094(29.06%)	5054(28.83%)	3935(22.44%)	3449(19.67%)	17532

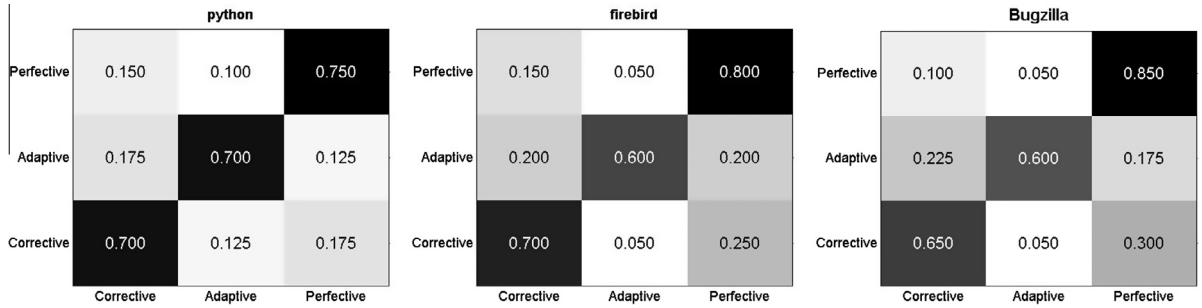

Fig. 5. The classification accuracy result of three projects (Python, Firebird, Bugzilla).

7. Validation

To evaluate our results, we did a survey with a small number of professional software developers who are working in different software domains. Our validation was constructed as follows:

- We made three lists of questions, each with 60 change descriptions of each project which were selected randomly by a program (20 of each category except the NS category).
- Three questionnaires for developers, each developer categorized one questionnaire.

In the procedure of generating the questionnaires, we randomly selected change messages to guarantee that the mathematical analysis performed later does not suffer from any bias due to their type or source.

7.1. Conducting the evaluation

Fig. 5 shows the agreement between developers and the automated classification method. In the survey, if a developer chooses two categories, a point was split between these categories. As Fig. 5 shows, the average agreement between developers and our automated classification method is 70.56%.

In order to evaluate more accurately, we took the F-measure value to evaluate our approach. The F-measure value considers both the precision (p) and the recall (r) of the classification to compute the value, where the p is the number of correct results divided by the number of all returned results and the r is the number of correct results divided by the number of results which should have been returned. The F-measure value is interpreted as a weighted average of the precision and recall and the higher score indicates a better classification result. Fig. 6 shows the F-measure value of the three projects.

$$F - measure = \frac{2 * (precision * recall)}{precision + recall} \quad (4)$$

7.2. Comparing to a lightweight approach

We used the term lists which was mentioned in Section 3.3.3 to simply implement the lightweight method mentioned by Hattori and Lanza [4], then it automatically classified each change commit into the extended Swanson's classification rules which we were used according to the first keywords found in the commit's comment. Here we call this method the first keyword method. We applied the first keyword method to automatically classify the test dataset, which was mentioned in Section 6, and calculated the agreement between the results of the survey, which we mentioned before, and the results of the first keyword method. As Table 4 shows, the average agreement between developers and the first keyword method is 50.56% and the amount of NS (Not Sure) is about 16.83%. Then we compared the classification accuracy of our method to first keyword method on the same dataset as shown in Fig. 7.

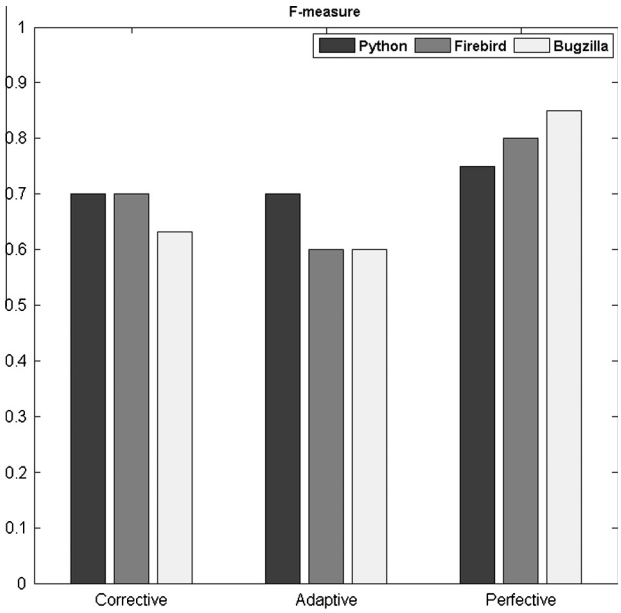

Fig. 6. The F-measure score between the professional manually labeled results and our automatic results on 3 open source projects.

7.3. Interpretation of the evaluation

The following conclusions are drawn from the results:

Table 4
Automatic classification results of first keyword method.

	Bugzilla (%)	Firebird (%)	Python (%)
Accuracy	41.67	53.33	56.67
NS (Not sure)	1.68	24.14	24.81

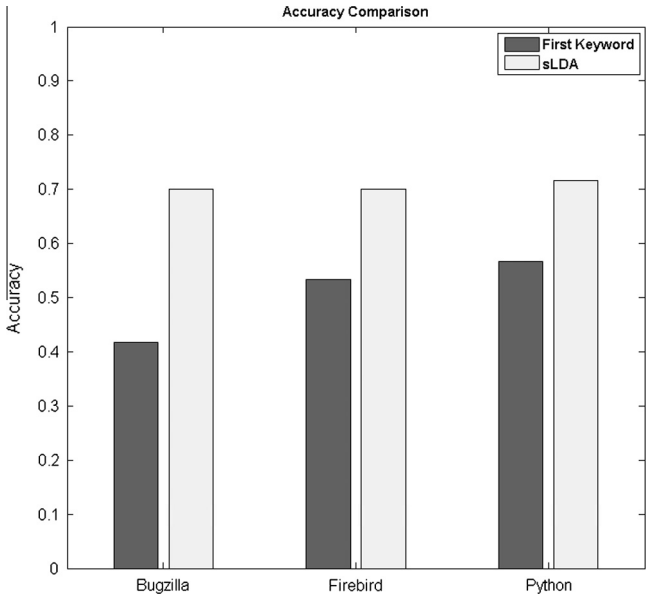

Fig. 7. The comparison of classification accuracy.

- The average agreement between the developers and our automatic classification method is 70.56% and the average F-measure value is about 0.7. These results indicate that our method is valid for cross-project analyzing software change messages.
- The amount of NS in our method is less than the first keyword method and the classification accuracy of our method is better than the first keyword method.
- The agreement for the perfective category is better than the other categories in all projects. It shows our semi-supervised LDA based classification method excelled at identifying the perfective maintenance tasks.
- The best results of our method and the first keyword method are in the Python project. By comparing and analyzing the descriptions of change message in the three projects which we surveyed, we found that the messages in Python are more regular and normative than others.

8. Conclusions

In this section we make a conclusion about our work and discuss the limitations and potential threats to the validity of our research.

8.1. Conclusions

In this paper, we studied change messages which are attached to software changes committed to a version control system. We presented a novel method to automatically classify change messages based on a semi-supervised LDA modeling using change messages description. The experimental results showed that our approach classified about 85% of the change messages. In addition, the validation surveys showed that 70.56% of the time our automatic classification results were in agreement with developer opinions. These show that the method is applicable to cross-project analysis of software change messages.

8.2. Limitation and threats to validity

The completeness of change messages. Attention needs to be paid to the problem of incompleteness and bias in the VCS [26]. Developers did not always make an explicit comment in the VCS when committing a change. We will explore how to cross-referencing bug tracker with VCS and use the corresponding relationships between change messages and bug reports to recover the missing change messages in our future work.

Quality of software change messages. As with most of software change analysis techniques based on the description of change messages, our results are dependent on the quality of the description style of the project developers. During our survey, we found that there were several change descriptions which cannot be categorized into any of the three categories. This was because most of these descriptions were irregularly written or there was little information indicating the reason for the change.

Preprocessing steps. We performed several preprocessing steps on the description of change messages, such as term splitting, filtering the stop words, and stemming. After checking the results of the preprocessing, we found a few of the words that were not precisely stemmed. This led to these words being classified into the wrong category or not being classified.

Parameter values. Our work involves choosing several parameters for the LDA model inputs, such as document smoothing parameters (α and β), the number of topics, and the number of sampling iterations. There is no theoretically guaranteed method for choosing optimal values for them currently, even though they will deeply affect the result of the research when using the LDA model. In our research, the number of topics K is set to 3 this corresponds to the software change classification labels, the values of α and β are the same as previous work [33,38], and we chose the number of iterations through experimentation.

Term list. The term lists, which is a key problem to our work, are derived from the help of Mauczka et al. [16]. The words in the list needed further editing and validation.

Validation method. Although we performed a survey with a small number of participants, we believe that their classifications are representative of developers in the industry. Nevertheless, it is desirable to enhance our work by having more participants to participate in the survey and for additional projects.

Acknowledgments

The authors sincerely thank the anonymous reviewers for their valuable comments that have led to the present improved version of the original manuscript.

The work described in this paper was partially supported by the National Natural Science Key Foundation (Grant No. 91118005), the National Natural Science Foundation of China (Grant No. 61173131), the Natural Science Foundation of Chongqing (Grant No. CSTS2010BB2061), Program for Changjiang Scholars and Innovative Research Team in University (Grant No. IRT1196) and the Fundamental Research Funds for the Central Universities (Grant Nos. CDJZR12098801 and CDJZR11095501).

References

[1] A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: Proceedings of International Conference on Software Maintenance (ICSE), 2000, IEEE, 2000, pp. 120–130.
[2] J. Kothari, T. Denton, A. Shokoufandeh, et al., Studying the evolution of software systems using change clusters, in: Proceedings of 14th IEEE

International Conference on Program Comprehension (ICPC), 2006, IEEE, pp. 46–55.
[3] L.C. Briand, V.R. Basili, A classification procedure for the effective management of changes during the maintenance process, in: Proceedings of the IEEE Conference on Software Maintenance (ICSE), 1992, IEEE, 1992, pp. 328–336.
[4] L.P. Hattori, M. Lanza, On the nature of commits, in: Proceedings of International Conference on Automated Software Engineering-Workshops (ASE), 2008, IEEE, 2008, pp. 63–71.
[5] A.E. Hassan, Automated classification of change messages in open source projects, in: Proceedings of International Conference on ACM symposium on Applied computing, 2008, ACM, 2008, pp. 837–841.
[6] A. Hindle, D.M. German, M.W. Godfrey, et al., Automatic classication of large changes into maintenance categories, in: Proceedings of International Conference on Program Comprehension (ICPC), 2009, IEEE, 2009, pp. 30–39.
[7] G. Antoniol, K. Ayari, M. Di Penta, et al., Is it a bug or an enhancement?: a text-based approach to classify change requests, in: Proceedings of the 2008 International Conference on the center for advanced studies on collaborative research: meeting of minds, 2008, ACM, 2008, pp. 304–318.
[8] L. Pollock, K. Vijay-Shanker, E. Hill, et al., Natural language-based software analyses and tools for software maintenance, in: Software Engineering, Springer, Berlin Heidelberg, 2013, pp. 94–125.
[9] A. Hindle, M.W. Godfrey, R.C. Holt, What's hot and what's not: windowed developer topic analysis, in: Proceedings of International Conference on Software Maintenance (ICSM), 2009, IEEE, 2009, pp. 339–348.
[10] T.H. Chen, S.W. Thomas, M. Nagappan, et al., Explaining software defects using topic models, in: Proceedings of International Conference on Mining Software Repositories (MSR), 2012, IEEE, 2012, pp. 189–198.
[11] T. Hofmann, Unsupervised learning by probabilistic latent semantic analysis, J. Mach. Learn. Res. 42 (1–2) (2001) 177–196.
[12] D.M. Blei, A.Y. Ng, M.I. Jordan, Latent dirichlet allocation, J. Mach. Learn. Res. 3 (2003) 993–1022.
[13] S.W. Thomas, Mining software repositories using topic models, in: Proceedings of the 33rd International Conference on Software Engineering (ICSE), 2011, ACM, 2011, pp. 1138–1139.
[14] A. Hindle, D.M. German, M.W. Godfrey, et al., Automatic classification of large changes into maintenance categories, in: Proceedings of the International Conference on Program Comprehension (ICPC), 2009, IEEE, 2009, pp. 30–39.
[15] S. Grant, J.R. Cordy, D.B. Skillicorn, Using topic models to support software maintenance, in: Proceedings of the International Conference on Software Maintenance and Reengineering (CSMR), 2012, IEEE, 2012, pp. 403–408.
[16] A. Mauczka, M. Huber, C. Schanes, W. Schramm, et al., Tracing your maintenance work – a cross-project validation of an automated classification dictionary for commit messages, in: Fundamental Approaches to Software Engineering, Springer, Berlin Heidelberg, 2012, pp. 301–315.
[17] A. Hindle, N.A. Ernst, M.W. Godfrey, et al., Automated topic naming to support cross-project analysis of software maintenance activities, in: Proceedings of the 8th Working Conference on Mining Software Repositories (MSR), 2011, ACM, 2011, pp. 163–172.
[18] D. Klein, S.D. Kamvar, C.D. Manning, From instance-level constraints to space-level constraints: making the most of prior knowledge in data clustering, in: Proceedings of the 19th International Conference on Machine Learning (ICML), 2002, ACM, 2002, pp. 307–314.
[19] M.S. Wu, J.T. Chien, A new topic-bridged model for transfer learning, in: Proceedings of the International Conference on Acoustics Speech and Signal Processing (ICASSP), 2010, IEEE, 2010, pp. 5346–5349.
[20] K. Somasundaram, G.C. Murphy, Automatic categorization of bug reports using latent dirichlet allocation, in: Proceedings of the 5th India Conference on Software Engineering (ICSE), 2012, ACM, 2012, pp. 125–130.
[21] E.B. Swanson, The dimensions of maintenance, in: Proceedings of the 2nd international conference on Software Engineering (ICSE), 1976, IEEE, 1976, pp. 492–497.
[22] T. Zimmermann, A. Zeller, P. Weissgerber, et al., Mining version histories to guide software changes, IEEE Trans. Software Eng. 31 (6) (2005) 429–445.
[23] R. Wu, H. Zhang, S. Kim, et al., Relink: recovering links between bugs and changes, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ACM, 2011, pp. 15–25.
[24] A. Sureka, S. Lai, L. Agarwal, Applying fellegi-sunter (fs) model for traceability link recovery between bug databases and version archives, in: Proceedings of the Asia Pacific conference on Software Engineering (SE), 2011, IEEE, 2011, pp. 146–153.
[25] A.T. Nguyen, T.T. Nguyen, H.A. Nguyen, et al., Multi-layered approach for recovering links between bug reports and fixes, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, pp. 63–74.
[26] A. Bachmann, C. Bird, F. Rahman, et al., The missing links: bugs and bug-fix commits, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2010, pp. 97–106.
[27] D. Blei, J. Lafferty, Correlated topic models, Adv. Neural Inform. Process. Syst. 18 (2006) 147–156.
[28] Iso I S O, IEC 9126-1: Software Engineering-Product Quality-Part 1: Quality Model, Geneva, Switzerland: International Organization for Standardization, 2001.
[29] E. Linstead, P. Rigor, S. Bajracharya, et al., Mining concepts from code with probabilistic topic models, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, 2007, pp. 461–464.
[30] Y. Liu, D. Poshyvanyk, R. Ferenc, et al., Modeling class cohesion as mixtures of latent topics, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2009, ICSM 2009, IEEE, 2009, pp. 233–242.
[31] H.U. Asuncion, A.U. Asuncion, R.N. Taylor, Software traceability with topic modeling, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), 2010, ACM, 2010, pp. 95–104.
[32] S. Dumais, G. Furnas, T. Landauer et al., Latent semantic indexing, in: Proceedings of the Text Retrieval Conference, 1999, TREC, 1995, pp. 219-230.
[33] T.L. Griffiths, M. Steyvers, Finding scientific topics, Proc. Natl. Acad. Sci. U.S.A. 101 (suppl. 1) (2004) 5228–5235.
[34] S. Grant, J.R. Cordy, Estimating the optimal number of latent concepts in source code analysis, in: Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation, 2010, SCAM, 2010, IEEE, 2010, pp. 65–74.
[35] H.M. Wallach, I. Murray, R. Salakhutdinov, et al., Evaluation methods for topic models, in: Proceedings of the 26th Annual International Conference on Machine Learning, 2009, ACM, 2009, pp. 1105–1112.
[36] Y. Lu, C. Zhai, Opinion integration through semi-supervised topic modeling, in: Proceedings of the 17th International Conference on World Wide Web, 2008, ACM, 2008, pp. 121–130.
[37] G. Salton, A. Wong, C.S. Yang, A vector space model for automatic indexing, Commun. ACM 18 (11) (1975) 613–620.
[38] A. De Lucia, M. Di Penta, R. Oliveto, et al., Using ir methods for labeling source code artifacts: is it worthwhile?, in: Proceedings of the International Conference on Program Comprehension (ICPC), 2012, IEEE, 2012, pp 193–202.